The MINIFP package*

Dan Luecking

2013/05/28

Abstract

This package provides minimal fixed point exact decimal arithmetic operations. 'Minimal' means numbers are limited to eight digits on either side of the decimal point. 'Exact' means that when a number *can* be represented exactly within those limits, it will be.

Contents

*This file has version number v0.95., last revised 2013/05/28. The code described here was developed by Dan Luecking.

1 Introduction

In working on an application that needed to be able to automatically generate numeric labels on the axes of a graph, I needed to be able to make simple calculations with real numbers. What TEX provides is far too limited. In fact, its only native user-level support for real numbers is as factors for dimensions. For example one can "multiply" 3.1×0.2 by `\dimen0=0.2pt \dimen0=3.1\dimen0` .

Unfortunately TEX stores dimensions as integer multiples of the "scaled points" (`sp`) with $\texttt{sp} = 2^{-16}\texttt{pt}$, and therefore `.2pt` is approximated by $\frac{13107}{65536}$, which is not exact. Then mutiplying by 3.1 produces $\frac{40631}{65536}$. If we ask TEX to display this, it produces 0.61998`pt` and not the exact value 0.62. This is sufficiently accurate for positioning elements on a page, but not for displaying automatically computed axis labels if five digit accuracy is needed.

The MINIFP package was written to provide the necessary calculations with the necessary accuracy for this application. The implementation would have been an order of magnitude smaller and faster if only four digit accuracy were provided (and I may eventually do that for the application under consideration), but I have decided to clean up what I have produced and release it as is. The full MINIFP package provides nearly the same operations as a subset of the FP package, but the latter carries calculations to 18 decimal places, which is far more than necessary for my purposes. I want something small and fast to embed in the MFPIC drawing package.

I decided on eight digits on both sides of the decimal point essentially because I wanted at least five digits and the design I chose made multiples of four the easiest to work with.

MINIFP also provides a simple stack-based language for writing assembly language-like programs. Originally, this was to be the native calculation method, but it turned out to be too unwieldy for ordinary use. I left it in because it adds only about 10% overhead to the code.

But why *only* eight digits? TEX only works with integers, and since the maximum integer allowed is about 2 000 000 000, the largest numbers that can be added are limited to about 999 999 999. It is very little trouble to add numbers by adding their fractional parts and integer parts separately as 9-digit integers. So it would seem multiples of 9 digits would be easy to implement.

However, something we have to do repeatedly in *division* is multiply the integer and fractional parts of a number by a one-digit number. For that purpose, nine digits would be too much, but eight digits is just right. For nine digits, we would have to inconveniently break the number into more than two parts. Limiting our numbers to eight-digit parts drastically simplifies division.

Another simplification: multiplication has to be done by breaking the number into parts. TEX can multiply any two 4-digit integers without overflow, but it cannot multiply most pairs of 5-digit integers. Two 8-digit numbers conveniently break into four 4-digit parts. To get even nine digits of accuracy would require six parts (five, if we don't insist on a separation occuring at the decimal point). The complexity of the multiplication process goes up as the square of the number of parts, so six parts would more than double the complexity.

A final simplification: TEX places a limit of nine on the number of arguments a macro can have. Quite often the last argument is needed to clear out unused text to be discarded. Thus, a string of eight digits can quite often be processed with one execution of one nine-argument macro.

Addition and subtraction can be exact, multiplication and division can extend numbers past the 8-digit limit so they might be rounded. However, when the exact answer fits in the 8-digit limit, our code should produce it. Overflow (in the sense that the integer part can exceed the allowed eight digits) is always possible, but is much more likely with multiplication and division.

Multiplication is carried out internally to an exact answer, with 16 digits on each side of the decimal point. The underflow digits (places 9 through 16 after the decimal point) are used to round to an 8-digit result. Overflow digits (those to the left of the lowest 8 in the integer part) are discarded, usually without warning. Division is internally carried to nine digits after the decimal, which is then also rounded to an 8-digit result. Overflow digits are ignored for division also.

We supply two kinds of operations in this package. There are stack-based operations, in which the operands are *popped* from a stack and the results *pushed* onto it, and argument-based, in which the operands (and a macro to hold the result[1]) are arguments of a macro. Both types load the arguments into internal macros (think of them as "registers"), then call internal commands (think "microcode") which return the results in internal macros. These results are then *pushed* onto the stack (stack-based operations) or stored in a supplied macro argument (think "variable"). The difference lies entirely in where the operands come from (arguments or stack) and where they go (macro or stack).

The stack is implemented as an internal macro which is redefined with each command. The binary operations act on the last two *pushed* objects in the order they were *pushed*. For example, the sequence "*push* 5, *push* 3, *subtract*" performs $5 - 3$ by popping 3 and 5 into registers (thereby removing them from the stack), subtracting them and then pushing the result (2) onto the stack.

Our implementation of the *push* operation first prepares the number in a standard form. Thus, stack-based operations always obtain numbers in this form. The argument based operations will prepare the arguments in the same way. The internal commands will thus have a standard form to operate on. All results are returned in standard form.

The standard form referred to above is an integer part (one to eight digits with no unnecessary leading zeros nor unnecessary sign) followed by the decimal point (always a dot, which is ASCII 46), followed by exactly eight digits, all of this preceded by a minus sign if the number is negative. Thus, $--0.25$ would be processed and stored as "`0.25000000`" and $-.333333$ as "`-0.33333300`".

[1] Unlike most other packages for decimal arithmetic, MINIFP puts the macro to hold the result last. This allows the calculation to be performed before the macro is even read, and this makes it somewhat easier for the stack- and argument-based versions to share code.

2 User macros

MINIFP provides (so far) six binary operations (that act on a pair of numbers): addition, subtraction, multiplication, division, maximum and minimum, as well as fourteen unary operations (that act on one number): negation, absolute value, doubling, halving, integer part, fractional part, floor, ceiling, signum, squaring, increment, decrement and inversion. With the "`extra`" option, the unary operations sine, cosine, logarithm, powers and square root are available, and the binary operation angle. See section 4.

These extra operations are made available using the `extra` option in LaTeX:

```
\usepackage[extra]{minifp}
```

In plain TeX, they will be loaded if you give the macro `\MFPextra` a definition (any definition) before inputting `minifp.sty`:

```
\def\MFPextra{} \input minifp.sty
```

The extras can also be loaded by means of the command `\MFPloadextra`, issued after `minifp.sty` is loaded. As of version 0.95 `mfpextra` can be directly `\input`. It will detect whether `minifp.sty` has been loaded and input it if not. This will work only in plain TeX.

If the extra operations are not needed, some memory and time might be saved by using `minifp.sty` alone. I have not seriously tried to keep `mfpextra.tex` as small or fast as possible, but I do try to improve the accuracy when I can.

As previously mentioned, each of these operations come in two versions: a version that acts on operands and stores the result in a macro, and a version that acts on the stack. The former all have names that begin `\MFP` and the latter begin with `\R`. The former can be used anywhere, while the latter can only be used in a "program". A program is started with `\startMFPprogram` and terminated with `\stopMFPprogram`. The `R` in the names is for 'real'. This is because it is possible that stacks of other types will be implemented in the future.

For example, `\MFPadd{1.2}{3.4}\X` will add 1.20000000 to 3.40000000 and then define `\X` to be the resulting `4.60000000`. These operand forms do not alter or even address the stack in any way. The stack-based version of the same operation would look like the following:

```
\Rpush{1.2}
\Rpush{3.4}
\Radd
\Rpop\X
```

which would *push* first `1.20000000` then `3.40000000` onto the stack, then replace them with `4.60000000`, then remove that and store it in `\X`. Clearly the stack is intended for calculations that produce a lot of intermediate values and only the final result needs to be stored.

The command `\startMFPprogram` starts a group. That group should be ended by `\stopMFPprogram`. Changes to the stack and defined macros are local to that group. Thus the macro `\X` in the example above might seem to be useful only as a temporary storage for later calculations in the same program group. However, there are commands provided to force such a macro to survive the group, and

even to force the contents of the stack to survive the group (see the end of subsection 2.3). Do not try to turn a MINIFP program into a LaTeX environment. The extra grouping added by environments would defeat the effects of these commands.

2.1 Nonstack-based operations

In the following tables, an argument designated ⟨*num*⟩ can be any decimal real number with at most 8 digits on each side of the decimal point, or it can be a macro that contains such a number. If the decimal dot is absent, the fractional part will be taken to be zero, if the integer part or the fractional part is absent, it will be taken to be zero. (One consequence of these rules is that all the following arguments produce the same internal representation of zero: `{0.0}`, `{0.}`, `{.0}`, `{0}`, `{.}`, and `{}`.) Spaces may appear anywhere in the ⟨*num*⟩ arguments and are stripped out before the number is used. For example, `{3 .  1415 9265}` is a valid argument. Commas are not permitted. The decimal dot (period, fullstop) character must be inactivated if some babel language makes it a shorthand.

The `\macro` argument is any legal macro. The result of using one of these commands is that the macro is defined (or redefined, there is no checking done) to contain the standard form of the result. If the ⟨*num*⟩ is a macro, the braces surrounding it are optional.

Binary Operations

Command	**operation**
`\MFPadd{`⟨*num*$_1$⟩`}{`⟨*num*$_2$⟩`}\macro`	Stores the result of ⟨*num*$_1$⟩ + ⟨*num*$_2$⟩ in `\macro`
`\MFPsub{`⟨*num*$_1$⟩`}{`⟨*num*$_2$⟩`}\macro`	Stores the result of ⟨*num*$_1$⟩ − ⟨*num*$_2$⟩ in `\macro`
`\MFPmul{`⟨*num*$_1$⟩`}{`⟨*num*$_2$⟩`}\macro`	Stores the result of ⟨*num*$_1$⟩ × ⟨*num*$_2$⟩, rounded to 8 places after the decimal point, in `\macro`
`\MFPmpy{`⟨*num*$_1$⟩`}{`⟨*num*$_2$⟩`}\macro`	Same as `\MFPmul`
`\MFPdiv{`⟨*num*$_1$⟩`}{`⟨*num*$_2$⟩`}\macro`	Stores the result of ⟨*num*$_1$⟩/⟨*num*$_2$⟩, rounded to 8 places after the decimal point, in `\macro`
`\MFPmin{`⟨*num*$_1$⟩`}{`⟨*num*$_2$⟩`}\macro`	Stores the smaller of ⟨*num*$_1$⟩ and ⟨*num*$_2$⟩ in `\macro`
`\MFPmax{`⟨*num*$_1$⟩`}{`⟨*num*$_2$⟩`}\macro`	Stores the larger of ⟨*num*$_1$⟩ and ⟨*num*$_2$⟩ in `\macro`

Unary Operations

Command	operation
`\MFPchs{`⟨*num*⟩`}\macro`	Stores $-\langle num\rangle$ in `\macro`.
`\MFPabs{`⟨*num*⟩`}\macro`	Stores $\lvert\langle num\rangle\rvert$ in `\macro`.
`\MFPdbl{`⟨*num*⟩`}\macro`	Stores $2 \times \langle num\rangle$ in `\macro`.
`\MFPhalve{`⟨*num*⟩`}\macro`	Stores $\langle num\rangle/2$, rounded to 8 places after the decimal point, in `\macro`.
`\MFPint{`⟨*num*⟩`}\macro`	Replaces the part of ⟨*num*⟩ after the decimal point with zeros (keeps the sign unless the result is zero) and stores the result in `\macro`.
`\MFPfrac{`⟨*num*⟩`}\macro`	Replaces the part of ⟨*num*⟩ before the decimal point with zero (keeps the sign unless the result is zero) and stores the result in `\macro`.
`\MFPfloor{`⟨*num*⟩`}\macro`	Stores the largest integer not more than ⟨*num*⟩ in `\macro`.
`\MFPceil{`⟨*num*⟩`}\macro`	Stores the smallest integer not less than ⟨*num*⟩ in `\macro`.
`\MFPsgn{`⟨*num*⟩`}\macro`	Stores -1, 0 or 1 (in standard form) in `\macro` according to whether ⟨*num*⟩ is negative, zero, or positive.
`\MFPsq{`⟨*num*⟩`}\macro`	Stores the square of ⟨*num*⟩ in `\macro`.
`\MFPinv{`⟨*num*⟩`}\macro`	Stores $1/\langle num\rangle$, rounded to 8 places after the decimal point, in `\macro`.
`\MFPincr{`⟨*num*⟩`}\macro`	Stores $\langle num\rangle + 1$ in `\macro`.
`\MFPdecr{`⟨*num*⟩`}\macro`	Stores $\langle num\rangle - 1$ in `\macro`.
`\MFPzero{`⟨*num*⟩`}\macro`	Ignores ⟨*num*⟩ and stores 0.00000000 in the `\macro`.
`\MFPstore{`⟨*num*⟩`}\macro`	Stores the ⟨*num*⟩, converted to standard form, in `\macro`

The command `\MFPzero` is useful for "macro programs". If you want to do something to a number depending on the outcome of a test, you may occasionally want to simply absorbed the number and output a default result. This is more efficient than multiplying by zero (but less efficient than simply defining the `\macro` to be zero.)

Note that one could easily double, halve, square, increment, decrement or invert a ⟨*num*⟩ using the binary versions of `\MFPadd`, `\MFPsub`, `\MFPmul` or `\MFPdiv`. The commands `\MFPdbl`, `\MFPhalve`, `\MFPsq`, `\MFPincr`, `\MFPdecr` and `\MFPinv` are designed to be more efficient versions, since they are used repeatedly in internal code.

Also, multiplication is far more efficient than division, so even if you use the two argument versions, `\MFPmul{`⟨*num*⟩`}{.5}` is faster than `\MFPdiv{`⟨*num*⟩`}{2}`.

There is one command that takes no argument and returns no value:

Do Nothing

Command	operation
`\MFPnoop`	Does nothing.

The following are not commands at all, but macros that contain convenient constants.

Constants

Constant	**value**
`\MFPpi`	`3.14159265`, the eight-digit approximation to π.
`\MFPe`	`2.71828183`, the eight-digit approximation to e.
`\MFPphi`	`1.61803399`, the eight-digit approximation to the golden ratio ϕ.

There also exist commands to check the sign of a number and the relative size of two numbers:

`\MFPchk{`⟨*num*⟩`}`
`\MFPcmp{`⟨num_1⟩`}{`⟨num_2⟩`}`

These influence the behavior of six commands:

`\IFneg{`⟨*true text*⟩`}{`⟨*false text*⟩`}`
`\IFzero{`⟨*true text*⟩`}{`⟨*false text*⟩`}`
`\IFpos{`⟨*true text*⟩`}{`⟨*false text*⟩`}`
`\IFlt{`⟨*true text*⟩`}{`⟨*false text*⟩`}`
`\IFeq{`⟨*true text*⟩`}{`⟨*false text*⟩`}`
`\IFgt{`⟨*true text*⟩`}{`⟨*false text*⟩`}`

Issuing `\MFPchk{\X}` will check the sign of the number stored in the macro `\X`. Then `\IFneg{A}{B}` will produce '`A`' if it is negative and '`B`' if it is zero or positive. Similarly, `\MFPcmp{\X}{1}` will compare the number stored in `\X` to 1. Afterward, `\IFlt{A}{B}` will produce '`A`' if `\X` is less than 1 and '`B`' if `\X` is equal to or greater than 1.

If users finds it tiresome to type two separate commands, they can easily define a single command that both checks a value and runs `\IF...`. For example

```
\def\IFisneg#1{\MFPchk{#1}\IFneg}
```

Used like

```
\IFisneg{\X}{A}{B}
```

this will check the value of `\X` and run `\IFneg` on the pair of alternatives that follow.

The user might never need to use `\MFPchk` because every one of the operators provided by the MINIFP package runs an internal version of `\MFPchk` on the result of the operation before storing it in the `\macro`. For example, after `\MFPzero` the command `\IFzero` will always return the first argument. For this reason one should not insert any MINIFP operations between a check/compare and the `\IF...` command that depends on it.

Note: the behavior of all six `\IF...` commands is influenced by *both* `\MFPchk` and `\MFPcmp`. This is because internally `\MFPchk{\X}` (for example) and `\MFPcmp{\X}{0}` do essentially the same thing. In fact there are only three internal booleans that govern the behavior of the six `\IF...` commands. The different names are for clarity: `\IFgt` after a compare is less confusing than the entirely equivalent `\IFpos`.

It should probably be pointed out that the settings for the `\IF...` macros are local to any TEX groups they are contained in.

2.2 Commands to process numbers for printing

After `\MFPadd{1}{2}\X` one can use `\X` anywhere and get 3.00000000. One might may well prefer 3.0, and so commands are provided to truncate a number or round it to some number of decimal places. Note: these are provided for printing and they will not invoke the above `\MFPchk`. They do not have any stack-based versions. The commands are

`\MFPtruncate{`⟨*int*⟩`}{`⟨*num*⟩`}\macro`
`\MFPround{`⟨*int*⟩`}{`⟨*num*⟩`}\macro`
`\MFPstrip{`⟨*num*⟩`}\macro`

where ⟨*int*⟩ is a whole number between −8 and 8 (inclusive). The other two arguments are as before.

These commands merely process ⟨*num*⟩ and define `\macro` to produce a truncated or rounded version, or one stripped of trailing zeros, or one with added trailing zeros. Note that truncating or rounding a number to a number of digits greater than it already has will actually lengthen it with added zeros. For example, `\MFPround{4}{3.14159}\X` will cause `\X` to be defined to contain `3.1416`, while `\MFPround{6}{3.14159}\X` will cause `\X` to contain `3.141590`. If `\Y` contains `3.14159`, then `\MFPtruncate{4}\Y\Y` will redefine `\Y` to contain `3.1415`. Also, `\MFPstrip{1.20000000}\Z` will cause `\Z` to contain `1.2`. All these commands first normalize the ⟨*num*⟩, so any spaces are removed and redundant signs are discarded.

