What's New in Frrraction

A quick survey of pre-0.9.9950

Originally there were just ratios: Pure Fractions n/d — an integer "stack" said to be Proper if the magnitude of n/d is less than 1, otherwise Improper). Then Mixed Fractions i + n/d — an integer plus a proper stack. All the arithmetic was oriented towards fractions. The only decimal number capability was confined to little frrrApps the user could create in the high-precision sections hp(...) of CF brackets. Frrraction's total focus was on *exact* arithmetic, not convenient approximations.

Eventually Decimal numbers got a toehold in the space normally occupied by fraction F1. The decimal was only allowed in F1int while F1's stack was forced to be 0/0. The indeterminate value in n/d showed that F1 was then totally defined by the decimal value in F1int. Only five things could be done with

A decimal number...

(1) ... could be entered from the key-strip. There was no decimal-point key so the decimal point was entered via a rapid double-tap on the 0-key while F1int was the active Fcell. The double-tap-0 was referenced in the documentation as "00".

(2) ... could be converted from decimal to fraction (from "flat" to "stack") by touching any Fcell except F1int. The resulting fraction was created by the standard conversion, superficially considered to be an exact representation of the decimal number: *i.f* —> *i* + *f* /*d* where *d* is "1" followed by the same number of zeros as *f* has digits, *e.g.* 3.23 —> 3 + 23/100.

(3) ... could be created in F1int from either of the two fractions F1 or F2 by executing 00 on any Fcell except F1int. As always, the decimal value was created in F1int and the F1-stack changed to 0/0.

(4) ... in F1int could be the target of the APRX function, which would place its best rational approximation — much better than the standard conversion — into fraction F2.

(5) The yellowFunction CF could convert back and forth between the decimal number in F1int or the n/d-stack in F1 or F2 and either its Continued Fraction [] or its Egyptian Fraction {} equivalent. For instance, 3.23 could be expressed as (a) continued fraction, (b) egyptian fraction:

(a) $3.23 = [3;4,2,1,7]$

$$= 3 + \cfrac{1}{4 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{7}}}}$$

(b) $3.23 = \{3;5,34,1700\}$

$$= 3 + \frac{1}{5} + \frac{1}{34} + \frac{1}{1700}$$

That's where things stood for quite a while. More recently, Frrraction has gotten serious about the decimal format.

An intro to post-0.9.9950

In a nutshell, since version 0.9.9950, Frrraction fully embraces decimals as well as ratios. For starters, Frrraction's key-strip now has a key dedicated to the decimal point and double-tap 00 is a thing of the past (although it still appears in the documentation since 00 is less awkward than • to read and write).

Integers and decimals

Decimal numbers look like 3.23 or 0.00323e+03 or 323.e-02 or 323e-02 (which all represent the same decimal number). They contain a decimal point or the letter "e" or both. All others are integers that look like 323 with no dot nor "e".

The “e” exponent notation multiplies the number by the indicated power of ten. It essentially shifts the position of the decimal point left (negative exponent) or right (positive exponent) by the indicated number of digit positions.

The backspace-delete key (above the Dot key on Frrraction’s key-strip) functions as usual on decimals as well as integers. If you backspace-delete past the decimal point, the Fcell reverts to integer.

Tapping the Dot key

- If you tap the Dot key, it **appends a decimal point** to the integer in the active Fcell if the Fcell did *not already have a decimal point*. It also changes the font-color to **blue**. Tapping digit keys or backspace-delete modifies the fractional portion of the decimal number.
- If the active Fcell is already decimal by virtue of having a decimal point *but no exponent* then tapping the Dot key **appends “e+0”** (which leaves the value of the number unchanged until tapping digit keys or backspace-delete modifies the exponent).
- If the active Fcell *already has an exponent* then tapping the Dot key **changes the sign of the exponent**. Y CHS changes the sign of the Fcell’s number, as usual.
- Several taps of the backspace-delete key can remove the exponent entirely if you wish. More b-del taps can remove the decimal point, reverting the number to an integer and font-color **black**.