If ⟨*int*⟩ is negative, places are counted to the left of the decimal point and 0 s are substituted for lower order digits. That is, `\MFPtruncate{-2}{1864.3}\X` will give `\X` the value `1800` and `\MFPround{-2}{1864}\X` will give `\X` the value `1900`.

If the first argument of `\MFPround` or `\MFPtruncate` is zero or negative then the dot is also omitted from the result. If `\MFPstrip` is applied to a number with all zeros after the dot, then one 0 is retained. There is a star form where the dot and the zero are dropped.

For these three commands, the sign of the number is irrelevant. That is, the results for negative numbers are the negatives of the results for the absolute values. The processing will remove redundant signs along with redundant leading zeros: `\MFPtruncate{-3}{-+123.456}` will produce `0`. The rounding rule is as follows: round up if the digit to the right of the rounding point is 5 or more, round down if the digit is 4 or less.

2.3 Stack-based macros

The stack-based macros can only be used in a MINIFP program group. This group is started by the command `\startMFPprogram` and ended by `\stopMFPprogram`. None of the stack-based macros takes an argument, but merely operate on values on the stack, replacing them with the results. There are also commands to manipulate the stack and save a value on the stack into a macro. Finally, since all changes to the stack (and to macros) are local and therefore lost after `\stopMFPprogram`, there are commands to selectively cause them to be retained.

To place numbers on the stack we have `\Rpush` and to get them off we have

`\Rpop`. The syntax is

`\Rpush{`⟨*num*⟩`}`
`\Rpop\macro`

The first will preprocess the ⟨*num*⟩ as previously discussed and put it on the stack, the second will remove the last number from the stack and define the given macro to have that number as its definition.

All the binary operations remove the last two numbers from the stack, operate on them in the order they were put on the stack, and *push* the result on the stack. Thus the program

```
\Rpush{1.2}
\Rpush{3.4}
\Rsub
```

will first put `1.20000000` and `3.40000000` on the stack and then replace them with `-2.20000000`. Note the order: $1.2 - 3.4$.

Binary Operations

Command	**operation**
`\Radd`	Adds the last two numbers on the stack.
`\Rsub`	Subtracts the last two numbers on the stack.
`\Rmul`	Multiplies the last two numbers on the stack, rounding to 8 decimal places.
`\Rmpy`	Same as `\Rmul`.
`\Rdiv`	Divides the last two numbers on the stack, rounding to 8 decimal places.
`\Rmin`	Replaces the last two elements on the stack with the smaller one.
`\Rmax`	Replaces the last two elements on the stack with the larger one.

The unary operations replace the last number on the stack with the result of the operation performed on it.

Unary Operations

Command	operation
`\Rchs`	Changes the sign.
`\Rabs`	Obtains the absolute value.
`\Rdbl`	Doubles the value.
`\Rhalve`	Halves the value, rounding to 8 places.
`\Rint`	Replaces the fractional part with zeros. If the result equals 0.0, any negative sign will be dropped.
`\Rfrac`	Replaces the integer part with `0`. If the result equals 0.0, any negative sign will be dropped.
`\Rfloor`	Obtains the largest integer not greater than the number.
`\Rceil`	Obtains the smallest integer not less than the number.
`\Rsgn`	Obtains -1, 0 or 1 according to whether the number is negative, zero, or positive. These numbers are pushed onto the stack with the usual decimal point followed by 8 zeros.
`\Rsq`	Obtains the square. Slightly more efficient than the equivalent `\Rdup\Rmul`. See below for `\Rdup`.
`\Rinv`	Obtains the reciprocal. Slightly more efficient than the equivalent division.
`\Rincr`	Increases by 1. Slightly more efficient than the equivalent addition.
`\Rdecr`	Decreases by 1. Slightly more efficient than the equivalent subtraction.
`\Rzero`	Replaces the number with zero. Slightly more convenient than the equivalent `\Rpop\X` followed by a `\Rpush{0}`.

There is one operation, which does not read the stack nor change it (nor do anything else).

Do Nothing

Command	operation
`\Rnoop`	Does nothing.

There also exist commands to check the sign of the last number, and the relative size of the last two numbers on the stack:

```
\Rchk
\Rcmp
```

They do not remove anything from the stack. Just like the nonstack counterparts, they influence the behavior of six commands: `\IFneg`, `\IFzero`, `\IFpos`, `\IFlt`, `\IFeq` and `\IFgt`. Issuing `\Rchk` will check the sign of the last number on the stack, while `\Rcmp` will compare the last two in the order they were pushed. For example, in the following

```
\Rpush{1.3}
\Rpush{-2.3}
\Rcmp
\IFgt{\Radd}{\Rsub}
\Rpush\X
```

```
\Rchk
\IFneg{\Radd}{\Rsub}
```

`\Rcmp` will compare 1.3 to −2.3. Since the first is greater than the second, `\IFgt` will be true and they will be added, producing −1.0. Following this the contents of the macro `\X` are pushed, it is examined by `\Rchk` and then either added to or subtracted from −1.0.

The user might never need to use `\Rchk` because every operator that puts something on the stack also runs `\Rchk`. In the above program, in fact, `\Rchk` is redundant since `\Rpush` will already have run it on the contents of `\X`.

There exist stack manipulation commands that allow the contents of the stack to be changed without performing any operations. These are really just conveniences, as there effects could be obtained with appropriate combinations of `\Rpop` and `\Rpush`. These commands, however, do not run `\Rchk` as `\Rpush` would.

Stack Manipulations

Command	**operation**
`\Rdup`	Puts another copy of the last element of the stack onto the stack.
`\Rexch`	Exchanges the last two elements on the stack.

After `\stopMFPprogram`, any changes to macros or to the stack are lost, unless arrangements have been made to save them. There are four commands provided. Two act on a macro which is the only argument, the other two have no arguments and act on the stack. The macro must simply contain a value, it cannot be more complicated and certainly cannot take an argument.

Exporting changed values

Command	**operation**
`\Export\macro`	Causes the definition of `\macro` to survive the program group.
`\Global\macro`	Causes the definition of `\macro` to be global.
`\ExportStack`	Causes the contents of the stack to survive the program group.
`\GlobalStack`	Causes the contents of the stack to be global.

The difference between `\Export` and `\Global` is solely in how *other* grouping is handled. If the program group is contained in another group (for example, inside an environment), then the result of `\Global\X` is that the definition of `\X` survives that group (and all containing groups) as well. On the other hand, after `\Export\X`, then the definition survives the program group, but not other containing groups.

If TEX grouping is used *inside* a program group, then using `\Export` inside that group has no effect at all, while `\Global` works as before.

The stack versions are implemented by running `\Export` or `\Global` on the internal macro that defines the stack, so they have the same behavior.

2.4 Errors

If one tries to *pop* from an empty stack, an error message will be issued. Ignoring the error causes the macro to have the value stored in the macro `\EndofStack`. Its default is `0.00000000`.

If one tries to divide by zero, an error message will be issued. Ignoring the error causes the result to be one of the following:

- Dividing 0 by 0 gives a result whose integer part is stored in `\ZeroOverZeroInt` and whose fractional part is stored in `\ZeroOverZeroFrac`. The default is `0.00000000`
- Dividing a nonzero x by 0 gives a result whose integer part is stored in `\xOverZeroInt` and whose fractional part is stored in `\xOverZeroFrac`. The defaults are both equal to `99999999`. The sign of the result will be the sign of x.

You can change any of these macros, but make sure they produce a result which is a number in standard form (as described earlier). These macros are copied directly into the result without checking. Then further processing steps may require the result to be a number in standard form.

Error messages may result from trying to process numbers given in incorrect format. However, there are so many ways for numbers to be incorrect that this package does not even try to detect them. Thus, they will only be caught if some TEX operation encounters something it cannot handle. (The LATEX manual calls these "weird errors" because the messages tend to be uninformative.) Incorrect numbers may even pass unnoticed, but leave random printed characters on the paper, or odd spacing.

3 Implementation

3.1 Utility macros

We announce ourself, and our purpose. We save the catcode of `@` and change it to letter. Several other catcodes are saved and set to other in this file. We also make provisions to load the extra definitions, either directly with `\MFPloadextra` or through a declared option in LATEX.

```
1 ⟨*sty⟩
2 \expandafter
3 \ifx \csname MFP@finish\endcsname\relax
4 \else \expandafter\endinput \fi
5 \expandafter\edef\csname MFP@finish\endcsname{%
6   \catcode64=\the\catcode64 \space
7   \catcode46=\the\catcode46 \space
8   \catcode60=\the\catcode60 \space
9   \catcode62=\the\catcode62 \space}%
10 \ifx\ProvidesPackage\UndEfInEd
11   \newlinechar`\^^J%
12   \message{%
```

```
      Package minifp: \MFPfiledate\space v\MFPfileversion. %
      Macros for real number operations %
      ^^Jand a stack-based programing language.^^J}%
\else
  \ProvidesPackage{minifp}[\MFPfiledate\space v\MFPfileversion. %
      Macros for real number operations %
      and a stack-based programing language.]%
  \DeclareOption{extra}{\def\MFPextra{}}%
  \ProcessOptions\relax
\fi
\catcode64=11
\ifx\MFPextra\UndEfInEd
  \def\MFP@loadextra{}%
\else
  \def\MFP@loadextra{\input mfpextra\relax}%
\fi
\def\MFPloadextra{\input mfpextra\relax}%
\catcode46=12
\catcode60=12
\catcode62=12
```

We check for LaTeX (ignoring LaTeX209); `\MFP@ifnoLaTeX`...`\MFP@end` is skipped in LaTeX and executed otherwise.

```
\long\def\gobbleto@MFP@end#1\MFP@end{}%
\def\MFP@end{\@empty}%
\ifx\documentclass\UndEfInEd
  \def\MFP@ifnoLaTeX{}%
\else
  \let\MFP@ifnoLaTeX\gobbleto@MFP@end
\fi
```

We have LaTeX's `\zap@space`. It pretty much *must* be used inside `\edef` or other purely expansion context. The rest of these are standard LaTeX internals. Note that the token list that `\zap@space` is applied to should probably never contain braces or expandable tokens.

Usage: `\edef\X{\zap@space<tokens>␣\@empty}`
The space is necessary in case none exist; the `\@empty` terminates the loop.

```
\let\@xp\expandafter
\def\@XP{\@xp\@xp\@xp}%
\MFP@ifnoLaTeX
  \def\@empty{}%
  \long\def\@gobble#1{}%
  \def\zap@space#1 #2{#1\ifx#2\@empty\else\@xp\zap@space\fi#2}%
  \long\def\@ifnextchar#1#2#3{%
    \let\reserved@d#1%
    \def\reserved@a{#2}%
    \def\reserved@b{#3}%
    \futurelet\@let@token\@ifnch}%
  \def\@ifnch{%
    \ifx\@let@token\@sptoken
```

```
      \let\reserved@c\@xifnch
    \else
      \ifx\@let@token\reserved@d
        \let\reserved@c\reserved@a
      \else
        \let\reserved@c\reserved@b
      \fi
    \fi
    \reserved@c}%
  {%
    \def\:{\global\let\@sptoken= }\: %
    \def\:{\@xifnch}\@xp\gdef\: {\futurelet\@let@token\@ifnch}%
  }%
  \def\@ifstar#1{\@ifnextchar*{\@firstoftwo{#1}}}%
  \long\def\@firstofone #1{#1}%
  \long\def\@firstoftwo #1#2{#1}%
  \long\def\@secondoftwo#1#2{#2}%
\MFP@end
```

We need to divide by both 10^4 and 10^8 several times. I could have allocated two count registers, but have taken the approach of only using those for intermediate calculations.

```
\def\MFP@tttfour {10000}% ttt = Ten To The
\def\MFP@ttteight{100000000}%
```

These are for manipulating digits. The `\...ofmany` commands require a sequence of arguments (brace groups or tokens) followed by `\MFP@end`. The minimum number of required parameters is surely obvious. For example, `\MFP@ninthofmany` must be used like

`\MFP@ninthofmany`⟨*9 or more arguments*⟩`\MFP@end`

All these are fully expandable.

```
\def\MFP@oneofmany#1#2\MFP@end{#1}%
\def\MFP@fifthofmany#1#2#3#4#5#6\MFP@end{#5}%
\def\MFP@ninthofmany#1#2#3#4#5#6#7#8{\MFP@oneofmany}%
\def\MFP@eightofmany#1#2#3#4#5#6#7#8#9\MFP@end{#1#2#3#4#5#6#7#8}%
```

3.2 Processing numbers and the stack

Our stack stores elements in groups, like

`{-1.234567890}{0.00001234}\MFP@eos`

with an end marker. The purpose of the marker is to prevent certain parameter manipulations from stripping off braces. This means we can't use `\@empty` to test for an empty stack. At the moment, only `\Rpop` actually checks, but all other stack commands (so far) use `\Rpop` to get their arguments.

```
\let\MFP@eos\relax
\def\MFP@EOS{\MFP@eos}%
\def\MFP@initRstack{\def\MFP@Rstack{\MFP@eos}}%
\MFP@initRstack
```

Define some scratch registers for arithmetic operations. We don't care that these might be already in use, as we only use them inside a group. However, we need one counter that will not be messed with by any of our operations. I must be sure not to use commands that change `\MFP@loopctr` in code that depends on it.

```
\countdef \MFP@tempa 0
\countdef \MFP@tempb 2
\countdef \MFP@tempc 4
\countdef \MFP@tempd 6
\countdef \MFP@tempe 8
\countdef \MFP@tempf 10
\newcount \MFP@loopctr
```

The following can only be used where unrestricted expansion is robust. It will allow results obtained inside a group to survive the group, but not be unrestrictedly global. Example: the code

`\MFP@endgroup@after{\def\noexpand\MFP@z@Val{\MFP@z@Val}}`

becomes

`\edef\x{\endgroup\def\noexpand\MFP@z@Val{\MFP@z@Val}}\x`

which gives, upon expansion of `\x`,

`\endgroup\def\MFP@z@Val{`⟨*expansion-of-*`\MFP@z@Val`⟩`}`

which defines `\MFP@z@Val` outside the current group to equal its expansion within the current group (provided the group was started with `\begingroup`).

We define a `\MFP@returned@values` to make all the conceivable produced values survive the group. The `\MFPcurr@Sgn` part is to permit testing the sign of the result and allow conditional code based on it.

I have been lax at making sure `\MFP@z@Ovr` is properly initiallized and properly checked whenever it could be relevant, and properly passed on. I think every internal command `\MFP@R`*xxx* should ensure it starts being zero and ends with a numerical value. At one time division could leave it undefined.

`\MFP@subroutine` executes its argument (typically a single command) with a wrapper that initializes all the macros that might need initializing, and returns the necessary results.

```
\def\MFP@endgroup@after#1{\edef\x{\endgroup#1}\x}%
\def\MFP@afterdef{\def\noexpand}%
\def\MFP@returned@values{%
  \MFP@afterdef\MFP@z@Val{\MFP@z@Sign\MFP@z@Int.\MFP@z@Frc}%
  \MFP@afterdef\MFP@z@Ovr{\MFP@z@Ovr}%
  \MFP@afterdef\MFP@z@Und{\MFP@z@Und}%
  \MFP@afterdef\MFPcurr@Sgn{\MFP@z@Sgn}}%
\def\MFP@subroutine#1{%
  \begingroup
    \MFP@Rzero
    \def\MFP@z@Ovr{0}%
    \def\MFP@z@Und{0}%
    #1%
  \MFP@endgroup@after\MFP@returned@values}%
\def\MFP@Rzero{%
```

```
  \def\MFP@z@Sgn{0}%
  \def\MFP@z@Int{0}%
  \def\MFP@z@Frc{00000000}}%
```

\EndofStack

We define here the error messages: popping from an empty stack and dividing by zero. In addition to the error messages, we provide some default values that hopefully allow some operations to continue.

We also have a warning or two.

```
\def\MFP@errmsg#1#2{%
\begingroup
  \newlinechar`\^^J\let~\space
  \def\MFP@msgbreak{^^J~~~~~~~~~~~~~~~}%
  \edef\reserved@a{\errhelp{#2}}\reserved@a
  \errmessage{MiniFP error: #1}%
\endgroup}%
\def\MFP@popempty@err{%
    \MFP@errmsg{cannot POP from an empty stack}%
      {There were no items on the stack for the POP operation. %
       If you continue, ^^Jthe macro will contain the %
       value \EndofStack.}}%
\def\EndofStack{0.00000000}%
\def\MFP@dividebyzero@err{%
  \MFP@errmsg{division by zero}%
    {You tried to divide by zero. What were you thinking? %
     If you continue, ^^Jthe value assigned will be either %
     \ZeroOverZeroInt.\ZeroOverZeroFrac~(numerator=0) or %
     ^^J+/-\xOverZeroInt.\xOverZeroFrac~(numerator<>0).}}%
\def\MFP@warn#1{%
\begingroup
  \newlinechar`\^^J\let~\space
  \def\MFP@msgbreak{^^J~~~~~~~~~~~~~~~~}%
  \immediate\write16{^^JMiniFP warning: #1.^^J}%
\endgroup}%
```

\MaxRealInt
\MaxRealFrac

These are the largest possible integer and fractional parts of a real number. They are returned for division by zero, for logarithm of zero, and when overflow is detected in the exponential function.

```
\def\MaxRealInt      {99999999}%
\def\MaxRealFrac     {99999999}%
```

\xOverZeroInt
\xOverZeroFrac
\ZeroOverZeroInt
\ZeroOverZeroFrac

These are the results returned when trying to divide by zero. Two are used when dividing a nonzero number by zero and and two when trying to divide zero by zero.

```
\def\xOverZeroInt    {\MaxRealInt}%
\def\xOverZeroFrac   {\MaxRealFrac}%
\def\ZeroOverZeroInt {0}%
\def\ZeroOverZeroFrac{00000000}%
```

These macros strip the spaces, process a number into sign, integer and fractional parts, and pad the fractional part out to eight decimals. They are used in

push so that the stack will only contains reals in a normalized form. Some of them are also used to preprocess the reals in the operand versions of commands

The `\MFP@*@Int` and `\MFP@*@Frc` parts are always positive, the sign being stored in `\MFP@*@Sgn` as -1, 0 or 1.

We strip the spaces and pad the fractional parts separately because they are unnecessary when processing *pop*ed reals (though they wouldn't hurt).

The number to be parsed is `#4` and the macros to contain the parts are the first three arguments. Since we normally call `\MFPparse@real` with one of two sets of macros, we have two shortcuts for those cases.

```
\def\MFPparse@real#1#2#3#4{%
  \MFPnospace@def\MFPtemp@Val{#4}%
  \MFPprocess@into@parts\MFPtemp@Val#1#2#3%
  \MFP@padtoeight#3}%
\def\MFPparse@x{\MFPparse@real\MFP@x@Sgn\MFP@x@Int\MFP@x@Frc}%
\def\MFPparse@y{\MFPparse@real\MFP@y@Sgn\MFP@y@Int\MFP@y@Frc}%
```

This macro strips all spaces out of the number (not just before and after). It takes a macro that will hold the result, followed by the number (as a macro or a group of actual digits).

```
\def\MFPnospace@def#1#2{%
  \edef#1{#2\space}\edef#1{\@xp\zap@space#1\@empty}}%
```

This is the process that splits a number into parts. The biggest difficulty is obtaining the sign. All four arguments are macros, with the first one holding the number. Following that are the macros to hold the sign, integer and fractional parts.

```
\def\MFPprocess@into@parts#1#2#3#4{%
  \@xp\MFPsplit@dot#1..\MFP@end #3#4%
```

At this point `#3` holds the part before the dot (or the whole thing if there was no dot) and `#4` holds the part after the dot, (or nothing). Now is the first place where having at most eight digits simplifies things. Note that `#3` could contain any number of consecutive signs followed by up to eight digits. It could be zero or empty, so to avoid losing the sign we append a `1` (for up to nine digits). We temporarily define the sign based on the result, but may need to drop it if both the integer and fractional parts are zero.

Prepending a zero to the fractional part pemits it to be empty. In the final `\edef`, `#3` is made positive.

```
  \ifnum#31<0 \def#2{-1}%
  \else \def#2{1}%
  \fi
  \ifnum #30=0
    \def#3{0}%
    \ifnum 0#4=0 \def#2{0}\fi
  \fi
  \edef#3{\number \ifnum #2<0 -\fi#3}}%
```

This only copies the parts before and after the dot, `#1` and `#2`, into macros `#4` and `#5`.

```
\def\MFPsplit@dot#1.#2.#3\MFP@end#4#5{\edef#4{#1}\edef#5{#2}}%
```

This is used to pad the fractional part to eight places with zeros. If a number with more than eight digits survives to this point, it gets truncated.

```
\def\MFP@padtoeight#1{%
  \edef#1{\@xp\MFP@eightofmany#100000000\MFP@end}}%
```

These take operands off the stack. We know already that there are no spaces and that the fractional part has eight digits.

```
\def\MFPgetoperand@x{\Rpop\MFP@x@Val
  \MFPprocess@into@parts\MFP@x@Val\MFP@x@Sgn\MFP@x@Int\MFP@x@Frc}%
\def\MFPgetoperand@y{\Rpop\MFP@y@Val
  \MFPprocess@into@parts\MFP@y@Val\MFP@y@Sgn\MFP@y@Int\MFP@y@Frc}%
```

Concatenate an argument (or two) to the front of stack. The material must already be in correct format.

```
\def\MFP@Rcat#1{\edef\MFP@Rstack{{#1}\MFP@Rstack}}%
\def\MFP@Rcattwo#1#2{\edef\MFP@Rstack{{#1}{#2}\MFP@Rstack}}%
```

Convert from a signum (a number) to a sign (− or nothing):

```
\def\MFP@Sign#1{\ifnum#1<0 -\fi}%
\def\MFP@x@Sign{\MFP@Sign\MFP@x@Sgn}%
\def\MFP@y@Sign{\MFP@Sign\MFP@y@Sgn}%
\def\MFP@z@Sign{\MFP@Sign\MFP@z@Sgn}%
```

Sometimes only parts of the number needs changing (used in CHS, ABS). This copies the integer and fractional parts of x into z.

```
\def\copyMFP@x{\edef\MFP@z@Int{\MFP@x@Int}\edef\MFP@z@Frc{\MFP@x@Frc}}%
```

We use `\MFPpush@result` to put the result of internal operations onto the stack. For convenience, we also have it set the sign flags.

```
\def\MFPpush@result{\MFP@Rchk\MFPcurr@Sgn\MFP@Rcat\MFP@z@Val}%
```

When *pop* encounters an empty stack it gobbles the code that would perform the *pop* (`#1`) and defines the macro (`#2`) to contain `\EndofStack`. The default meaning for this macro is 0.

```
\def\if@EndofStack{%
  \ifx\MFP@EOS\MFP@Rstack
    \@xp\@firstoftwo
  \else
    \@xp\@secondoftwo
  \fi}%
```

The macro `\Rpop` calls `\MFP@popit` followed by the contents of the stack, the token `\MFP@end` and the macro to *pop* into. If the stack is not empty, `\doMFP@popit` will read the first group `#1` into that macro `#3`, and then redefine the stack to be the rest of the argument `#2`. If the stack is empty, `\doMFP@EOS` will equate the macro to `\EndofStack` (initialized to `0.00000000`) after issuing an error message.

```
\def\MFP@popit{\if@EndofStack\doMFP@EOS\doMFP@popit}%
\def\doMFP@EOS#1\MFP@end#2{\MFP@popempty@err\let#2\EndofStack}%
\def\doMFP@popit#1#2\MFP@end#3{\edef\MFP@Rstack{#2}\edef#3{#1}}%
```

3.3 The user-level operations

All operations that can be done on arguments as well as the stack will have a common format: The stack version pops the requisite numbers and splits them into internal macros (`\MFPgetoperand@*`), runs an internal command that operates on these internal macros, then "pushes" the result returned. The internal commands take care to return the result in proper form so we don't actually run `\Rpush`, but only `\MFPpush@result`.

The operand version processes the operands into normalized form (as if pushed, using `\MFPparse@*`), then proceeds as in the stack version, but copies the result into the named macro instead of to the stack (`\MFPstore@result`).

For unary operations we process one argument or stack element. We call it x and use the `x` version of all macros. All internal commands (`#1`) return the results in `z` versions.

`\MFPchk` The `\MFPchk` command examines its argument and sets a flag according to its sign.

```
\def\MFPchk#1{%
  \MFPparse@x{#1}%
  \MFP@Rchk\MFP@x@Sgn}%
```

We make `\MFP@Rchk` a little more general than is strictly needed here, by giving it an argument (instead of only examining `\MFP@x@Sgn`). This is so we can apply it to the results of operations (which would be in `\MFPcurr@Sgn`).

```
\def\MFP@Rchk#1{%
  \MFPclear@flags
  \ifnum#1>0  \MFP@postrue
  \else\ifnum#1<0  \MFP@negtrue
  \else  \MFP@zerotrue
  \fi\fi}%
\def\MFPclear@flags{\MFP@zerofalse \MFP@negfalse \MFP@posfalse}%
```

`\IFzero` `\IFneg` `\IFpos` These are the user interface to the internal `\ifMFP@zero`, `\ifMFP@neg`, `\ifMFP@pos`

```
\def\IFzero{\ifMFP@zero\@xp\@firstoftwo\else\@xp\@secondoftwo\fi}%
\def\IFneg {\ifMFP@neg \@xp\@firstoftwo\else\@xp\@secondoftwo\fi}%
\def\IFpos {\ifMFP@pos \@xp\@firstoftwo\else\@xp\@secondoftwo\fi}%
\newif\ifMFP@zero \newif\ifMFP@neg \newif\ifMFP@pos
```

`\MFPcmp` `\IFeq` `\IFlt` `\IFgt` Our comparison commands parallel the check-sign commands. They even reuse the same internal booleans. The differences: the internal `\MFP@Rcmp` doesn't take arguments and the comparison test is a little more involved. We could simply subtract, which automatically sets the internal booleans, but it is way more efficient to compare sizes directly.

```
\newif\ifMFPdebug
\def\MFPcmp#1#2{\MFPparse@x{#1}\MFPparse@y{#2}\MFP@Rcmp}%
\def\MFP@Rcmp{\MFPclear@flags
  \ifnum \MFP@x@Sign\MFP@x@Int>\MFP@y@Sign\MFP@y@Int\relax
    \MFP@postrue
  \else\ifnum \MFP@x@Sign\MFP@x@Int<\MFP@y@Sign\MFP@y@Int\relax
```

```
    \MFP@negtrue
  \else\ifnum \MFP@x@Sign\MFP@x@Frc>\MFP@y@Sign\MFP@y@Frc\relax
    \MFP@postrue
  \else\ifnum \MFP@x@Sign\MFP@x@Frc<\MFP@y@Sign\MFP@y@Frc\relax
    \MFP@negtrue
  \else
    \MFP@zerotrue
  \fi\fi\fi\fi}%
\let\IFeq\IFzero\let\IFlt\IFneg \let\IFgt\IFpos
```

Given an operation (*pop*, *chs*, or whatever), the stack version will have the same name with "`R`" (for "real") prepended. The operand versions will have the same name with "`MFP`" prepended. The internal version has the same name as the stack version, with an "`MFP@`" prepended.

The unary operations are:

chs change sign of x.
abs absolute value of x.
dbl double x.
halve halve x.
sgn $+1$, -1 or 0 depending on the sign of x.
sq square x.
int zero out the fractional part of x.
frac zero out the integer part of x.
floor largest integer not exceeding x.
ceil smallest integer not less than x.

The binary operations are (x represents the first and y second):

add add x and y.
sub subtract y from x.
mul multiply x and y.
div divide x by y.

There are also some operations that do not actually change any values, but may change the stack or the state of some boolean:

cmp compare x and y (stack version does not change stack).
chk examine the sign of x (stack version does not change stack).
dup stack only, duplicate the top element of the stack.
push stack only, put a value onto the top of the stack.
pop stack only, remove the top element of the stack, store it in a variable.
exch stack only, exchange top two elements of the stack.

`\startMFPprogram` `\stopMFPprogram`

The purpose of `\startMFPprogram` is to start the group, inside of which all the stack operations can be used. The ensuing `\stopMFPprogram` closes the group.

```
\def\startMFPprogram{%
\begingroup
```

\Rchs
\Rabs
\Rdbl
\Rhalve
\Rsgn
\Rsq
\Rinv
\Rint
\Rfrac
\Rfloor
\Rceil
\Rincr
\Rdecr
\Rzero

Then we give definitions to all the stack-based macros. These definitions are all lost after the group ends.

We start with the unary operations. Note that all they do is call a wrapper macro `\MFP@stack@Unary` with an argument which is the internal version of the command.

```
\def\Rchs  {\MFP@stack@Unary\MFP@Rchs}%
\def\Rabs  {\MFP@stack@Unary\MFP@Rabs}%
\def\Rdbl  {\MFP@stack@Unary\MFP@Rdbl}%
\def\Rhalve{\MFP@stack@Unary\MFP@Rhalve}%
\def\Rsgn  {\MFP@stack@Unary\MFP@Rsgn}%
\def\Rsq   {\MFP@stack@Unary\MFP@Rsq}%
\def\Rinv  {\MFP@stack@Unary\MFP@Rinv}%
\def\Rint  {\MFP@stack@Unary\MFP@Rint}%
\def\Rfrac {\MFP@stack@Unary\MFP@Rfrac}%
\def\Rfloor{\MFP@stack@Unary\MFP@Rfloor}%
\def\Rceil {\MFP@stack@Unary\MFP@Rceil}%
\def\Rincr {\MFP@stack@Unary\MFP@Rincr}%
\def\Rdecr {\MFP@stack@Unary\MFP@Rdecr}%
\def\Rzero {\MFP@stack@Unary\MFP@Rzero}%
```

\Radd
\Rsub
\Rmul
\Rmpy
\Rdiv
\Rmin
\Rmax

Then the binary operations, which again call a wrapper around the internal version.

```
\def\Radd  {\MFP@stack@Binary\MFP@Radd}%
\def\Rmul  {\MFP@stack@Binary\MFP@Rmul}%
\let\Rmpy\Rmul
\def\Rsub  {\MFP@stack@Binary\MFP@Rsub}%
\def\Rdiv  {\MFP@stack@Binary\MFP@Rdiv}%
\def\Rmin  {\MFP@stack@Binary\MFP@Rmin}%
\def\Rmax  {\MFP@stack@Binary\MFP@Rmax}%
```

\Rnoop
\Rcmp
\Rchk
\Rpush
\Rpop
\Rexch
\Rdup

And finally some special commands. There is a no-op and commands for comparing, checking, and manipulation of the stack.

```
\let\Rnoop\relax
\def\Rcmp{%
  \MFPgetoperand@y\MFPgetoperand@x
  \MFP@Rcat\MFP@x@Val\MFP@Rcat\MFP@y@Val
  \MFP@Rcmp}%
\def\Rchk{%
  \MFPgetoperand@x
  \MFP@Rcat\MFP@x@Val
  \MFP@Rchk\MFP@x@Sgn}%
\def\Rpush##1{%
  \MFPparse@x{##1}%
  \edef\MFP@z@Val{\MFP@x@Sign\MFP@x@Int.\MFP@x@Frc}%
  \edef\MFPcurr@Sgn{\MFP@x@Sgn}%
  \MFPpush@result}%
\def\Rpop{\@xp\MFP@popit\MFP@Rstack\MFP@end}%
\def\Rexch{%
  \Rpop\MFP@x@Val\Rpop\MFP@y@Val
  \MFP@Rcattwo\MFP@y@Val\MFP@x@Val}%
```

```
  \def\Rdup{%
    \Rpop\MFP@x@Val
    \MFP@Rcattwo\MFP@x@Val\MFP@x@Val}%
```

If `mfpextra.tex` is input, then `\MFP@Rextra` makes the additional commands in that file available to an MINIFP program.

`\Global` `\GlobalStack` `\Export` `\ExportStack`

The last four commands allow computed values to be made available outside the program group

```
  \MFP@Rextra
  \let\Global\MFP@Global
  \let\GlobalStack\MFP@GlobalStack
  \let\Export\MFP@Export
  \let\ExportStack\MFP@ExportStack}%
\def\stopMFPprogram{\@xp\endgroup\MFPprogram@returns}%
\let\MFP@Rextra\@empty
\let\MFPprogram@returns\@empty
```

`\MFPchs` `\MFPabs` `\MFPdbl` `\MFPhalve` `\MFPsgn` `\MFPsq` `\MFPinv` `\MFPint` `\MFPfrac` `\MFPfloor` `\MFPceil` `\MFPincr` `\MFPdecr` `\MFPzero` `\MFPstore`

Now we define the operand versions. These also are defined via a wrapper command that executes the very same internal commands as the stack versions.

First the unary operations.

```
\def\MFPchs  {\MFP@op@Unary\MFP@Rchs}%
\def\MFPabs  {\MFP@op@Unary\MFP@Rabs}%
\def\MFPdbl  {\MFP@op@Unary\MFP@Rdbl}%
\def\MFPhalve{\MFP@op@Unary\MFP@Rhalve}%
\def\MFPsgn  {\MFP@op@Unary\MFP@Rsgn}%
\def\MFPsq   {\MFP@op@Unary\MFP@Rsq}%
\def\MFPinv  {\MFP@op@Unary\MFP@Rinv}%
\def\MFPint  {\MFP@op@Unary\MFP@Rint}%
\def\MFPfrac {\MFP@op@Unary\MFP@Rfrac}%
\def\MFPfloor{\MFP@op@Unary\MFP@Rfloor}%
\def\MFPceil {\MFP@op@Unary\MFP@Rceil}%
\def\MFPincr {\MFP@op@Unary\MFP@Rincr}%
\def\MFPdecr {\MFP@op@Unary\MFP@Rdecr}%
\def\MFPzero {\MFP@op@Unary\MFP@Rzero}%
\def\MFPstore{\MFP@op@Unary\MFP@Rstore}%
```

`\MFPadd` `\MFPsub` `\MFPmul` `\MFPmpy` `\MFPdiv` `\MFPmin` `\MFPmax`

Then the binary operations.

```
\def\MFPadd{\MFP@op@Binary\MFP@Radd}%
\def\MFPmul{\MFP@op@Binary\MFP@Rmul}%
\let\MFPmpy\MFPmul
\def\MFPsub{\MFP@op@Binary\MFP@Rsub}%
\def\MFPdiv{\MFP@op@Binary\MFP@Rdiv}%
\def\MFPmin{\MFP@op@Binary\MFP@Rmin}%
\def\MFPmax{\MFP@op@Binary\MFP@Rmax}%
```

A *nullary* operation is one that produces a result with no operand. Thus, it could return a fixed constant, or it could perform calculations that obtain input from the system (e.g., current time). At the moment we don't define any.

```
\def\MFP@stack@Nullary#1{%
  \MFP@subroutine{#1}\MFPpush@result}%
\def\MFP@op@Nullary#1{%
```

```
  \MFP@subroutine{#1}\MFPstore@result}%
```

These are the wrappers for unary operations. The operand versions have a second argument, the macro that stores the result. But this will be the argument of `\MFPstore@result`.

```
\def\MFP@stack@Unary#1{%
  \MFPgetoperand@x
  \MFP@subroutine{#1}\MFPpush@result}%
\def\MFP@op@Unary#1#2{%
  \MFPparse@x{#2}%
  \MFP@subroutine{#1}\MFPstore@result}%
\def\MFPstore@result#1{\MFP@Rchk\MFPcurr@Sgn\edef#1{\MFP@z@Val}}%
```

These are the wrappers for binary operations. The top level definitions are almost identical to those of the unary operations. The only difference is they *pop* or parse two operands.

```
\def\MFP@stack@Binary#1{%
  \MFPgetoperand@y \MFPgetoperand@x
  \MFP@subroutine{#1}\MFPpush@result}%
\def\MFP@op@Binary#1#2#3{%
  \MFPparse@x{#2}\MFPparse@y{#3}%
  \MFP@subroutine{#1}\MFPstore@result}%
```

`\MFPnoop` We end with a traditional, but generally useless command, the no-op, which does nothing. It doesn't even have a wrapper.

```
\let\MFPnoop\relax
```

3.4 The internal computations

To change the sign or get the absolute value, we just need to set the value of `\MFP@x@Sgn`.

```
\def\MFP@Rabs{%
  \copyMFP@x \edef\MFP@z@Sgn{\ifnum\MFP@x@Sgn=0 0\else1\fi}}%
\def\MFP@Rchs{\copyMFP@x \edef\MFP@z@Sgn{\number-\MFP@x@Sgn}}%
```

The doubling and halving operations are more efficient ways to multiply or divide a number by 2. For doubling, copy x to y and add. For halving, we use basic TEX integer division, more efficient than multiplying by 0.5 and far more than using `\MFP@Rdiv`.