Decimal arithmetic + – x ÷ (that’s + – x / on the key-strip)

A fraction is always a team by itself: Its numerator Fnum, denominator Fden, and integer part Fint cooperate inseparably in all operations. In Frrraction, the two visible fractions work as a pair: F1 and F2 are mates and at any time one is the active one which receives the results of various operations.

Similarly, decimals work in pairs: If the space occupied by either F1num or F2num presently holds a decimal each is the other’s mate and both are highlighted by blue font-color. Likewise F1den and F2den are mates, as are F1int and F2int, sharing font color in pairs.

If the active Fcell or its mate (or both) contains a decimal then the four arithmetic operations +add, -sub, *mul, and /div use ***decimal arithmetic*** to combine *just* the active Fcell and its mate — not the whole fractions. The active Fcell gets the result and its mate remains unchanged. The arithmetic result is always decimal so the font-color of the result and its mate is always blue. The UNDO gesture Y-RCL-Dot swaps back and forth between the original data and font-colors in the active Fcell and the result.

If neither the active Fcell nor its mate is decimal then the arithmetic ops perform ***fraction arithmetic*** — not decimal — on the two complete fractions. The after-the-decimal-point fractional parts of any decimals in both F1 and F2 are ignored in the arithmetic, and the result in the active fraction is composed entirely of three integers. The UNDO gesture swaps back and forth between the original contents of the active fraction and the op results all with the correct font-colors.

Frrraction still refers to all the Fcells by names like F1num and F2num even though, while decimal, they have no role as numerators. For convenience and to play down their ratio roles, all the fraction cells now have shortened nicknames: f1n, f2n, f1d,... *etc*. The nicknames are especially handy when writing frrrApps.

XCH and DUP

If the currently active Fcell contains an integer then XCH exchanges the entire fractions F1 ↔ F2. If the active Fcell currently contains a decimal then XCH exchanges only it with its mate, e.g. F1d ↔ F2d. In either case if you change your mind, XCH serves as its own UNDO function.

DUP copies active to mate analogously: whole fractions Factive → Fmate if active cell is an integer, or single cells like F2n → F1n if active cell is decimal. UNDO is set to restore the original *inactive* mate.

00c makes Frrraction into a major full-function calculator

There's been a big decimal enhancement in the CF hp(...) facility: The new 00c keyword opens the door to most of the classic C-language math.h functions that process decimals. Everything the big calculators offer and more (except graphing).

There are four groups of functions: Single-argument, double-argument, triple-argument, and no-argument. In Frrraction, the single-arg functions pull their argument from X (the top of the hp-stack) and push their result onto the stack into X, *e.g.* hp(100 00c log10) leaves X = $\log_{10}(100) = 2$.

> One argument: **X = f(X)**
> floor, ceil, nearbyint, rint, round, trunc, fabs, sqrt, cbrt, exp, exp2, __exp10, expm1, log, log2, log10, log1p,logb, cos, sin, tan, cosh, sinh, tanh, acos, asin, atan, acosh, asinh, atanh, erf, erfc, tgamma, lgamma, j0, j1, y0, y1, remainder, fdim, fmax, fmin, hypot, pow, atan2, fmod.

The two-arg functions pull their two double-precision arguments from X then from Y and push their result back into X, *e.g.* hp(2 3 00c pow) leaves $3^2 = 9$ in X. The function pow(x,y) is defined as x^y. Note that '2' got pushed initially into X, but '3' made room for itself in X by pushing '2' into Y, so that's how pow(X,Y) became pow(3,2) = $3^2 = 9$.

Note that there's an integer for one of the arguments of some of the functions. Also there's an address-of-double for the second argument of modf and an address-of-integer for frexp--just give the argument a slot in the Y-position of the stack; the function will leave the main result in X and computed value of the address-of argument in Y on the stack, *viz.* `hp( 0 12.34 00c frexp 00put f1n puts f1d )` puts 0.77125000 / 4 into fraction F1. Subsequently, `hp( get f1d get f1n 00c ldexp 00put f1i )` makes F1 = 12.340000 + 0.77125000 / 4.