In `\MFP@Rhalve`. we add 1 to the fractional part for rounding purposes, and we move any odd 1 from the end of the integer part to the start of the fractional part.

```
\def\MFP@Rdbl{\MFP@Rcopy xy\MFP@Radd}%
\def\MFP@Rhalve{%
  \MFP@tempa\MFP@x@Int
  \MFP@tempb\MFP@x@Frc\relax
  \ifodd\MFP@tempb
    \def\MFP@z@Und{5}%
    \advance\MFP@tempb 1
    \ifnum\MFP@ttteight=\MFP@tempb
```

```
      \MFP@tempb0 \advance\MFP@tempa1
    \fi
  \fi
  \ifodd \MFP@tempa
    \advance\MFP@tempb \MFP@ttteight\relax
  \fi
  \divide\MFP@tempa 2
  \divide\MFP@tempb 2
  \MFP@Rloadz\MFP@x@Sgn\MFP@tempa\MFP@tempb}%
```

The signum is 0.0, 1.0 or -1.0 to match the sign of x.

```
\def\MFP@Rsgn{\MFP@Rloadz \MFP@x@Sgn{\ifnum\MFP@x@Sgn=0 0\else1\fi}0}%
```

The squaring operation just calls `\MFP@Rmul` after copying x to y. Its gain in efficiency over a multiplication is that it can skip preprocessing of the second (identical) operand.

```
\def\MFP@Rsq{\MFP@Rcopy xy\MFP@Rmul}%
```

The inversion operation just calls `\MFP@Rdiv` after copying x to y and 1 to x. Its advantage over a divide is it skips the preprocessing of 1 as an operand.

```
\def\MFP@Rinv{\MFP@Rcopy xy\MFP@Rload x110\MFP@Rdiv}%
```

Integer part: replace fractional part with zeros.

```
\def\MFP@Rint{%
  \MFP@Rloadz {\ifnum\MFP@x@Int=0 0\else\MFP@x@Sgn\fi}\MFP@x@Int 0}%
```

Fractional part: replace integer part with a zero.

```
\def\MFP@Rfrac{%
  \MFP@Rloadz {\ifnum\MFP@x@Frc=0 0\else\MFP@x@Sgn\fi}0\MFP@x@Frc}%
```

To increment and decrement by 1, except in border cases, we need only address the integer part of a number. This doesn't seem so simple written out but, even so, it is more efficient than full-blown addition. It would be very slightly more efficient if `\MFP@Rdecr` did not call `\MFP@Rincr`, but instead was similarly coded.

```
\def\MFP@Rincr{%
 \ifnum\MFP@x@Sgn<0
   \ifcase\MFP@x@Int
     \MFP@tempa\MFP@ttteight
     \advance\MFP@tempa -\MFP@x@Frc\relax
     \MFP@Rloadz 10\MFP@tempa
   \or
      \MFP@Rloadz{\ifnum\MFP@x@Frc=0 0\else -1\fi}0\MFP@x@Frc
   \else
     \MFP@tempa\MFP@x@Int
     \advance\MFP@tempa -1
     \MFP@Rloadz{-1}\MFP@tempa\MFP@x@Frc
   \fi
 \else
   \MFP@tempa\MFP@x@Int
   \advance\MFP@tempa 1
   \MFP@Rloadz 1\MFP@tempa\MFP@x@Frc
 \fi}%
```

```
\def\MFP@Rdecr{%
  \edef\MFP@x@Sgn{\number -\MFP@x@Sgn}\MFP@Rincr
  \edef\MFP@z@Sgn{\number -\MFP@z@Sgn}}%
\def\MFP@Rstore{\MFP@Rcopy xz}%
```

The floor of a real number x is the largest integer not larger than x. The ceiling is the smallest integer not less than x. For positive x, floor is the same as integer part. Not true for negative x. Example: $\text{int}(-1.5) = -1$ but floor $= -2$

We use the same code to get floor or ceiling, the appropriate inequality character being its argument.

```
\def\MFP@Rfloorceil#1{%
  \MFP@tempa\MFP@x@Int\relax
  \ifnum \MFP@x@Sgn #10
    \ifnum\MFP@x@Frc=0
    \else
      \advance\MFP@tempa1
    \fi
  \fi
  \MFP@Rloadz{\ifnum\MFP@x@Int=0 0\else\MFP@x@Sgn\fi}\MFP@tempa0}%
\def\MFP@Rfloor{\MFP@Rfloorceil<}%
\def\MFP@Rceil {\MFP@Rfloorceil>}%
```

For multiplication, after the usual break into integer and fractional parts, we further split these parts into 4-digit pieces with `\MFP@split`. The first argument (`#1`) holds the eight digit number, then `#2` is a macro that will hold the top four digits and `#3` will hold the bottom four.

```
\def\MFP@split#1#2#3{%
  \begingroup
    \MFP@tempa#1\relax
    \MFP@tempb\MFP@tempa
    \divide\MFP@tempb by\MFP@tttfour
    \edef#2{\number\MFP@tempb}%
    \multiply\MFP@tempb by\MFP@tttfour
    \advance\MFP@tempa-\MFP@tempb
  \MFP@endgroup@after{%
    \MFP@afterdef#2{#2}%
    \MFP@afterdef#3{\number\MFP@tempa}%
  }}%
%
\def\MFP@@split{%
  \MFP@split\MFP@x@Int\MFP@x@Int@ii\MFP@x@Int@i
  \MFP@split\MFP@x@Frc\MFP@x@Frc@i\MFP@x@Frc@ii
  \MFP@split\MFP@y@Int\MFP@y@Int@ii\MFP@y@Int@i
  \MFP@split\MFP@y@Frc\MFP@y@Frc@i\MFP@y@Frc@ii}%
```

We will store the intermediate and final products in `\MFP@z@*`. Each one is ultimately reduced to four digits, like the parts of x and y. As each base-10000 digit of y is multiplied by a digit of x, we add the result to the appropriate digit of the partial result z.

The underflow ends up in \MFP@z@Frc@iv and \MFP@z@Frc@iii. Overflow will be in \MFP@z@Int@iii. Unlike the rest, it can be up to eight digits because we do not need to carry results out of it.

This command prepends zeros so a number fills four slots. Here #1 is a macro holding the value and it is redefined to contain the result. A macro that calls this should ensure that #1 is not empty and is less than 10,000.

```
\def\makeMFP@fourdigits#1{%
  \edef#1{\@xp\MFP@fifthofmany\number#1{}{0}{00}{000}\MFP@end\number#1}}%
```

This is the same, but produces eight digits. Similarly #1 should be nonempty and less than 100,000,000.

```
\def\makeMFP@eightdigits#1{%
  \edef#1{\@xp\MFP@ninthofmany\number#1%
    {}{0}{00}{000}{0000}{00000}{000000}{0000000}\MFP@end\number#1}}%
```

The following macros implement carrying. The macros \MFP@carrya and \MFP@carrym should be followed by two macros that hold numbers. The first number can have too many digits. These macros remove extra digits from the front and add their value to the number in the second macro (the "carry"). Both act by calling \MFP@carry, which is told the number of digits to keep via #1 (10,000 for four digits, 100,000,000 for eight). The "a" in \MFP@carrya is for addition and "m" is for multiplication, which indicates where these will mainly be used.

```
\def\MFP@carrya{\MFP@carry\MFP@ttteight}%
\def\MFP@carrym{\MFP@carry\MFP@tttfour}%
\def\MFP@carry#1#2#3{%
  \begingroup
    \MFP@carryi{#1}#2#3%
  \MFP@endgroup@after{%
    \MFP@afterdef#3{\number\MFP@tempa}%
    \MFP@afterdef#2{\number\MFP@tempb}%
  }}%
```

This is the "internal" carry. #1, #2, and #3 are as in \MFP@carry. Its advantage is that it can be used used where #2 and #3 are not macros, leaving the result in \MFP@tempa and \MFP@tempb with \MFP@tempb in the correct range, $[0, \#1)$. Its disadvantage is it does not protect temporary registers. Warning: do not use it with #2=\MFP@tempa and do not use it without grouping if you want to preserve the values in these temporary count registers.

```
\def\MFP@carryi#1#2#3{%
    \MFP@tempa=#3\relax
    \MFP@tempb=#2\relax
    \MFP@tempc=\MFP@tempb
    \divide  \MFP@tempc #1\relax
    \advance \MFP@tempa \MFP@tempc
    \multiply\MFP@tempc #1\relax
    \advance \MFP@tempb -\MFP@tempc}%
```

This adds #1 to #2, the result goes into macro #3. This does no checking. It is basicly used to add with macros instead of registers.

```
\def\MFP@addone#1#2#3{%
  \begingroup
    \MFP@tempa#1%
    \advance\MFP@tempa#2\relax
  \MFP@endgroup@after{%
    \MFP@afterdef#3{\number\MFP@tempa}%
  }}%
```

Multiply `#1` by `\MFP@tempb` and add to `#2`. `\MFP@tempb` is one digit (base=10000) of y in multiplying $x \times y$, `#1` (usually a macro) holds one digit of x. `#2` is a macro that will hold one digit of the final product z. The product is added to it (overflow is taken care of later by the carry routines).

```
\def\MFP@multiplyone#1#2{%
  \MFP@tempa#1%
  \multiply\MFP@tempa\MFP@tempb
  \advance\MFP@tempa#2%
  \edef#2{\number\MFP@tempa}}%
```

This does the above multiplication-addition for all four "digits" of x. This is where `\MFP@tempb` is initialized for `\MFP@multiplyone`. The first argument represents a digit of y, the remaining four arguments are macros representing the digits of z that are involved in multiplying the digits of x by `#1`.

```
\def\MFP@multiplyfour#1#2#3#4#5{%
  \MFP@tempb #1\relax
  \MFP@multiplyone\MFP@x@Int@ii #2%
  \MFP@multiplyone\MFP@x@Int@i  #3%
  \MFP@multiplyone\MFP@x@Frc@i  #4%
  \MFP@multiplyone\MFP@x@Frc@ii #5}%
```

Now we begin the internal implementations of the binary operations. All four expect macros `\MFP@x@Sgn`, `\MFP@x@Int`, `\MFP@x@Frc`, `\MFP@y@Sgn`, `\MFP@y@Int` and `\MFP@y@Frc` to be the normalized parts of two real numbers x and y.

`\MFP@Rsub` just changes the sign of y and then calls `\MFP@Radd`.

`\MFP@Radd` checks whether x and y have same or different signs. In the first case we need only add absolute values and the sign of the result will match that of the operands. In the second case, finding the sign of the result is a little more involve (and "borrowing" may be needed).

```
\def\MFP@Rsub{\edef\MFP@y@Sgn{\number-\MFP@y@Sgn}\MFP@Radd}%
\def\MFP@Radd{%
  \MFP@tempa\MFP@x@Sgn
  \multiply\MFP@tempa\MFP@y@Sgn\relax
  \ifcase\MFP@tempa
    \ifnum \MFP@x@Sgn=0
      \MFP@Rcopy yz%
    \else
      \MFP@Rcopy xz%
    \fi
  \or
    \@xp\MFP@Radd@same
  \else
```

```
    \@xp\MFP@Radd@diff
  \fi}%
```

`\MFP@Radd@same` adds two numbers which have the same sign. The sign of the result is the common sign. The fractional and integer parts are added separately, then a carry is invoked. The overflow (`\MFP@z@Ovr`) could be only a single digit 0 or 1.

```
\def\MFP@Radd@same{%
  \MFP@addone\MFP@x@Frc\MFP@y@Frc\MFP@z@Frc
  \MFP@addone\MFP@x@Int\MFP@y@Int\MFP@z@Int
  \MFP@carrya\MFP@z@Frc\MFP@z@Int
  \MFP@carrya\MFP@z@Int\MFP@z@Ovr
  \makeMFP@eightdigits\MFP@z@Frc
  \edef\MFP@z@Sgn{\MFP@x@Sgn}}%
```

We are now adding two numbers with opposite sign. Since $x \neq 0$ this is the same as $\mathrm{sgn}(x)(|x| - |y|)$. So we subtract absolute values, save the result in `\MFP@z@Sgn`, `\MFP@z@Int` and `\MFP@z@Frc` (with the last two nonnegative, as usual), then change the sign of `\MFP@z@Sgn` if `\MFP@x@Sgn` is negative. Since the difference between numbers in $[0, 10^8)$ has absolute value in that range, there is no carrying. However, there may be borrowing.

```
\def\MFP@Radd@diff{%
  \MFP@addone\MFP@x@Frc{-\MFP@y@Frc}\MFP@z@Frc
  \MFP@addone\MFP@x@Int{-\MFP@y@Int}\MFP@z@Int
```

Now we need to establish the sign and arrange the borrow. The sign of the result is the sign of `\MFP@z@Int` unless it is 0; in that case it, is the sign of `\MFP@z@Frc`. There must be a simpler coding, though.

```
  \MFP@tempa=\MFP@z@Int
  \MFP@tempb=\MFP@z@Frc\relax
  \ifnum\MFP@tempa=0 \else \MFP@tempa=\MFP@Sign\MFP@tempa 1 \fi
  \ifnum\MFP@tempb=0 \else \MFP@tempb=\MFP@Sign\MFP@tempb 1 \fi
  \ifnum\MFP@tempa=0 \MFP@tempa=\MFP@tempb \fi
```

Now we have the sign of $|x| - |y|$ in `\MFP@tempa`, and we multiply that sign by the sign of x to get `\MFP@z@Sgn`. Then we multiply the current value of z by that sign to get the absolute value, stored in `\MFP@tempa` and `\MFP@tempb`.

```
  \edef\MFP@z@Sgn{\number\MFP@x@Sign\MFP@tempa}%
  \MFP@tempb\MFP@tempa
  \multiply\MFP@tempa \MFP@z@Int
  \multiply\MFP@tempb \MFP@z@Frc\relax
```

What we should have now is a positive number which might still be represented with a negative fractional part. A human being performing the subtraction would have borrowed first. Being a computer, we do it last, and we're done.

```
  \ifnum\MFP@tempb<0
    \advance\MFP@tempb\MFP@ttteight
    \advance\MFP@tempa-1
  \fi
  \edef\MFP@z@Int{\number\MFP@tempa}%
```

```
  \edef\MFP@z@Frc{\number\MFP@tempb}%
  \makeMFP@eightdigits\MFP@z@Frc}%
```

`\MFP@Rmul` first computes the (theoretical) sign of the product: if it is zero, return zero, otherwise provisionally set the sign of the product and call `\MFP@@Rmul`.

```
\def\MFP@Rmul{%
  \ifnum\MFP@x@Sgn=0 \MFP@Rzero
  \else\ifnum\MFP@y@Sgn=0 \MFP@Rzero
  \else \edef\MFP@z@Sgn{\number\MFP@x@Sign\MFP@y@Sgn}%
    \@XP\MFP@@Rmul
  \fi\fi}%
```

`\MFP@@Rmul` first initializes the macros that will hold the base-10000 digits of z. Then it splits the four expected macros into eight macros that hold the base-10000 digits for each of x and y. Then each digit of y is used to multiply the four digits of x and the results are added to corresponding digits of z.

```
\def\MFP@@Rmul{%
  \def\MFP@z@Frc@iv {0}\def\MFP@z@Frc@iii{0}%
  \def\MFP@z@Frc@ii {0}\def\MFP@z@Frc@i  {0}%
  \def\MFP@z@Int@i  {0}\def\MFP@z@Int@ii {0}%
  \def\MFP@z@Int@iii{0}%
  \MFP@@split
  \MFP@multiplyfour \MFP@y@Frc@ii \MFP@z@Frc@i
    \MFP@z@Frc@ii \MFP@z@Frc@iii \MFP@z@Frc@iv
  \MFP@multiplyfour \MFP@y@Frc@i  \MFP@z@Int@i
    \MFP@z@Frc@i  \MFP@z@Frc@ii  \MFP@z@Frc@iii
  \MFP@multiplyfour \MFP@y@Int@i  \MFP@z@Int@ii
    \MFP@z@Int@i  \MFP@z@Frc@i   \MFP@z@Frc@ii
  \MFP@multiplyfour \MFP@y@Int@ii \MFP@z@Int@iii
    \MFP@z@Int@ii \MFP@z@Int@i   \MFP@z@Frc@i
```