> Two arguments: **X = f(X,Y)**
> copysign, nextafter, fmod, remainder, remquo, fdim, fmax, fmin, hypot, modf(d,&d), frexp(d,&i), ldexp(d,i), pow, soclbn(d,i), jn(i,d), yn(i,d), atan2, frrrdemo

There's only one three-arg function, a fast, accurate multiply-add x*y+z combination:

> Three arguments: **X = f(X,Y, Z)**
> fma

Each of the constants just pushes its value into X, *e.g.* hp(00c M_E) leaves the base $e \cong$ 2.718281828459044864 of the natural logarithm in X.

> No arguments: **X = constant**
> M_PI, M_E, M_LOG2E, M_LOG10E, M_LN2, M_LN10, M_PI_2, M_PI_4, M_1_PI, M_2_PI, M_2_SQRTPI, M_SQRT2, M_SQRT1_2

The functions in those four groups comprise without doubt the most famous collection in the world of computation, having been a central part of the C programming language and the Unix operating system from their inception on a Digital Equipment Corporation DEC PDP-11 since the 1970's. Locate their definitions (except for frrrdemo) on the web by Googling math.h.

The presence of math.h in Frrraction makes it easy to explore the functions' detailed behaviors. For example, have you ever made the effort to learn how the floor and ceiling functions handle negative numbers? Now there's no excuse not to do it:

Applet:	Result:

`<{{ hp( 3.45 dupx 00c floor`
`-3.45 dupx 00c floor`
`=T )`
`}}`
	X: -4
	Y: -3.45
	Z: 3
	T: 3.45

But...

Decimals are deceptive

When any Fcell contains a decimal, the cell no longer contains a truly unique number. Exact integer stuff like ratios, factors, common divisors, common multiples, and the like essentially become irrelevant. Frrraction's GCD function, for instance, has nothing meaningful to do because *any decimal is a factor of any decimal* — there is no least or greatest.

The problem is that decimals are generally not exact. They're not even intended to be exact.

For example if you're given the decimal 3.452 the exact number can be anywhere from 3.4515 (that's -5 in the *f*irst unspecified digit) up to almost 3.4525 (anything less than +5 in the first unspecified digit) — and it would still round off to the same 3.452. The denominators of exact fractions that represent numbers in that range have no upper limit. (There is a lower limit — it is 30 for the 3.452 example; determining that lower limit is the essence of APRX's job.)

The GCD function

If decimals are present in the active fraction's stack there is no "greatest common divisor". Rather than make an arbitrary assumption about how the user wants to handle decimals in the active fraction, GCD simply discards the .fraction portions of the decimals in the active fraction and works with the integer portions. So, actually, it *does* make an arbitrary assumption. If you didn't want the .fractions discarded, UNDO will recover the original contents for you.

GCD does one other thing if Frrraction is running in mixed-fraction mode: It displays an 8-digit decimal approximation of each fraction in the fraction's Hden cell.

The SIIF "Stacken or Flatten" function

The SIIF (Y Dot) function looks at the active fraction's specific active Fcell to decide what to do.

If the active Fcell is a decimal i.f then SIIF stacks the standard i+n/d ratio value of that decimal in the active fraction.

If the active Fcell is an integer then SIIF discards the fraction-portions of any decimals in the active fraction's other Fcells, flattens the resulting i+n/d ratio value into a single decimal in the active fraction's Fint cell, and sets n/d = 0/0.

In either case, UNDO will let you toggle back and forth between the original contents of the active fraction and the stacked or flattened result.

The ApPRoXimation function

The APRX function does either a precise match or a "kitchen approximation". The result replaces the original contents of the active fraction.

That **precise match** is one of the showcase features of Frrraction➢. Given a decimal (a flat number with decimal point and/or power-of-ten exponent), APRX counts the number of digits you gave in the specified number. It then locates — among all ratio fractions which match the specification to that

number of digits — the one with the smallest possible denominator. It compares five different methods in the computation and reports the best as the result.

The **kitchen approximation** restricts its search to finding the best among ratio fractions with the kitchen denominators 2, 3, 4, 6, 8, and 16. It compares the best two and uses the best to replace the original contents of the active fraction. That's **useful outside the kitchen, too**: Your high school physics instructor told you Planck's Constant was...what? He said 6.62606957e-34 and you didn't remember it five minutes later. But if he had given you APRX's kitchen version 6 $5/8$ (times 10^{-34}) then you might have had a chance — and it's only off by 0.1%

APRX looks at the contents of the active Fcell to decide whether to do precise or kitchen. If it contains a decimal point or exponent, it goes for precision. An integer gets you a kitchen simplification.

➢ Note: APRX's precise match is a vast improvement over the standard match used by most fraction programs. That standard conversion converts i.f into $i + f/10^n$ where n is simply the number of digits in f. For example, from 3.1415929 the standard produces 3 + 1,415,929/10,000,000. In contrast, APRX finds 3 + 16/113. (Full disclosure: I stumbled upon that example while using APRX to find an approximation for Pi that would be good to the first 8 digits of Pi: 3.1415927. It found 3 + 4,639/32,763. Then I gave it 3.1415928, and got 3 + 7,359/51,973 – much worse than for eight digits of Pi itself. But who could have guessed that going 0.0000001 larger would win the gold medal? I don't know who found it first, but the spectacular denominator 113 has been known since the 1600's as yielding the most efficient approximation for Pi. It is also what APRX yields if you request an approximation good for just 7 digits: 3.141593.

APRX present status

Now that Frrraction has gotten so cosy with decimals I've started looking at how APRX should adapt.

My first thought was to incorporate the power-of-ten multiplier notation. That was easy enough: I just made it strip off the "exx" suffix, run the old APRX, then tack the exx back on. One of the early examples I tested was 6.78e2 which became 6e2 + 7e2/9 whose 32-bit decimal is 6.7777778e2 or 677.77778. Hmmm, I thought. It shouldn't be that hard to approximate 6.78e2 to three digits; it is just 678 which is not only more accurate but also easier to get. So I changed APRX to look for the special cases where the given decimal is an integer in disguise. So far, so good: Given decimals like 1.23456e5 it now reliably gives the obvious accurate answer 123,456. It also does disguised-integer cases like 1.234e6 → 1,234,000. If the power-of-ten exponent is too large, it handles it gracefully, for example 1.234e10 → 1e10 + 11e10/47 whose 32-bit decimal form is 1.2340426e10, accurate to the specified four digits.

Now the question is: When does this stop being a good thing to do? By fiddling with exponents, any APRX problem has an integer solution exact to all the digits Frrraction can accept. For example, 3.14159265 is 314,159,265.e-8 which is an exact match to all nine given digits. But as an efficient ratio approximation is it anywhere near as good as 3+16/113 whose decimal version is off by 2 in the eighth digit. Of course not.

So. What's the future of this line of development?

1. I'm tempted to go back to never allowing exponents in the specification decimals for APRX to work with. Actually, though, APRX is all about how you present the numbers, so I think the best course will be for users to experiment until they get the effect they want in each circumstance.
2. Sometimes an Fcell contains a blue integer or a black decimal. I'm rooting those out as they appear.
3. To highlight the special arithmetic that applies to decimal mates, I'm thinking of making — no, I have forced — each Fcell and its mate have the same font color: blue if either is decimal, black only if both are integers. I think this should reduce the surprise sometimes caused by expecting Fcell-by-Fcell arithmetic or whole-fraction-by-whole-fraction arithmetic but actually getting the other. Luckily, UNDO will always restore the originals.