Now apply the carry routines on the underflow digits...

```
  \MFP@carrym\MFP@z@Frc@iv\MFP@z@Frc@iii
  \MFP@carrym\MFP@z@Frc@iii\MFP@z@Frc@ii
```

...and pause to round the lowest digit that will be kept...

```
  \ifnum\MFP@z@Frc@iii<5000 \else
    \MFP@tempb\MFP@z@Frc@ii
    \advance\MFP@tempb1
    \edef\MFP@z@Frc@ii{\number\MFP@tempb}%
  \fi
```

...and continue carrying.

```
  \MFP@carrym\MFP@z@Frc@ii\MFP@z@Frc@i
  \MFP@carrym\MFP@z@Frc@i \MFP@z@Int@i
  \MFP@carrym\MFP@z@Int@i \MFP@z@Int@ii
  \MFP@carrym\MFP@z@Int@ii\MFP@z@Int@iii
```

To end, we arrange for all macros to hold four digits (except `\MFP@z@Int@ii` and `\MFP@z@Int@iii` which don't need leading 0s) and load them into the appropriate 8-digit macros. The underflow digits are stored in `\MFP@z@Und` in case we ever need to examine them (we now do: in our unit conversion routine `\MFP@DPmul`),

and the overflow in \MFP@z@Ovr in case we ever want to implement an overflow error. Theoretically $z \neq 0$, but it is possible that $z = 0$ after reducing to eight places. If so, we must reset \MFP@z@Sgn.

```
\makeMFP@fourdigits\MFP@z@Frc@iv
\makeMFP@fourdigits\MFP@z@Frc@iii
\makeMFP@fourdigits\MFP@z@Frc@ii
\makeMFP@fourdigits\MFP@z@Frc@i
\makeMFP@fourdigits\MFP@z@Int@i
\edef\MFP@z@Int{\number\MFP@z@Int@ii\MFP@z@Int@i}%
\edef\MFP@z@Frc{\MFP@z@Frc@i\MFP@z@Frc@ii}%
\edef\MFP@z@Ovr{\number\MFP@z@Int@iii}%
\edef\MFP@z@Und{\MFP@z@Frc@iii\MFP@z@Frc@iv}%
\ifnum\MFP@z@Int>0
\else\ifnum\MFP@z@Frc>0
\else \def\MFP@z@Sgn{0}%
\fi\fi}%
```

For division, we will obtain the result one digit at a time until the 9th digit after the decimal is found. That 9th will be used to round to eight digits (and stored as underflow). We normalize the denominator by shifting left until the integer part is eight digits. We do the same for the numerator. The integer quotient of the integer parts will be one digit (possibly a 0). If the denominator is shifted d digits left and the numerator n digits left, the quotient will have to be shifted $n - d$ places right or $d - n$ places left. Since the result is supposed to have 9 digits after the dot, our quotient needs $9 + d - n + 1$ total digits. Since d can be as high as 15 and n as low as 0, we could need 25 repetitions. However, that extreme would put 15 or 16 digits in the integer part, a 7 or 8 digit overflow. (It can be argued that only 16 significant digits should be retained in any case.) If d is 0 and n is 15 we would need -5 digits. That means the first nonzero digit is in the 15th or 16th place after the dot and the quotient is effectively zero.

Here I explain why we normalize the parts in this way. If a numerator has the form $n_1.n_2$ and the denominator has the form $d_1.d_2$ then TEX can easily obtain the integer part of n_1/d_1, because these are within its range for integers. The resulting quotient (let's call it q_1) is the largest integer satisfying $q_1 d_1 \leq n_1$. What we seek, however is the largest integer q such that $q(d_1.d_2) \leq n_1.n_2$. It can easily be shown that $q \leq q_1$. It is true, but not so easily shown, that $q \geq q_1 - 1$. This is only true if d_1 is large enough, in our case it has to be at least five digits. Thus we only have to do one simple division and decide if we need to reduce the quotient by one. If we arrange for d_1 to have eight digits, then q_1 will be one digit and the test for whether we need to reduce it becomes easier.

This test is done as follows. The first trial quotient, q_1, will work if

$$q_1 d_1 (10)^8 + q_1 d_2 \leq n_1 (10)^8 + n_2$$

This means

$$0 \leq (n_1 - q_1 d_1)(10)^8 + n_2 - q_1 d_2. \tag{1}$$

Since d_2 is no more than eight digits, $q_1 d_2$ is less than $9(10)^8$. Inequality (1) is therefore satisfied if $n_1 - q_1 d_1 \geq 9$. If that is not the case then the right side of (1)

is computable within TEX's integer ranges and we can easily test the inequality. If the inequality holds, then $q = q_1$, otherwise $q = q_1 - 1$.

Note also that when $q = q_1$, then both terms in (1) (ignoring the 10^8 factor) will be needed to calculate the remainder. If $q = q_1 - 1$, we simply add d_1 and d_2 to the respective parts. Thus we will save these values for that use.

Now I need to get it organized. `\MFP@Rdiv` will have `\MFP@x@*` and `\MFP@y@*` available. One step (could be first or last). Is to calculate the sign. Let's do it first (because we need to check for zero anyway).

We invoke an error message upon division by zero, but nevertheless return a value. By default it is 0 for $0/0$ and the maximum possible real for $x/0$ when x is not zero. If the numerator is zero and the denominator not, we do nothing as z was initialized to be zero.

If neither is zero, we calculate the sign of the result and call `\MFP@@Rdiv` to divide the absolute values.

```
\def\MFP@Rdiv{%
  \ifnum\MFP@y@Sgn=0 \MFP@dividebyzero@err
    \ifnum\MFP@x@Sgn=0
      \edef\MFP@z@Int{\ZeroOverZeroInt}%
      \edef\MFP@z@Frc{\ZeroOverZeroFrac}%
    \else
      \edef\MFP@z@Int{\xOverZeroInt}%
      \edef\MFP@z@Frc{\xOverZeroFrac}%
    \fi
    \edef\MFP@z@Sgn{\MFP@x@Sgn}%
  \else\ifnum\MFP@x@Sgn=0 \MFP@Rzero
  \else \edef\MFP@z@Sgn{\number\MFP@x@Sign\MFP@y@Sgn}\MFP@@Rdiv
  \fi\fi}%
```

Now we have two positive values to divide. Our first step is to shift the denominator (y) left and keep track of how many places. We store the shift in `\MFP@tempa`. This actually changes the value of y, but knowing the shift will give us the correct quotient in the end.

We first arrange that `\MFP@y@Int` is nonzero by making it `\MFP@y@Frc` if it is zero (a shift of eight digits). Then the macro `\MFP@numdigits@toshift` computes 8 minus the number of digits in `\MFP@y@Int`, which is how many positions left y will be shifted. We then call `\MFP@doshift@y` on the concatenation of the digits in the integer and fractional parts (padded with zeros to ensure there are at least 16). All this macro does is read the first eight digits into `\MFP@y@Int` and the next eight into `\MFP@y@Frc`.

```
\def\MFP@@Rdiv{%
  \ifnum\MFP@y@Int=0
    \edef\MFP@y@Int{\number\MFP@y@Frc}%
    \def\MFP@y@Frc{00000000}%
    \MFP@tempa=8
  \else
    \MFP@tempa=0
  \fi
  \advance\MFP@tempa\MFP@numdigits@toshift\MFP@y@Int\relax
```

```
  \@XP\MFP@doshift@y\@xp\MFP@y@Int\MFP@y@Frc0000000\MFP@end
```

We repeat all that on the numerator x, except shifting its digits left means the final outcome will need a corresponding *right* shift. We record that fact by reducing `\MFP@tempa`, which ends up holding the net shift necesary.

This has the advantage that we know the result will be in the range $[0.1, 10)$. It also means we can reduce the number of places we will need to shift left as well as reduce the number of iterations of the loop that calculates the digits.

```
  \ifnum\MFP@x@Int=0
    \edef\MFP@x@Int{\number\MFP@x@Frc}%
    \def\MFP@x@Frc{00000000}%
    \advance\MFP@tempa -8
  \fi
  \advance\MFP@tempa-\MFP@numdigits@toshift\MFP@x@Int\relax
  \@XP\MFP@doshift@x\@xp\MFP@x@Int\MFP@x@Frc0000000\MFP@end
```

Since our result will have at most one digit in the integer part, a rightward shift of 10 places will make every digit 0, including the rounding digit, so we do nothing (returning 0).

```
  \ifnum\MFP@tempa<-9
  \else
```

Now we perform the division, which is a loop repeated $10 +$ `\MFP@tempa` times. Therefore, we add 10 to `\MFP@tempa` in `\MFP@tempf`, our loop counter. We also initialize the macro that will store the digits and then, after the division, shift and split it into parts.

```
    \MFP@tempf\MFP@tempa
    \advance\MFP@tempf 10
    \def\MFP@z@digits{}%
    \MFP@Rdivloop
    \MFPshiftandsplit@z@digits
```

The last remaining step is to round and carry and get the fractional part in the appropriate 8-digit form..

```
    \ifnum\MFP@z@Und>4
      \MFP@addone\MFP@z@Frc1\MFP@z@Frc
      \MFP@carrya\MFP@z@Frc\MFP@z@Int
      \MFP@carrya\MFP@z@Int\MFP@z@Ovr
      \makeMFP@eightdigits\MFP@z@Frc
    \fi
  \fi}%
```

If `#1` of `\MFP@numdigits@toshift`, has n digits then `\MFP@numdigits@toshift` picks out the value $8 - n$. `\MFP@doshift@x` reads the first eight digits into `\MFP@x@Int` and then pulls out eight more from the rest (`#9`) inside `\MFP@x@Frc`. The same with `\MFP@doshift@y`.

```
\def\MFP@numdigits@toshift#1{\@xp\MFP@ninthofmany#101234567\MFP@end}%
\def\MFP@doshift@x#1#2#3#4#5#6#7#8#9\MFP@end{%
  \def\MFP@x@Int{#1#2#3#4#5#6#7#8}%
  \edef\MFP@x@Frc{\MFP@eightofmany#9\MFP@end}}%
```

```
\def\MFP@doshift@y#1#2#3#4#5#6#7#8#9\MFP@end{%
  \def \MFP@y@Int{#1#2#3#4#5#6#7#8}%
  \edef\MFP@y@Frc{\MFP@eightofmany#9\MFP@end}}%
```

The loop counter is `\MFP@tempf`, `\MFP@tempa` is reserved for the shift required later, the quotient digit will be `\MFP@tempb`. The remainder will be calculated in `\MFP@tempc` and `\MFP@tempd`. `\MFP@tempe` will hold the value whose size determines whether the quotient needs to be reduced.

```
\def\MFP@Rdivloop{%
  \MFP@tempb\MFP@x@Int                    % \MFP@tempb = n_1
  \MFP@tempc\MFP@y@Int                    % \MFP@tempc = d_1
  \divide\MFP@tempb \MFP@tempc        % \MFP@tempb = n_1/d_1 = q_1
  \multiply \MFP@tempc \MFP@tempb     % \MFP@tempc = q_1 d_1
  \MFP@tempd \MFP@y@Frc                   % \MFP@tempd = d_2
  \multiply \MFP@tempd \MFP@tempb     % \MFP@tempd = q_1 d_2
  \MFP@tempe \MFP@tempc
  \advance \MFP@tempe -\MFP@x@Int\relax   % \MFP@tempe = -n_1 + q_1 d_1
  \ifnum \MFP@tempe > -9              % n_1 - q_1 d_1 < 9
    \multiply \MFP@tempe\MFP@ttteight    % -(n_1 - q_1 d_1)(10)^8
    \advance \MFP@tempe \MFP@tempd       % add q_1 d_2
    \advance \MFP@tempe -\MFP@x@Frc\relax % add -n_2
    \ifnum \MFP@tempe>0               % Crucial inequality fails
      \advance\MFP@tempb -1           % new q = q_1 - 1
      \advance\MFP@tempc -\MFP@y@Int      % q_1 d_1 - d_1 = q d_1
      \advance\MFP@tempd -\MFP@y@Frc\relax% q_1 d_2 - d_2 = q d_2
    \fi
  \fi
  \edef\MFP@z@digits{\MFP@z@digits\number\MFP@tempb}%
```

It remains to:

- Do the carry from `\MFP@tempd` to `\MFP@tempc`. Then `\MFP@tempc.\MFP@tempd` will represent $q \cdot y$.
- Subtract them from `\MFP@x@Int` and `\MFP@x@Frc` (i.e. remainder $= x - qy$).
- Borrow, if needed, and we will have the remainder in `\MFP@x@Int.\MFP@x@Frc`.

Then we decrement the loop counter, and decide whether to repeat this loop. If so, we need to shift the remainder right one digit (multiply by 10). We don't use `\MFP@carrya` since it requires macros; its internal code, `\MFP@carryi` just leaves the results in `\MFP@tempa.\MFP@tempb`.

```
  \begingroup
    \MFP@carryi\MFP@ttteight\MFP@tempd\MFP@tempc
  \MFP@endgroup@after{%
    \MFP@tempc=\number\MFP@tempa
    \MFP@tempd=\number\MFP@tempb\relax
  }%
% subtract
  \MFP@addone\MFP@x@Int{-\MFP@tempc}\MFP@x@Int
  \MFP@addone\MFP@x@Frc{-\MFP@tempd}\MFP@x@Frc
% borrow
  \ifnum\MFP@x@Frc<0
```

```
    \MFP@addone\MFP@x@Frc\MFP@ttteight\MFP@x@Frc
    \MFP@addone\MFP@x@Int{-1}\MFP@x@Int
  \fi
  \advance\MFP@tempf -1
  \ifnum\MFP@tempf>0
    \edef\MFP@x@Int{\MFP@x@Int0}%
    \edef\MFP@x@Frc{\MFP@x@Frc0}%
    \MFP@carrya\MFP@x@Frc\MFP@x@Int
    \@xp\MFP@Rdivloop
  \fi}%
```

Now `\MFPshiftandsplit@z@digits`. At this point, the digits of the quotient are stored in `\MFP@z@digits`. We need to shift the decimal `\MFP@tempa` places left, and perform the rounding. There are `\MFP@tempa` + 10 digits. This could be as little as 1 or as great as 25. In the first case `\MFP@tempa` is −9, and this (rightward) shift produces 0 plus a rounding digit. In the latter case `\MFP@tempa` is 15, and the shift produces 8 digits overflow, an 8-digit integer part, an 8-digit fractional part and a rounding digit. In the example 0123456, `\MFP@tempa` + 10 is 7, so `\MFP@tempa` is −3. The shift produces 0.0001 2345 6. The rounding digit (6) makes the answer 0.0001 2346.

We take two cases:

- `\MFP@tempa` ≤ 7, prepend 7 − `\MFP@tempa` zeros. The first 8 digits will become the integer part, and there should be exactly 9 more digits.
- `\MFP@tempa` > 7, pluck `\MFP@tempa` − 7 digits for overflow, the next 8 for integer part, leaving 9 more digits

In either case, the 9 last digits will be processed into a fractional part (with possible carry if the rounding increases it to 10^8).

After this, we will return to `\MFP@Rdiv` so overwriting `\MFP@temp*` won't cause any problems.

```
\def\MFPshiftandsplit@z@digits{%
    \advance \MFP@tempa -7
    \ifnum\MFP@tempa>0
      \def\MFP@z@Ovr{}%
      \@xp\MFPget@Ovrdigits\MFP@z@digits\MFP@end
    \else
      \ifnum\MFP@tempa<-7
        \edef\MFP@z@digits{00000000\MFP@z@digits}%
        \advance\MFP@tempa8
      \fi
      \ifnum\MFP@tempa<-3
        \edef\MFP@z@digits{0000\MFP@z@digits}%
        \advance\MFP@tempa4
      \fi
      \edef\MFP@z@digits{%
        \ifcase-\MFP@tempa\or
          0\or
          00\or
          000\or
```

```
          0000\else
          00000%
        \fi \MFP@z@digits}%
      \@xp\MFPget@Intdigits\MFP@z@digits\MFP@end
    \fi}%
```

The macro `\MFPget@Ovrdigits` is a loop that loads the first `\MFP@tempa` digits of what follows into `\MFP@z@Ovr`. It does this one digit (`#1`) at a time. Once the counter reaches 0, we call the macro that processes the integer part digits.

```
\def\MFPget@Ovrdigits#1{%
  \edef\MFP@z@Ovr{\MFP@z@Ovr#1}%
  \advance\MFP@tempa -1
  \ifnum\MFP@tempa>0
    \@xp\MFPget@Ovrdigits
  \else
    \@xp\MFPget@Intdigits
  \fi}%
```

The macro `\MFPget@Intdigits` should have exactly 17 digits following it. It puts eight of them in `\MFP@z@Int`, then calls `\MFPget@Frcdigits` to read the fractional part. That requires exactly nine digits follow it, putting eight in `\MFP@z@Frc` and the last in `\MFP@z@Und`. Still, to allow a graceful exit should there be more, we gobble the rest of the digits.

```
\def\MFPget@Intdigits#1#2#3#4#5#6#7#8{%
  \def\MFP@z@Int{\number#1#2#3#4#5#6#7#8}%
  \MFPget@Frcdigits}%
\def\MFPget@Frcdigits#1#2#3#4#5#6#7#8#9{%
  \def\MFP@z@Frc{#1#2#3#4#5#6#7#8}%
  \def\MFP@z@Und{#9}\gobbleto@MFP@end}%
```

The max amd min operations simply run the compare operation and use and use the resultant booleans to copy x or y to z.

```
\def\MFP@Rmax{%
  \MFP@Rcmp \ifMFP@neg \MFP@Rcopy yz\else\MFP@Rcopy xz\fi}%
\def\MFP@Rmin{%
  \MFP@Rcmp \ifMFP@pos \MFP@Rcopy yz\else\MFP@Rcopy xz\fi}%
```

3.5 Commands to format for printing

`\MFPtruncate` This first runs the parsing command so the fractional part has exactly eight digits. These become the arguments of `\MFP@@Rtrunc`, which just keeps the right number. For negative truncations we prepend zeros to the integer part so it too is exactly eight digits. These become the arguments of `\MFP@@iRtrunc`, which substitutes 0 for the last `-\MFP@tempa` of them.