4. I'm thinking of using three different decimal font colors: maybe blue for Fnums, green for Fdens, and some other — maybe purple — for Fints. Especially now that mates always have matching colors. But I'm leaning towards not doing that, as it might add visual clutter without adding clarity.
5. I'm not pleased with the way GCD and SIIF deal with decimals. If decimals are present I'm thinking to have GCD make no change in the n/d stack, and revert to decimal arithmetic to compute i + n/d for the flattened display. Likewise, maybe SIIF should do that same kind of decimalized flattening computation if decimals are present. It really makes no sense to truncate the decimals to their integer parts then treat the result as though it had magically become an exact fraction.
6. CF functions are presently able to edit results and comments into the frrrNote, which sometimes become a nuisance to remove. They can also write them to the fShow area in the upper-right quarter of the screen--which evaporate as soon as you tap any key. I might provide a separate text field which the user can display or hide at will.
7. Presently Frrraction's "computed decisions" facility: SkipWhen <Fcell> <meets condition> only provides for skipping the very next bracket. This allows a very useful but limited if-then and an if-then-else execution. I want to strengthen it by adding a To <specific bracket>. The plan is to allow brackets to have identifying labels of the form #<n> with n between 0 and 9. For instance:

   @<{{ ... SkipWhen F2d NE0 To #7 }}> @<{{ ... }}> @<{{ ... }}> ... #7@<{{ ... }}> @<{{ ... }}>

   would have this effect: If F2den is not zero when the SkipWhen executes then the next bracket to execute will be the one labeled #7. All the intervening brackets including #7 will lose an enabler so, after #7 finishes, control will pass on to the next enabled brackets after #7.
8. When Frrraction got its own private file structure I didn't realize how important it would become. It's starting to seem too awkward to have to EDIT a file command (<Inl to reload a saved note, >Inl to store the present note, -Inl to delete a stored note, or ?Inl to show the list of stored notes) into the current fNote then Return-to-frrr then FILE to execute the command. I think I want to convert all that to a ListView so it looks and acts something like the list of reminders in iOS 7's Reminders app.

What else is new?

(1) As of version 0.9.9958 I changed Frrraction's opening screen to offer a more useful startup.

> For as long as I can remember, after your mandatory yellowKey tap, the easiest thing to do was to tap FILE and see a description of the most recent main change in the program.You can still see all those if you want to, by loading frrrFile l0l. But it's no longer the easiest thing to do.
>
> Now the easiest thing to do is to tap FILE which is preset in the frrrNote to load frrrFile l3l. File l 3l is an major example of a frrrApp. after you specify in F1int the number of terms to add to the sum then Ytap CF, it adds that many terms to Brouncker's continued fraction formula for Pi. Every 25 steps along the way, it tells its current estimate of Pi, as well as a peculiar cancellation of powers-of-ten it needs to do in order to avoid gigantic numbers in the intermediate steps.

(2) FrrrFile l1l has also been updated, making it a better guide to Frrraction's programmability.

(3) On the easy side: Frrraction needs a new •Dot button. And I liked the older more three-dimensional digit buttons. And the arithmetic buttons in the vertical part of the key-strip should be the same size as the digit buttons.

(4) Frrraction can now sometimes save your work if the system closes in the background, and offers to reload that job next time you run the app. I say "sometimes" because the system doesn't always deliver a warning before the closing.

(5) hp(...)'s cell manipulation facility has been enriched, allowing access to all Fcells and Hcells.

(6) I've added a faux-math.h function named frrrdemo and a frrrApp in file l6l which let you explore the way math.h handles decimal roundoff.

Appendix 1: LIST OF math.h FUNCTIONS

The following came from: https://developer.apple.com/library/mac/documentation/Darwin/Reference/Manpages/man3/math.3.html Similar descriptions are widely available on the internet by Googling for math.h.

Each of the functions that use floating-point values are provided by math.h in single, double, and extended precision; the double precision prototypes are listed here. The man pages for the individual functions provide more details on their use, special cases, and prototypes for their single and extended precision versions. In Frrraction the math.h facility is accessed using the hp keyword 00c. Frrraction commands are usually case-insensitive, but 00c is case-sensitive so the function names must be lowercase and the mathematical constants must be uppercase exactly as shown. The arguments are pulled from the hp stack and the results are pushed onto the hp stack, so 00c needs only the function name.
Example: <{{ hp(7 53 00c fmod =X) }} yields X: 4

double **copysign**(double, double)
double **nextafter**(double, double)

copysign(x, y) returns the value equal in magnitude to x with the sign of y. **nextafter**(x, y) returns the next floating-point number after x in the direction of y. Both are correctly-rounded.