The macro to store the result in follows `#2`. It is read and defined by either `\MFP@Rtrunc` or `\MFP@iRtrunc`.

```
\def\MFPtruncate#1#2{%
  \begingroup
    \MFP@tempa#1\relax
```

```
    \MFPparse@x{#2}%
    \ifnum\MFP@tempa<1
      \@xp\MFP@iRtrunc
    \else
      \@xp\MFP@Rtrunc
    \fi}%
\def\MFP@Rtrunc#1{%
    \edef\MFP@x@Frc{\@xp\MFP@@Rtrunc\MFP@x@Frc\MFP@end}%
    \ifnum\MFP@x@Int=0
      \ifnum\MFP@x@Frc=0
        \def\MFP@x@Sgn{0}%
      \fi
    \fi
  \MFP@endgroup@after{%
    \MFP@afterdef#1{\MFP@x@Sign\MFP@x@Int.\MFP@x@Frc}}}%
\def\MFP@@Rtrunc#1#2#3#4#5#6#7#8#9\MFP@end{%
  \ifcase\MFP@tempa\or
    #1\or
    #1#2\or
    #1#2#3\or
    #1#2#3#4\or
    #1#2#3#4#5\or
    #1#2#3#4#5#6\or
    #1#2#3#4#5#6#7\else
    #1#2#3#4#5#6#7#8\fi}%
\def\MFP@iRtrunc#1{%
  \makeMFP@eightdigits\MFP@x@Int
  \edef\MFP@x@Val{\number\MFP@x@Sign\@xp\MFP@@iRtrunc\MFP@x@Int\MFP@end}%
  \MFP@endgroup@after{\MFP@afterdef#1{\MFP@x@Val}}}%
\def\MFP@@iRtrunc#1#2#3#4#5#6#7#8#9\MFP@end{%
  \ifcase-\MFP@tempa
    #1#2#3#4#5#6#7#8\or
    #1#2#3#4#5#6#70\or
    #1#2#3#4#5#600\or
    #1#2#3#4#5000\or
    #1#2#3#40000\or
    #1#2#300000\or
    #1#2000000\or
    #10000000\else
    00000000\fi}%
```

`\MFPround` For rounding we simply add the appropriate fraction and truncate. The macro in which to store the result will follow `#2`, and be picked up by the `\MFPtruncate` command.

```
\def\MFPround#1#2{%
  \begingroup
    \MFP@tempa#1\relax
    \ifnum 0>\MFP@tempa
      \edef\MFP@y@Tmp{%
        \ifcase-\MFP@tempa\or
```

```
          5\or
          50\or
          500\or
          5000\or
          50000\or
          500000\or
          5000000\else
          50000000\fi
      }%
    \else
      \edef\MFP@y@Tmp{%
        \ifcase\MFP@tempa
          .5\or
          .05\or
          .005\or
          .0005\or
          .00005\or
          .000005\or
          .0000005\or
          .00000005\else
        0\fi
      }%
    \fi
    \MFPchk{#2}\ifMFP@neg\edef\MFP@y@Tmp{-\MFP@y@Tmp}\fi
    \MFPadd{#2}\MFP@y@Tmp\MFP@z@Tmp
  \MFP@endgroup@after{\MFP@afterdef\MFP@z@Tmp{\MFP@z@Tmp}}%
  \MFPtruncate{#1}\MFP@z@Tmp}%
```

\MFPstrip Stripping zeros from the right end of the fractional part. The star form differs only in the handling of a zero fractional part. So we check whether it is zero and when it is, we either append '.0' or nothing. The rest of the code grabs a digit at a time and stops when the rest are zero.

```
\def\MFPstrip{%
  \@ifstar{\MFP@strip{}}{\MFP@strip{.0}}}%
\def\MFP@strip#1#2#3{%
  \MFPparse@x{#2}%
  \ifnum \MFP@x@Frc=0
    \edef#3{\MFP@x@Sign\MFP@x@Int#1}%
  \else
    \edef#3{\MFP@x@Sign\MFP@x@Int.\@xp\MFP@@strip\MFP@x@Frc\MFP@end}%
  \fi}%
\def\MFP@@strip#1#2\MFP@end{%
  #1%
  \ifnum 0#2>0
    \@xp\MFP@@strip
  \else
    \@xp\gobbleto@MFP@end
  \fi#2\MFP@end}%
```

3.6 Miscellaneous

Here is the code that allows definitions to survive after `\stopMFPprogram`. The `\Global` variants are easiest.

```
\def\MFP@Global#1{\toks@\@xp{#1}\xdef#1{\the\toks@}}%
\def\MFP@GlobalStack{\MFP@Global\MFP@Rstack}%
```

The `\Export` command adds the command and its definition to a macro that is executed after the closing group of the program.

```
\def\MFP@Export#1{%
  \begingroup
    \toks@\@xp{\MFPprogram@returns}%
  \MFP@endgroup@after{%
    \MFP@afterdef\MFPprogram@returns{\the\toks@ \MFP@afterdef#1{#1}}%
  }}%
\def\MFP@ExportStack{\MFP@Export\MFP@Rstack}%
```

The various operations `\MFP@R...` together make up a "microcode" in terms of which the stack language and the operand language are both defined. As a language in its own right, it lacks only convenient ways to move numbers around, as well as a few extra registers for saving intermediate results. In this language, numbers are represented by a three part data structure, consisting of a signum, an integer part and a fractional part.

Here we define extra commands to remedy this lack, starting with a way to load a number (or rather, a three part data structure representing a number) directly into a register. Here `#1` is a register name (we always us a single letter) and the remaining arguments are the signum, the integer part and the fractional part (automatically normalized to 8 digits). The "register" is just a set of three macros created from the name given.

We make loading a number into a register a little more general than strictly needed, allowing the parts to be specified as anything TEX recognizes as a number and allowing any register name. This generality might reduce efficiency but it simplifies code. Because register z is by far the most common one to load, we make more efficient version of it.

```
\def\MFP@Rload #1#2#3#4{%
  \@xp\edef\csname MFP@#1@Sgn\endcsname{\number#2}%
  \@xp\edef\csname MFP@#1@Int\endcsname{\number#3}%
  \@xp\edef\csname MFP@#1@Frc\endcsname{\number#4}%
  \@xp\makeMFP@eightdigits\csname MFP@#1@Frc\endcsname}%
\def\MFP@Rcopy#1#2{%
  \MFP@Rload #2{\csname MFP@#1@Sgn\endcsname}%
               {\csname MFP@#1@Int\endcsname}%
               {\csname MFP@#1@Frc\endcsname}}%
\def\MFP@Rloadz#1#2#3{%
  \edef\MFP@z@Sgn{\number#1}%
  \edef\MFP@z@Int{\number#2}%
  \edef\MFP@z@Frc{\number#3}%
  \makeMFP@eightdigits\MFP@z@Frc}%
```

`\MFPpi` These are some miscellaneous constants. The 8-digit approximation to π, is

\MFPe \MFPpi and the constant mathematicians call e is \MFPe. Finally, the golden ratio
\MFPphi (often called ϕ) is obtained by \MFPphi.

```
\def\MFPpi{3.14159265}%
\def\MFPe{2.71828183}%
\def\MFPphi{1.61803399}%
```

Load (conditionally) mfpextra.tex.

```
\MFP@loadextra
\MFP@finish
⟨/sty⟩
```

4 Extras

The extras consist so far of sine, cosine, angle, logarithm, powers, and square root. For completeness, here is the table of user-level commands available.

Operand versions

Command	**operation**
\MFPsin{⟨*num*⟩}\macro	Stores sin(⟨*num*⟩) in \macro, where ⟨*num*⟩ is an angle in degrees.
\MFPcos{⟨*num*⟩}\macro	Stores cos(⟨*num*⟩) in \macro, where ⟨*num*⟩ is an angle in degrees.
\MFPangle{⟨x⟩}{⟨y⟩}\macro	Stores in \macro the polar angle coordinate θ of the point (x, y), where $-180 < \theta \leq 180$.
\MFPrad{⟨*num*⟩}\macro	The angle ⟨*num*⟩ in degrees is converted to radians, and result is stored in \macro.
\MFPdeg{⟨*num*⟩}\macro	The angle ⟨*num*⟩ in radians is converted to degrees, and result is stored in \macro.
\MFPlog{⟨*num*⟩}\macro	Stores log(⟨*num*⟩) in \macro (base 10 logarithm).
\MFPln{⟨*num*⟩}\macro	Stores ln(⟨*num*⟩) in \macro (natural logarithm).
\MFPexp{⟨*num*⟩}\macro	Stores exp(⟨*num*⟩) (i.e., e^x) in \macro.
\MFPsqrt{⟨*num*⟩}\macro	Stores the square root of ⟨*num*⟩ in \macro.
\MFPpow{⟨*num*⟩}{⟨*int*⟩}\macro	Stores the ⟨*int*⟩ power of ⟨*num*⟩ in \macro. The second operand must be an integer (positive or negative).

Command	operation
`\Rsin`	The number is interpreted as degrees, and its sine is computed.
`\Rcos`	The number is interpreted as degrees, and its cosine is computed.
`\Rangle`	The top two numbers are interpreted as coordinates of a point P in the order they were pushed. The polar angle coordinate θ of P, with $-180 < \theta \leq 180$ is computed.
`\Rrad`	The number of degrees is converted to radians.
`\Rdeg`	The number of radians is converted to degrees.
`\Rlog`	Computes the base-10 logarithm.
`\Rln`	Computes the natural logarithm.
`\Rexp`	Computes the exponential of the number (i.e., e^x).
`\Rsqrt`	Computes the square root of the number.
`\Rpow`	Computes x^y. The last number pushed (y) must be an integer.

The user could easily convert between radians and degrees using multiplication and/or division. One could similarly convert between natural logarithms and base ten logarithms. The commands `\Rdeg`, `\Rrad`, `\Rlog` and `\Rln` (and their `\MFP...` counterparts) aim for more accurate results.

4.1 Loading the extras

`\Rsin` `\Rcos` `\Rangle` `\Rrad` `\Rdeg` `\Rlog` `\Rln` `\Rexp` `\Rsqrt` `\Rpow`

We start `mfpextra` with the hook `\MFP@Rextra` that `\startMFPprogram` will call to make available the extra operations defined here. If `minifp.sty` has been loaded, this macro is `\@empty`, otherwise it should be undefined. If it is undefined we load `minifp.sty`. If it is then not `\@empty` we assume `mfpextra.tex` was previously loaded and end input here.

```
⟨*extra⟩
% check if mfpextra already loaded:
\expandafter\ifx\csname MFP@xfinish\endcsname\relax
\else \expandafter\endinput\fi
\expandafter\edef\csname MFP@xfinish\endcsname{%
  \catcode64=\the\catcode64 \space
  \catcode46=\the\catcode46 \space
  \catcode60=\the\catcode60 \space
  \catcode62=\the\catcode62 \space}%
\catcode64=11 % @
\catcode46=12 % . (period)
\catcode60=12 % <
\catcode62=12 % >
\ifx\MFP@Rextra\UndEfInEd \input minifp.sty \fi
\ifx\MFP@Rextra\@empty
\else
  \immediate\write16{mfpextra.tex: already loaded.^^J}%
  \MFP@xfinish
  \expandafter\endinput
\fi
```

```
\immediate\write16{%
    mfpextra.tex: extra operations for MiniFP package.^^J}%
\def\MFP@Rextra{%
  \def\Rcos  {\MFP@stack@Unary\MFP@Rcos }%
  \def\Rsin  {\MFP@stack@Unary\MFP@Rsin }%
  \def\Rangle{\MFP@stack@Binary\MFP@Rangle}%
  \def\Rrad  {\MFP@stack@Unary\MFP@Rrad }%
  \def\Rdeg  {\MFP@stack@Unary\MFP@Rdeg }%
  \def\Rlog  {\MFP@stack@Unary\MFP@Rlog }%
  \def\Rln   {\MFP@stack@Unary\MFP@Rln  }%
  \def\Rexp  {\MFP@stack@Unary\MFP@Rexp }%
  \def\Rsqrt {\MFP@stack@Unary\MFP@Rsqrt}%
  \def\Rpow  {\MFP@stack@Binary\MFP@Rpow}}%
```

\MFPsin \MFPcos \MFPrad \MFPdeg \MFPlog \MFPln \MFPexp \MFPsqrt \MFPpow

Then the wrappers for the operand versions.

```
\def\MFPcos   {\MFP@op@Unary\MFP@Rcos }%
\def\MFPsin   {\MFP@op@Unary\MFP@Rsin }%
\def\MFPangle {\MFP@op@Binary\MFP@Rangle}%
\def\MFPrad   {\MFP@op@Unary\MFP@Rrad }%
\def\MFPdeg   {\MFP@op@Unary\MFP@Rdeg }%
\def\MFPlog   {\MFP@op@Unary\MFP@Rlog }%
\def\MFPln    {\MFP@op@Unary\MFP@Rln  }%
\def\MFPexp   {\MFP@op@Unary\MFP@Rexp }%
\def\MFPsqrt  {\MFP@op@Unary\MFP@Rsqrt}%
\def\MFPpow   {\MFP@op@Binary\MFP@Rpow}%
```

4.2 Error messages

These extra commands come with a few possible new warnings and errors.

\LogOfZeroInt \LogOfZeroFrac

Trying to take the logarithm of zero will result in an error message. If one allows TEX to continue, the returned value will be negative, with an integer part whose absolute value is equal to the contents of `\LogOfZeroInt` and a fractional part equal to the contents of `\LogOfZeroFrac`. The defaults are both 99999999.

Trying to take the logarithm of a negative number will produce the warning

```
MFP warning: Log of a negative number is complex.
             Only the real part will be computed.
```

The log of the absolute value is returned.

Trying to take the square root of a negative number has similar behavior. It produces a warning and returns 0.

Trying to take the exponential of a number larger than about 18.42 will cause an error and the number returned has integer part 99999999 and fractional part 99999999.

Trying to take a negative power of 0 produces an error and returns the same value as trying to divide 1 by 0.

Messages for errors related to impossible powers and logarithms.

```
\def\MFP@logofzero@err{%
  \MFP@errmsg{logarithm of zero}%
    {You tried to take the logarithm of zero. What were you %
```

```
804     thinking? If you ^^Jcontinue, the value %
805     assigned will be -\LogOfZeroInt.\LogOfZeroFrac.}}%
806 \def\LogOfZeroInt {\MaxRealInt}%
807 \def\LogOfZeroFrac{\MaxRealFrac}%
808 \def\MFP@expoverflow@err{%
809   \MFP@errmsg{Power too large}%
810     {The power you tried to calculate is too large for %
811      8 digits. If you continue, ^^Jthe value assigned will be %
812      \MaxRealInt.\MaxRealFrac.}}%
813 \def\MFP@badpower@err{%
814   \MFP@errmsg{negative power of zero}%
815     {You tried to take a negative power of zero. What were you
816      thinking? If you ^^Jcontinue, the value assigned will be %
817      \xOverZeroInt.\xOverZeroFrac.}}%
```

A debugging utility, `\MFPshowreg` displays the contents of a register.

```
818 \def\MFPshowreg #1{%
819 \ifMFPdebug
820 \begingroup
821   \edef\theregister{%
822     #1 =  \expandafter \MFP@Sign
823           \csname MFP@#1@Sgn\endcsname %
824           \csname MFP@#1@Int\endcsname.%
825           \csname MFP@#1@Frc\endcsname}%
826   \show\theregister
827 \endgroup
828 \fi}%
```

4.3 Sine and Cosine

For iterated code, the most common register to copy is z and the most common place to copy it is to x or y so we make single commands to do those.

```
829 \def\MFP@Rcopyz#1{\MFP@Rload {#1}\MFP@z@Sgn\MFP@z@Int\MFP@z@Frc}%
830 \def\MFP@Rcopyzx{\MFP@Rcopyz x}%
831 \def\MFP@Rcopyzy{\MFP@Rcopyz y}%
```

Our code assumes the number x is an angle in degrees. To get sine and cosine of numbers as radians, simply convert your radians to degrees using `\MFPdeg` or `\Rdeg`. Then find the sine or cosine of the result. For example, if `\X` holds the angle in in radians and you want the result to be stored in `\S`:

```
\MFPdeg\X\Y \MFPsin\Y\S
```

For unit conversions such as radian to degree we try to be more accurate than a multiplication by an eight-digit conversion factor allows. If x is large and the factor is off by 0.5×10^{-8}, then the result can be significantly off. But if we are able to give the conversion factor 16 digits precision, then only the imprecision of x will significantly affect the result.

We express the conversion factor as an integer part and two eight-digit fractional parts. We multiply x by the integer and first fractional part (`#1` and `#2`) with a normal `\MFP@Rmul`, but we save the underflow digits and undo the rounding that

occured at the 8th digit. Together these give us an essentially exact result. Then we multiply x by the second fractional part (`#3`) and add the saved underflow to the result. Finally, we round and add the result to the first product. Argument `#3`, as well as the underflow digits, represent numbers less than 10^{-8}, so we effectively scale them up by 10^8, round the result to an integer and scale that back down.