Omitted from Frrraction: double **nan**(const char *tag)

The **nan**() function returns a quiet NaN, without raising the invalid flag.

double **ceil**(double)
double **floor**(double)
double **nearbyint**(double)
double **rint**(double) Rounded to integer based on current rounding mode: down, nearest, toward zero, or up
double **round**(double) Rounded to integer closest to x, e.g. 5.8 → 6, -5.2 → -5; tie → away from zero
Omitted from Frrraction: long int **lrint**(double)
Omitted from Frrraction: long int **lround**(double)
Omitted from Frrraction: long long int **llrint**(double)
Omitted from Frrraction: long long int **llround**(double)
double **trunc**(double) integer by simply discarding fraction, e.g. 5.8 → 5 and -5.8 → 5

These functions provide various means to round floating-point values to integral values. They are correctly rounded.

double **fmod**(double, double) gives true value of x (mod y) if x and y are both > 0
double **remainder**(double, double) peculiar, see below
double **remquo**(double x, double y, int *) peculiar like **remainder**

These return a remainder of the division of x by y with an integral quotient. **remquo**() additionally provides access to a few lower bits of the quotient. They are correctly rounded. r = x - q·y where (peculiarly) q is the integer closest to x/y

double **fdim**(double, double)
double **fmax**(double, double)
double **fmin**(double, double)

fmax(x, y) and **fmin**(x, y) return the maximum and minimum of x and y, respectively. **fdim**(x, y) returns the positive difference of x and y. All are correctly rounded.

double **fma**(double x, double y, double z)

fma(x, y, z) computes the value (x*y) + z as though without intermediate rounding. It is correctly rounded.

double **fabs**(double)
double **sqrt**(double)
double **cbrt**(double)
double **hypot**(double, double)

fabs(x), **sqrt**(x), and **cbrt**(x) return the absolute value, square root, and cube root of x, respectively.
hypot(x, y) returns sqrt(x*x + y*y). **fabs**() and **sqrt**() are correctly rounded.

double **exp**(double)
double **exp2**(double)
double **__exp10**(double)
double **expm1**(double)

exp(x), **exp2**(x), **__exp10**(x), and **expm1**(x) return e**x, 2**x, 10**x, and e**x - 1, respectively.

double **log**(double)
double **log2**(double)
double **log10**(double)
double **log1p**(double)

log(x), **log2**(x), and **log10**(x) return the natural, base-2, and base-10 logarithms of x, respectively.
log1p(x) returns the natural log of 1+x.

double **logb**(double)
Omitted from Frrraction: int **ilogb**(double)

logb(x) and **ilogb**(x) return the exponent of x.

double **modf**(double, double *) hp(456.7e-1 00c modf =Y) → X = 0.670000, Y = 45
double **frexp**(double, int *)

modf(x, &y) returns fractional part of x and stores integral part in y.
frexp(x, &n) returns the mantissa of x and stores the exponent in n. They are correctly rounded.

double **ldexp**(double, int)
double **scalbn**(double, int)
double **scalbln**(double, long int)

ldexp(x, n), **scalbn**(x, n), and **scalbln**(x, n) return x*2**n. They are correctly rounded.

double **pow**(double, double)

pow(x,y) returns x raised to the power y.
00c complies, but all other Frrraction powers and hp calculators do y to the power x

double **cos**(double)
double **sin**(double)
double **tan**(double)

cos(x), **sin**(x), and **tan**(x) return the cosine, sine and tangent of x, respectively. Note that x is interpreted as specifying an angle in radians.

double **cosh**(double)
double **sinh**(double)
double **tanh**(double)

cosh(x), **sinh**(x), and **tanh**(x) return the hyperbolic cosine, hyperbolic sine and hyperbolic tangent of x, respectively.

double **acos**(double)
double **asin**(double)
double **atan**(double)
double **atan2**(double, double)

acos(x), **asin**(x), and **atan**(x) return the inverse cosine, inverse sine and inverse tangent of x, respectively. Note that the result is an angle in radians. **atan2**(y, x) returns the inverse tangent of y/x in radians, with sign chosen according to the quadrant of (x,y).

double **acosh**(double)
double **asinh**(double)
double **atanh**(double)

acosh(x), **asinh**(x), and **atanh**(x) return the inverse hyperbolic cosine, inverse hyperbolic sine and inverse hyperbolic tangent of x, respectively.

double **tgamma**(double)
double **lgamma**(double)

tgamma(x) and **lgamma**(x) return the values of the gamma function and its logarithm evalutated at x, respectively. [Don't bother to look for this on your standard calculator!]

double **j0**(double)
double **j1**(double)
double **jn**(int, double)
double **y0**(double)
double **y1**(double)
double **yn**(int, double)

j0(x), **j1**(x), and **jn**(x) return the values of the zeroth, first, and nth Bessel function of the first kind evaluated at x, respectively. **y0**(x), **y1**(x), and **yn**(x) return the values of the zeroth, first, and nth Bessel function of the second kind evaluated at x, respectively. [Don't look for Bessel functions on your old calculator.]

double **erf**(double)
double **erfc**(double)

erf(x) and **erfc**(x) return the values of the error function and the complementary error function evaluated at x, respectively.

double **frrrdemo**(double, int)
frrrdemo(x,n) Version 1 returns x rounded according to the spec coded in n = 10c+r where r specifies the rounding mode: 0 ↔ downward, 1 ↔ to nearest integer, 2 ↔ toward zero, and 3 ↔ upward. c = 0 uses **rint** and c = 1 uses **round**. Example: n = 1 (r=1,c=0) and x = 6.37e1 yield 64 while x = 6.37 yields 6.

MATHEMATICAL CONSTANTS

CONSTANT	VALUE
M_E	base of natural logarithm, e
M_LOG2E	log2(e)
M_LOG10E	log10(e)
M_LN2	ln(2)
M_LN10	ln(10)
M_PI	pi
M_PI_2	pi / 2
M_PI_4	pi / 4
M_1_PI	1 / pi
M_2_PI	2 / pi
M_2_SQRTPI	2 / sqrt(pi)
M_SQRT2	sqrt(2)
M_SQRT1_2	sqrt(1/2)

Appendix 2: A Pre-math.h CF App

This example uses a beautiful continued fraction expansion to calculate Pi

Around the year 1650, a man destined to be the first president of The Royal Philosophical Society of London

Pi by cont'd frac'ns l29lv6
Here's math.h Pi to 16 digits:
{{ hp(00c M_PI =X) }}
X: 3.141592653589793

F1num = 5001 got Pi ~= 3.1412 (four digits) in about 26 minutes on iPad mini Model ME277LL/A (using >98% of the dual core 1.3 GHz Apple A7 cpu)
Avg(501,503)=3.1415848 (off by 1 in digit 6) took only about 7 minutes!
Avg(1001,1003)=3.1415906,
exact to 6 digits, rounds to be off by only 2 in the 7th digit.

S2 accumulates 4/Pi
S3 is to restore original n
{{ F1num MakeOdd
hp(2 sto 2
get f1n sto 3) }}>

S0 is short-loop counter
S1 decrements n
•<{{ hp(get f1n sto 1
get f1i sto 0) }}>

@<{{ hp(
rcl 1 dupx 00*
rcl 2 00/
2 00+
sto 2
) }}>

@<{{ hp(rcl 1 2 - dupx sto 1 put f1n)
SkipWhen F1n LT0 }}>

Stop closing the loop when n goes negative
@<{{ @ }}

Unwind 4/Pi to Pi
@<{{ hp(rcl 2 1 00-
1 xchxy 00/
4 00*
00put f1d
rcl 3 put f1n
) }}

Appendix 2: An Example CF app

This example adjusts numeric precision

Internally, Frrraction's APRX does something like this to compare ratio fractions with the user-specified target in search of the best match. Without the recent addition of math.h, Frrraction's brackets could not accomplish the same thing. Now they can. Here's one that lets you specify in F1int the number of digits of precision you want to use for representing the decimal in F1num.