The registers w and v are used to save intermediate results. The "`DP`" in `\MFP@DPmul` refers to the fact that we are multiplying by a "double precision" real.

```
\def\MFP@DPmul#1#2#3{%
  \ifnum\MFP@x@Sgn=0
    \MFP@Rzero
  \else
    \MFP@Rcopy xv%
    \MFP@Rload y1{#1}{#2}\MFP@Rmul
    \edef\MFP@w@Und{\MFP@z@Und}%
    \ifnum\MFP@z@Frc@iii>4999
      \MFP@tempa\MFP@z@Frc \advance\MFP@tempa-1
      \edef\MFP@z@Frc{\number\MFP@tempa}%
      \makeMFP@eightdigits\MFP@z@Frc
    \fi
    \MFP@Rcopyz w%
    \MFP@Rcopy vx\MFP@Rload y10{#3}\MFP@Rmul
    \MFP@Rcopyzx\MFP@Rload y\MFP@v@Sgn 0{\MFP@w@Und}\MFP@Radd
    \MFP@tempa\MFP@z@Int\relax
    \ifnum\MFP@z@Frc<50000000 \else \advance\MFP@tempa 1 \fi
    \ifnum\MFP@tempa<\MFP@ttteight\relax
      \MFP@Rload x{\ifnum\MFP@tempa>0 \MFP@z@Sgn\else0\fi}0\MFP@tempa
    \else
      \MFP@Rload x\MFP@z@Sgn10%
    \fi
    \MFP@Rcopy wy\MFP@Radd
  \fi}%
```

Conversion factors:

- radians to degrees: 57.2957795130823209
- degrees to radians: 0.0174532925199433
- natural log to common log: 0.4342944819032518
- common log to natural log: 2.3025850929940457

Note that the comparatively large size of the first number means that the $\pm 0.5 \cdot 10^{-8}$ imprecision that x implicitly carries will be multiplied to approximately $\pm 29.6 \cdot 10^{-8}$ in the result. The only way around this would be to operate with higher precision internally. We do that in the code for computing angles.

```
\def\MFP@Rdeg{\MFP@DPmul{57}{29577951}{30823209}}%
\def\MFP@Rrad{\MFP@DPmul{0}{01745329}{25199433}}%
\def\MFP@RbaseX{\MFP@DPmul{0}{43429448}{19032518}}%
\def\MFP@RbaseE{\MFP@DPmul{2}{30258509}{29940457}}%
```

There are very few angles that are expressible in eight digits whose sine or cosine can be expressed exactly in eight digits. For these, we do obtain an exact

result. Other values produce inexact results. It would be nice if we could at least obtain these correctly rounded to eight decimals, but unfortunately our methods will often produce a result off by 1 in the eighth decimal from the correctly rounded value. Anything that involves the addition of two or more rounded results can have this problem. The only way to get correctly rounded results is to carry out all operations internally to additional places. Even then, there will be the occasional .4999... that should round to 0 but rounds to 1 instead.

For the cosine, just compute $\sin(90 - x)$.

```
\def\MFP@Rcos{%
  \MFP@Rcopy xy\MFP@Rload x1{90}0\MFP@Rsub
  \MFP@Rcopyzx\MFP@Rsin}%
```

Reduce $|x|$ by subtracting 180 from the integer part until it is less than 180. Of course, $\sin x = \operatorname{sgn}(x)\sin(|x|)$ so we only need to compute $\sin(|x|)$. The sign will be that of x; each reduction by 180 changes the sign, but the reduction code keeps track of that. If x is 0 after the reduction, return zero.

```
\def\MFP@Rsin{%
  \MFP@tempa\MFP@x@Int
  \MFP@tempb\MFP@x@Frc
  \MFP@tempc\MFP@x@Sgn\relax
  \MFP@reduce@angle
  \ifnum\MFP@tempa>0  \MFP@@Rsin
  \else\ifnum\MFP@tempb>0  \MFP@@Rsin
  \else  \MFP@Rzero
  \fi\fi}%
```

This following reduces $|x|$ to the case $0 \le |x| < 180$. It assumes the integer part is in count register `\MFP@tempa`, the sign in `\MFP@tempc`.

```
\def\MFP@reduce@angle{%
  \ifnum\MFP@tempa<180
  \else
    \advance\MFP@tempa-180
    \MFP@tempc-\MFP@tempc
    \@xp\MFP@reduce@angle
  \fi}%
```

At this point, $|x|$ is represented by `\MFP@tempa` (integer part) and `\MFP@tempb` (fractional part). Also, we already know the sign stored in `\MFP@tempc`. Moreover $0 <$ `\MFP@tempa` < 180. We now reduce to $0 < |x| \le 90$ using $\sin(x) = \sin(180 - |x|)$, and return 1 if equal to 90.

The calculation of $180 - x$ is optimized, taking advantage of the fact that both x and the result are known to be positive. If the fractional part is positive, we borrow 1 by reducing 180 to 179.

```
\def\MFP@@Rsin{%
  \ifnum\MFP@tempa<90
  \else
    \MFP@tempa -\MFP@tempa
    \ifnum\MFP@tempb>0
      \MFP@tempb -\MFP@tempb
```

```
      \advance\MFP@tempb \MFP@ttteight\relax
      \advance\MFP@tempa 179
    \else  \advance\MFP@tempa 180
    \fi
  \fi
  \ifnum\MFP@tempa=90
    \MFP@Rloadz \MFP@tempc10%
  \else
```

We would need to convert x to radians (multiply by $\pi/180$) to use the standard power series, but instead we will incorporate the conversion factor into the power series coefficients.

We will, however, try to increase accuracy by reducing the size of x and correspondingly increasing the appropriate factors. Since the number of significant figures of a product is limited by the least number of significant figures of the two factors, the bottleneck on accuracy is that of the smaller term: all our numbers have eight digits so if a number is small, the number of nonzero digits is small.

Dividing by 100 seems a good choice (so our units are "hectodegrees"). This makes $0 < x < .9$ and the integer part (`\MFP@tempa`) will be henceforth ignored.

The addition of 50 is for rounding purposes. After that, our computations amount to concatenating the top six digits of `\MFP@tempb` to the digits of `\MFP@tempa`. This will produce the integer form of the fractional part of $x/100$ (the integer part of $x/100$ is zero).

Division by 100 can turn a number into 0. This is one place we can lose accuracy (up to ± 1 in the last digit of the result). In compensation, the rest of the calculations become very much more accurate.

```
    \advance\MFP@tempb 50 \divide\MFP@tempb 100
    \multiply\MFP@tempa 1000000 \advance\MFP@tempb\MFP@tempa
    \ifnum\MFP@tempb=0
      \MFP@Rzero
    \else
```

We save some multiplications by working with $t = x^2$. As we don't need the original x anymore, we simply replace it with the newly reduced value. We also save this reduced x in another register, s, as we will need it again at the end, and our intermediate calculations do not preserve the x register. Then we square x and, if that square is 0 we can skip all the power series and simply return x converted to radians. If x^2 is not zero, we save it in temporary register t and call our power series. When this program is finished, all that remains is the final multiplication by a conversion factor (`\MFP@DPmul`).

```
      \MFP@Rload s\MFP@tempc0\MFP@tempb
      \MFP@Rcopy sx%
      \MFP@Rsq
      \ifnum \MFP@z@Frc>0
        \MFP@Rcopyz t\MFP@Rsin@prog
      \else
        \MFP@Rcopy sx%
      \fi
```

```
      \MFP@DPmul 1{74532925}{19943296}%
    \fi
  \fi}%
```

`\MFP@Rsin@prog` is the power series computation. The power series need only go to the x^{13} term as the next is less than 10^{-9} and in our 8-place computations is indistingushable from 0. Our series is:

$$rx(1 - r^2t/3! + r^4t^2/5! - r^6t^3/7! + r^8t^4/9! - r^{10}t^5/11! + r^{12}t^6/13!)$$

where r is the factor that converts x to radian measure (hectodegrees to radians). When x is so small as to produce $t = 0$ we have skipped all this.

We minimize any multiplications of tiny numbers by computing this as

$$rx(1 - ft(1 - et(1 - dt(1 - ct(1 - bt(1 - at)))))).$$

In this format, additional terms might actually make a difference, because at is not particularly small. However, the more computations we have, the more errors accumulate. Therefore we take the fewest that produce acceptable accuracy.

Now $r = 1.7453292519943296$ and a, b, etc., have formulas:

$$a = r^2/13/12,\ b = r^2/11/10,\ c = r^2/9/8,$$
$$d = r^2/7/6,\ e = r^2/5/4,\ f = r^2/3/2.$$

An alternative method would be to accumulate a sum, computing each term from the previous one (e.g., if $u = t^3/7!$ is the fourth term, the next one is $u * t * (1/(8 * 9))$). This is a bit more complicated to code and requires moving values around more. It would have the advantage that we can stop whenever a term evaluates to zero, making computation faster for small values of x. I have not determined whether it would compromise accuracy.

We avoid divisions by precomputing the coefficients a, b, c, etc. Note that without the reduction in x, the value of a for example would be 0.00000195, with only three significant figures of accuracy. Now we can have seven, and the accuracy is more-or-less determined by that of the reduced x.

$$a = 0.01952676,\ b = 0.02769249,\ c = 0.04230797,,$$
$$d = 0.07252796,\ e = 0.15230871,\ f = 0.50769570.$$

It is important to note that the following operations step all over the `\MFP@temp`x `\count` registers, so we have made sure that we no longer need them.

The `\MFP@flipz` computes $1-z$, where z is the result of the previous operation. Instead of simply subtracting, we optimize based on the fact that z is known to be nonnegative and not larger than 1.

The macro `\MFP@com@iter` 'flipz' the previous result then multiplies by t and the indicated coefficient. (The name of this macro stands for "common iterated" code; it is reused for some other power series.)

For extra efficiency, the power series uses a "small" version of multiplication `\MFP@Rsmul`, used only when the factors are sure to lie in $[0, 1]$. This does not take into account the sign of x, whence the ending `\edef`.

```
\def\MFP@Rsin@prog{%
  \MFP@Rcopy tx\MFP@Rload y10{01952676}\MFP@Rsmul%
  \MFP@com@iter{02769249}\MFP@com@iter{04230797}\MFP@com@iter{07252796}%
  \MFP@com@iter{15230871}\MFP@com@iter{50769570}\MFP@flipz \MFP@Rcopyzx
  \MFP@Rcopy sy\MFP@Rsmul\MFP@Rcopyzx\edef\MFP@x@Sgn{\MFP@s@Sgn}}%
\def\MFP@flipz{%
  \ifnum\MFP@z@Sgn=0
    \MFP@Rloadz 110%
  \else
      \MFP@tempa\MFP@ttteight
      \advance\MFP@tempa-\MFP@z@Frc\relax
      \MFP@Rloadz{\ifcase\MFP@tempa 0\else1\fi}0\MFP@tempa
  \fi}%
\def\MFP@com@iter#1{\MFP@flipz
  \MFP@Rcopyzx\MFP@Rcopy ty\MFP@Rsmul
  \MFP@Rcopyzx\MFP@Rload y10{#1}\MFP@Rsmul}%
```

As to the accuracy of these computations, we can certainly lose accuracy at each step. In principle, if x is known to 10 significant figures ($x \geq 10$ degrees), then even though we lose two figures with division by 100, the accuracy bottleneck is the fact that our coefficients have only seven figures. Now we have 17 multiplications, and while products are said to have the same number of significant figures as the factors, in the worse case we can accumulate inaccuracy of about $.5 \times 10^{-8}$ per multiplication. So we are not guaranteed an accuracy of more than about $\pm 10^{-7}$. Numerical tests, however, show that it isn't that bad, probably because the direction of inaccuracies usually varies randomly, and inaccuracies in one direction compensate for those going the other way. I have not seen a case where the result is off by more than 1 in the last decimal place (i.e., $\pm 1.5 \times 10^{-8}$). In the case where we can know the result exactly, $x = 30$, we get an exact answer, even though we don't single it out (as we do 0, 90 and 180).

The following is the "small" version of `\MFP@Rmul`. Limited to non-negative numbers less than or equal to 1. Theoretically all the numbers are strictly between 0 and 1, but in practice a multiplication could round to 0 and then, after subtraction, a 1 could occur. We handle those easy cases separately, so that in `\MFP@@Rsmul` we don't have to worry about the integer parts at all.

Also, since these are completely internal, we don't even define the overflow and underflow macros.

```
\def\MFP@Rsmul{%
  \ifnum \MFP@x@Sgn=0 \MFP@Rzero
  \else\ifnum \MFP@y@Sgn=0 \MFP@Rzero
  \else\ifnum\MFP@x@Int>0 \MFP@Rcopy yz%
  \else\ifnum\MFP@y@Int>0 \MFP@Rcopy xz%
  \else \MFP@@Rsmul
  \fi\fi\fi\fi}%
\def\MFP@@Rsmul{%
  \MFP@split\MFP@x@Frc\MFP@x@Frc@i\MFP@x@Frc@ii
  \MFP@split\MFP@y@Frc\MFP@y@Frc@i\MFP@y@Frc@ii
  \def\MFP@z@Frc@i  {0}\def\MFP@z@Frc@ii {0}%
```

```
\def\MFP@z@Frc@iii{0}\def\MFP@z@Frc@iv {0}%
\MFP@tempb\MFP@y@Frc@ii\relax
\MFP@multiplyone\MFP@x@Frc@ii\MFP@z@Frc@iv
\MFP@multiplyone\MFP@x@Frc@i\MFP@z@Frc@iii
\MFP@tempb\MFP@y@Frc@i\relax
\MFP@multiplyone\MFP@x@Frc@ii\MFP@z@Frc@iii
\MFP@multiplyone\MFP@x@Frc@i\MFP@z@Frc@ii
\MFP@carrym\MFP@z@Frc@iv\MFP@z@Frc@iii
\MFP@carrym\MFP@z@Frc@iii\MFP@z@Frc@ii
\ifnum\MFP@z@Frc@iii<5000 \else
  \MFP@tempb\MFP@z@Frc@ii
  \advance\MFP@tempb1
  \edef\MFP@z@Frc@ii{\number\MFP@tempb}\fi
\MFP@carrym\MFP@z@Frc@ii\MFP@z@Frc@i
\makeMFP@fourdigits\MFP@z@Frc@ii
\makeMFP@fourdigits\MFP@z@Frc@i
\def\MFP@z@Int{0}%
\edef\MFP@z@Frc{\MFP@z@Frc@i\MFP@z@Frc@ii}%
\edef\MFP@z@Sgn{\ifnum\MFP@z@Frc=0 0\else 1\fi}}%
```

4.4 Polar angle

Instead of supplying the arcsine and arccosine functions, we supply the more general angle function. This is a binary operation that accepts the two coordinates of a point and computes its angle in polar coordinates. One then has, for example, $\arctan x = \text{angle}(1, x)$ and $\arccos x = \text{angle}(x, \sqrt{1 - x^2})$.

We start, as usual, with a few reductions. When the y-part is 0, we immediately return 0 or 180. If the y-part is negative, we compute the angle for $(x, |y|)$ and negate it. If the x-part is negative, we compute the angle for $|x|$ and subtract it from 180. Finally, reduced to both coordinates positive, if $y > x$ we compute the angle of (y, x) and subtract that from 90. Ultimately, we apply a power series formula for $mathop\text{angle}(1, y/x)$ and get convergence when the argument is less than 1, but convergence is poor unless the argument is less than $1/2$. When that is not the case, conceptually, we rotate the picture clockwise by the arctangent of $1/2$, compute the angle of the new point and then add a precomputed value of $\arctan(1/2)$.

```
\def\MFP@Rangle{%
  \ifcase\MFP@y@Sgn\relax
    \ifcase\MFP@x@Sgn\relax
      \MFP@warn{Point (0,0) has no angle. Returning 0 anyway}%
      \MFP@Rzero
    \or
      \MFP@Rzero
    \else
      \MFP@Rloadz 1{180}0%
    \fi
    \@xp\@gobble
  \or
```

```
    \def\MFP@angle@Sgn{1}\@xp\@firstofone
  \else
    \def\MFP@y@Sgn{1}%
    \def\MFP@angle@Sgn{-1}\@xp\@firstofone
  \fi
 {\ifcase\MFP@x@Sgn\relax
    \MFP@Rloadz1{90}0%
  \or  \MFP@@Rangle
  \else
    \def\MFP@x@Sgn{1}\MFP@@Rangle
    \MFP@Rcopyzy\MFP@Rload x1{180}0\MFP@Rsub
  \fi
  \let\MFP@z@Sgn\MFP@angle@Sgn}}%
\def\MFP@@Rangle{%
  \MFP@Rcmp
  \ifMFP@neg
    \MFP@Rcopy xw\MFP@Rcopy yx\MFP@Rcopy wy%
    \MFP@@@Rangle
    \MFP@Rload x1{90}0\MFP@Rcopyzy\MFP@Rsub
  \else
    \MFP@@@Rangle
  \fi}%
```

Precisely what we do when we are finally in the case $0 < y < x$ is perform a couple of reductions. Ultimately we want to compute the arctan of $z = y/x$. We once again use a power series but, for fast convergence, we require z to be considerably less than 1. For reasons we discuss later, we won't be able to use the more efficient `\MFP@Rsmul` so we want to keep the number of iterations of our power series calculations low.

So we start with two iterations of the algorithm used by Knuth: if $y/x > 1/2$ we transform the pair (x, y) to a new one whose angle has been reduced by $\arctan(1/2)$. The new pair is $(x', y') = (2x + y, 2y - x)$. If we still have $y/x > 1/4$, we perform $(x'', y'') = (4x + y, 4y - x)$, which then satisfies $y''/x'' \leq 1/4$. When either of these transformations is performed, we add the corresponding angle to the "angle-so-far" in register a.

We could continue this iteration 32 times to get (theoretically) the angle in degrees to $\pm 10^{-8}$. That seems a bit long, plus the accumulation of errors over 32 iterations could (in the worst case) produce less than $\pm 10^{-7}$ accuracy.

To get the accuracy we need, we work in "scaled reals". That is, we get 10 effective decimal places of accuracy by letting an x in the range $0 < x < 100$ stand for $0 < x/100 < 1$.