The method is to normalize x = i.f by shifting the decimal point left or right to obtain the form 0.f where the fraction part f itself has no leading 0's. Then multiply by 10^n to make the leading n digits become the integer part. Then use the math.h round function to eliminate the fractional part. Then divide by 10^n to get n digits back into normalized form. Then de-normalize to the desired result by shifting the decimal point back to the position originally specified in F1num by the user.

(Note: Shifting the decimal point of a number can be achieved by changing the integer part of the base-10 logarithm of the decimal number.)

```
Keep n digits of x  |24|v3
( restricted to 1 <= n <= 8 )
Setup: F1 = n+x/0
Result: n-digit f2n

Save the requested precision n
as 10-to-the-n in S1
@<{{ hp( get f1i
00c __exp10 sto 1



put/- 0: x-/ h1n
put/- 0: n-/ h1i ) }}>

Set to normalize x, putting
int( log(x)) into S2
@<{{ hp( get f1n
00c log10 dupx
00put f1d



    put/- 0: log x-/ h1d
sto 2 )
SkipWhen f1den LT0 }}>



    Adjust if log x >= 0 i.e. x >= 1
    @<{{ hp( get f1d 1 00+
    sto 2 ) }}>
@<{{ hp( rcl 2
00c trunc sto 2
```

|24| is an inactive FILE command

Normal parentheses ( ) are not brackets

Text not enclosed by{{}} or | | or [ ] or { } is just text

00c __exp10 computes 10^X into X

sto <n> stores value into hp storage Sn (0 <= n <= 9)
Sn holds a decimal or an integer;
not the same as frrr's Fn which stores a ratio fraction.

put/- <text> -/ <cell> displays text in Fcell or Hcell

h1n and h1i are two of the Hcells (H for help)

f1n is an Fcell (nickname for F1num) (F for Fraction)

00c log10 computes $\log_{10}(X)$ into X

Dupx is needed for the display copy

00put <cell> stores decimal into named F- or Hcell

In this case, f1d is $\log_{10}(x)$ so is negative if x has no integer part, i.*e.* x = i.f with i = 0.

Causes next bracket to be skipped if F1den < 0

Its <enabler gets removed though, and its >nopause (continue without pause) gets obeyed if present.

Do you see why this is not necessary when $\log_{10}(x) < 0$?

In either case, this grabs just the integer part...

... and saves it in S2.

```
 ) }}>

Normalize x
S2 holds int of log
@<{{ hp( get f1n
00c log10
rcl 2 00-
00c __exp10 dupx
00put f2i
    put/- 1: norm'd-/ h2i
sto 3 ) }}>


Round the norm'd x by making
the first n digits comprise
the integer part of a number
then discarding the fractional part.
(remember, S1 contains 10^n)
@<{{ hp( rcl 3 rcl 1 00*
00c round
rcl 1 00/ dupx
00put f2d
     put/- 2: round norm-/ h2d
sto 3
 ) }}>

Denormalize result
@<{{ hp( rcl 3 00c log10
rcl 2 + 00c __exp10
00put f2n
     put/- 3: result-/ h2n
 ) @}}
```

Remember, F1num holds the user's decimal x
This...
...accomplishes...
...the normalization of x
This displays it in F2int
and this informs the user
Needed since hp stack is re-zeroed between brackets
(We separated into brackets so between-bracket text could be inserted for explanatory documentation.)

Rcl 3 to recover norm'd x from the previous bracket.

Another sto 3 and...

... rcl 3 maneuver required for the
... Denormalize result
...between-brackets documentation text

the inside-@ re-enables all the above @{{ to @<{{
ready for the next yTap of CF for a new x-value.