Our initial reductions can increase x by a factor of about 13. Moreover, we ultimately need to scale y by 100 when we convert to scaled computations. Thus, if we make sure x is less than 1 000 000, we will prevent overflow in both cases.

```
\def\MFP@Rquad{\MFP@Rdbl\MFP@Rcopyzx\MFP@Rdbl}%
\def\MFP@@@Rangle{%
  \MFP@Rcopy xs\MFP@Rcopy yt%
  \ifnum\MFP@x@Int<1000000
```

```
  \else
    \MFP@RdivC \MFP@Rcopyz s%
    \MFP@Rcopy tx\MFP@RdivC \MFP@Rcopyz t%
  \fi
  \ifnum\MFP@t@Sgn=0 \MFP@Rzero
  \else
    \MFP@Rcopy tx\MFP@Rdbl\MFP@Rcopyzx\MFP@Rcopy sy\MFP@Rcmp
    \ifMFP@pos
      \MFP@Rsub\MFP@Rcopyz u\MFP@Rcopy sx\MFP@Rdbl
      \MFP@Rcopyzx\MFP@Rcopy ty\MFP@Radd
      \MFP@Rcopyz s\MFP@Rcopy ut%
      \MFP@Rload a1{2656}{50511771}%
    \else
      \MFP@Rload a000%
    \fi
    \MFP@Rcopy tx\MFP@Rquad\MFP@Rcopyzx\MFP@Rcopy sy\MFP@Rcmp
    \ifMFP@pos
      \MFP@Rsub\MFP@Rcopyz u\MFP@Rcopy sx\MFP@Rquad
      \MFP@Rcopyzx\MFP@Rcopy ty\MFP@Radd
      \MFP@Rcopyz s\MFP@Rcopy ut%
      \MFP@Rcopy ax\MFP@Rload y1{1403}{62434679}%
      \MFP@Radd\MFP@Rcopy za%
    \fi
    \MFP@Rcopy tx\MFP@RmulC
    \MFP@Rcopyzx\MFP@Rcopy sy\MFP@Rdiv
    \MFP@Rcopyzx\MFP@Ratanc
    \MFP@Rcopyzx\MFP@Rdeg
    \MFP@Rcopyzx\MFP@Rcopy ay\MFP@Radd
    \MFP@Rcopyzx\MFP@RdivC
  \fi}%
```

Here are fast multiplication and division by 100. We need these because we are going to compute the arctangent in radians to ten decimal places. We do this by computing with scaled reals in which, for example, 0.5 is represented by 50.0. When we do this, multiplication requires a division by 100: $.5 \times .5 = .25$ would be computed as $(50 \times 50)/100 = 25$.

```
\def\MFP@twoofmany#1#2#3\MFP@end{#1#2}%
\def\MFP@gobbletwo#1#2{}%
\def\MFP@RmulC{%
  \edef\MFP@z@Int{\MFP@x@Int\@xp\MFP@twoofmany\MFP@x@Frc\MFP@end}%
  \edef\MFP@z@Frc{\@xp\MFP@gobbletwo\MFP@x@Frc00}%
  \edef\MFP@z@Sgn{\MFP@x@Sgn}}%
\def\MFP@RdivC{%
  \makeMFP@eightdigits\MFP@x@Int
  \makeMFP@eightdigits\MFP@x@Frc
  \@XP\MFP@@RdivC\@xp\MFP@x@Int\MFP@x@Frc\MFP@end}%
\def\MFP@@RdivC#1#2#3#4#5#6{%
  \edef\MFP@z@Int{\number#1#2#3#4#5#6}%
  \MFP@@@RdivC}%
\def\MFP@@@RdivC#1#2#3#4#5#6#7#8#9\MFP@end{%
```

```
\MFP@tempa#1#2#3#4#5#6#7#8\relax
\ifnum#9>49 \advance\MFP@tempa1 \fi
\edef\MFP@z@Frc{\number\MFP@tempa}%
\makeMFP@eightdigits\MFP@z@Frc
\edef\MFP@z@Sgn{\MFP@x@Sgn}%
\ifnum\MFP@tempa=0
  \ifnum\MFP@z@Int=0 \def\MFP@z@Sgn{0}\fi
\fi}%
```

Finally, we compute the arctan of a scaled real producing a result as a scaled number (i..e., as "centiradians"—100 times the number of radians) using a power series. Since that number could be around 0.25 (represented by 25.0), we have to sum to at least its 15th power ($4^{-15}/15 \approx .6 \times 10^{-10}$ and the next term in the series is effectively 0). Fortunately, the power series has only odd terms, so there are only eight terms we actually need to calculate. The calculation proceeds much like the one for the sine, starting with the sum

$$x\left(1 - \frac{u}{3} + \frac{u^2}{5} - \frac{u^3}{7} + \cdots - \frac{u^7}{15}\right),$$

where $u = x^2$.

We start with the common iterated code. It assumes a scaled value in x to be multiplied by the saved (scaled) value of x^2 (in register u) and by a coefficient (supplied in separate integer and fractional parts). It ends with the new value in x.

```
\def\MFP@scaledmul{\MFP@Rmul\MFP@Rcopyzx\MFP@RdivC}%
\def\MFP@atan@iter#1#2{%
  \MFP@Rcopy uy\MFP@scaledmul
  \MFP@Rcopyzx\MFP@Rload y1{#1}{#2}\MFP@scaledmul
  \MFP@Rcopyzy\MFP@Rload x1{100}{00000000}%
  \MFP@Rsub\MFP@Rcopyzx}%
\def\MFP@Ratanc{%
  \MFP@Rcopy xs\MFP@Rcopy xy\MFP@scaledmul
  \ifnum \MFP@z@Sgn=0
    \MFP@Rcopy sz%
  \else
    \MFP@Rcopyz u\MFP@Rcopyzx
    \MFP@Rload y1{86}{66666667}\MFP@scaledmul
    \MFP@Rcopyzy\MFP@Rload x1{100}{00000000}\MFP@Rsub\MFP@Rcopyzx
    \MFP@atan@iter{84}{61538462}\MFP@atan@iter{81}{81818182}%
    \MFP@atan@iter{77}{77777778}\MFP@atan@iter{71}{42857143}%
    \MFP@atan@iter{60}{00000000}\MFP@atan@iter{33}{33333333}%
    \MFP@Rcopy sy\MFP@scaledmul
  \fi}%
```

4.5 Logarithms

Now for logarithms. We are going to compute both common logarithms (base 10) and natural logarithms (base e). The first step of the calculation is be essentially

trivial and works with base 10: to get the integer part of the log for numbers with positive integer part, count the digits in the integer part and subtract 1. For numbers less than one, count the number of zeros at the beginning of the fractional part and add 1 (subtract this from the result of the second part). This reduces the problem to numbers $1 \le x < 10$. A few divisions (when necessary) reduce to the case where $x = 1 + u$ with u small enough that the power series for $\log(1 + u)$ can be computed accurately in an acceptable number of of terms. Then we proceed as in the code for sine.

The power series produces a logarithm in base e so we ultimately get the answer in two parts, with the parts calculated for different bases. The last step is to multiply the second part by a conversion factor and add the first to it. For natural log, convert the first and add the second. Which one is to be returned is passed as a boolean

We keep the value-so-far in register s and the modified x-value in register t.

```
\newif\ifMFP@natural
\def\MFP@Rlog{\MFP@naturalfalse\MFP@Rlog@}%
\def\MFP@Rln{\MFP@naturaltrue\MFP@Rlog@}%
\def\MFP@Rlog@{%
  \ifnum\MFP@x@Sgn=0
    \MFP@logofzero@err
    \MFP@Rloadz{-1}\LogOfZeroInt\LogOfZeroFrac
  \else
    \ifnum \MFP@x@Sgn<0
      \MFP@warn{The logarithm of a negative number is complex.
      \MFP@msgbreak Only the real part will be computed}%
      \def\MFP@x@Sgn{1}%
    \fi
    \MFP@Rload s000%
```

If the integer part is 0, the fractional part is not. Save the number of places that will be shifted in `\MFP@tempa`. We use `\number` to strip the leading zeros and (essentially) we count the number of digits that remain. Then we shift left, putting the first digit into the integer part of s and the rest into the fractional part.

```
    \ifnum \MFP@x@Int=0
      \edef\MFP@x@Tmp{\number\MFP@x@Frc}%
      \MFP@tempa=\MFP@numshiftL\MFP@x@Tmp\relax
      \def\MFP@s@Sgn{-1}%
      \edef\MFP@t@Int{\@xp\MFP@oneofmany\MFP@x@Tmp\MFP@end}%
      \edef\MFP@t@Frc{\@xp\@gobble\MFP@x@Tmp0}%
      \MFP@padtoeight\MFP@t@Frc
    \else
```

When the integer part is not 0, we get the number of digits to shift again in `\MFP@tempa`. It will be one less than the number of integer digits.

```
      \MFP@tempa\MFP@numshiftR\MFP@x@Int
      \edef\MFP@x@Tmp{\MFP@x@Int\MFP@x@Frc}%
      \ifnum\MFP@tempa>0 \def\MFP@s@Sgn{1}\fi
      \edef\MFP@t@Int{\@xp\MFP@oneofmany\MFP@x@Tmp\MFP@end}%
```

```
      \edef\MFP@x@Tmp{\@xp\@gobble\MFP@x@Tmp}%
      \edef\MFP@t@Frc{\@xp\MFP@eightofmany\MFP@x@Tmp\MFP@end}%
    \fi
```

Now the integer part of $\log_{10} x$ is known. We save it in s Also, set the sign of the reduced argument (positive). Then call `\MFP@Rlog@reduce`, which reduces x to less than 1.161 while possibly increasing s. For the natural log, we convert the value in s.

If the reduced x is 1, return the value in s, otherwise call the power series program (discarding the integer part of t, which should be a 1). Finally, convert the returned result if necessary and add register s to it.

```
    \edef\MFP@s@Int{\number\MFP@tempa}%
    \def\MFP@t@Sgn{1}%
    \MFP@Rlog@reduce
    \ifMFP@natural \MFP@Rcopy sx\MFP@RbaseE \MFP@Rcopy zs\fi
    \ifnum\MFP@t@Frc=0
      \MFP@Rcopy sz%
    \else
      \def\MFP@t@Int{0}\MFP@Rlog@prog
      \ifMFP@natural\else \MFP@Rcopyzx \MFP@RbaseX \fi
      \MFP@Rcopy sy\MFP@Rcopyzx\MFP@Radd
    \fi
  \fi}%
```

We determine the size of a right shift by lining up the digits in the integer part, followed by the possible numbers, and picking out the ninth argument. Similarly, to get a left shift we line up the digits of the fractional part (minus the leading zeros) followed by the possible numbers, and again picking the ninth.

```
\def\MFP@numshiftR#1{\@xp\MFP@ninthofmany#176543210\MFP@end}%
\def\MFP@numshiftL#1{\@xp\MFP@ninthofmany#112345678\MFP@end}%
```

In `\MFP@Rlog@reduce` we divide by the square root of 10 if the number is significantly larger than that (adding .5 to value-so-far). We repeat with the 4th, 8th and 16th roots. It seems that this could be where errors can accumulate, so the divisions are done with double precision multiplication and x is scaled by 100. Our check whether $x > \sqrt{10}$ is rather rough: comparing the first three digits only, but even in the worst case, the final x is less than 1.1605, so at most 0.161 is fed to the power series.

```
\def\MFP@Rlog@reduce{%
  \MFP@Rcopy tx\MFP@RmulC\MFP@Rcopyz t%
  \MFP@reduceonce {316}{31622776}{60168379}{50000000}%
  \MFP@reduceonce {177}{56234132}{51903491}{25000000}%
  \MFP@reduceonce {133}{74989420}{93324558}{12500000}%
  \MFP@reduceonce {115}{86596432}{33600654}{06250000}%
  \MFP@Rcopy tx\MFP@RdivC\MFP@Rcopyz t}%
\def\MFP@reduceonce#1#2#3#4{%
  \ifnum\MFP@t@Int>#1\relax
    \MFP@Rcopy tx%
    \MFP@DPmul 0{#2}{#3}\MFP@Rcopyz t%
    \MFP@Rcopy sx\MFP@Rload y10{#4}\MFP@Radd
```

```
    \MFP@Rcopyz s%
  \fi}%
```

Now we have a value for t of the form $1+u$ with $0 \le u < 0.161$. We will use the formula

$$\ln(1+u) = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{u^n}{n}.$$

We only need to carry it far enough to assure that the remaining terms would be zero in our finite resolution arithmetic, that is $(.161)^k/k < .5 \times 10^{-8}$. This is satisfied by $k = 10$. So we carry the sum to 9 places.

Again, we compute this by

$$u(1 - au(1 - bu(1 - cu(1 - du(1 - eu(1 - fu(1 - gu(1 - hu)))))))))$$

where $a = 1/2$, $b = 2/3$,..., $g = 7/8$, and $h = 8/9$ This arrangement allows us to reuse `\MFP@com@iter`.

```
\def\MFP@Rlog@prog{%
  \MFP@Rcopy tx\MFP@Rload y10{88888889}\MFP@Rsmul
  \MFP@com@iter{87500000}\MFP@com@iter{85714286}\MFP@com@iter{83333333}%
  \MFP@com@iter{80000000}\MFP@com@iter{75000000}\MFP@com@iter{66666667}%
  \MFP@com@iter{50000000}\MFP@flipz\MFP@Rcopyzx\MFP@Rcopy ty\MFP@Rsmul}%
```

4.6 Powers

With the exponential function we immediately return 1 if $x = 0$. We call two separate handlers for positive and negative x. This is because the issues are different between positive and negative exponents.

```
\def\MFP@Rexp{%
  \ifcase\MFP@x@Sgn\relax
    \MFP@Rloadz 110%
  \or
    \MFP@Rexp@pos
  \else
    \def\MFP@x@Sgn{1}%
    \MFP@Rexp@neg
  \fi}%
```

One issue for positive exponents is overflow, so we issue an error message for that case. The largest mumber that will not produce overflow is 18.42068074 so we first compare to that; if larger, issue the error message and return 99999999.99999999.

We compute the integer power first, using an `\ifcase`. Because there are only 19 cases to consider a table lookup is faster than multiplications.

Then, we examine the first digit d after the decimal and compute $e^{0.d}$, again by cases. This is multiplied by the integer power previously found. What remains is the rest of the fractional part of x, which is strictly less than 0.1. The exponential of this is computed using the first several terms of the power series for e^x.

```
\def\MFP@Rexp@pos{%
```

```
\MFP@Rload y1{18}{42068074}\MFP@Rcmp
\ifMFP@pos
  \MFP@expoverflow@err
  \MFP@Rloadz 1\MaxRealInt\MaxRealFrac
\else
  \MFP@tempa\MFP@x@Int
  \edef\MFP@powerof@e{%
   1\ifcase\MFP@tempa
    10\or
    2{71828183}\or
    7{38905610}\or
    {20}{08553692}\or
    {54}{59815003}\or
    {148}{41315910}\or
    {403}{42879349}\or
    {1096}{63315843}\or
    {2980}{95798704}\or
    {8103}{08392758}\or
    {22026}{46579481}\or
    {59874}{14171520}\or
    {162754}{79141900}\or
    {442413}{39200892}\or
    {1202604}{28416478}\or
    {3269017}{37247211}\or
    {8886110}{52050787}\or
    {24154952}{75357530}\or
    {65659969}{13733051}\else
    {\MaxRealInt}{\MaxRealFrac}\fi}%
  \@xp\MFP@Rloadz\MFP@powerof@e
  \ifnum\MFP@x@Frc=0
  \else
    \MFP@Rcopyz s%
    \MFP@tempa=\@xp\MFP@oneofmany\MFP@x@Frc\MFP@end
    \edef\MFP@powerof@e{%
    y1\ifcase\MFP@tempa
      10\or
      1{10517092}\or
      1{22140276}\or
      1{34985881}\or
      1{49182470}\or
      1{64872127}\or
      1{82211880}\or
      2{01375271}\or
      2{22554093}\or
      2{45960311}\else
      10\fi}%
    \edef\MFP@t@Frc{0\@xp\@gobble\MFP@x@Frc}%
    \MFP@Rcopy sx\@xp\MFP@Rload\MFP@powerof@e\MFP@Rmul
    \ifnum\MFP@t@Frc=0
    \else
```

```
        \MFP@Rcopyz s\MFP@Rload t10\MFP@t@Frc
        \MFP@Rexp@pos@prog
        \MFP@Rcopy sx\MFP@Rcopyzy\MFP@Rmul
      \fi
    \fi
  \fi}%
```

Since the x value is now less than 0.1, we can get eight places of accuracy with only six terms of the power series. We can also arrange to use the more efficient `\MFP@Rsmul` for multiplication.

We organize the computation thusly

$$1 + (x + x/2(x + x/3(x + x/4(x + x/5(x + x/6)))))$$

We start by loading x (now in register t) into register z, then repeatedly run `\MFP@Rexp@iter` feeding it the successive values of $1/n$. This iterator first multiplies the most recent result (the z register) by $1/n$, then that by x and then adds x to that. The final step is to add 1.

```
\def\MFP@Rexp@pos@prog{%
  \MFP@Rcopy tz\MFP@Rexp@iter{14285714}\MFP@Rexp@iter{16666667}%
  \MFP@Rexp@iter{20000000}\MFP@Rexp@iter{25000000}%
  \MFP@Rexp@iter{33333333}\MFP@Rexp@iter{50000000}\MFP@Rcopyzx
  \MFP@Rincr}%
\def\MFP@Rexp@iter#1{%
  \MFP@Rcopyzx\MFP@Rload y10{#1}\MFP@Rsmul
  \MFP@Rcopyzx\MFP@Rcopy ty\MFP@Rsmul
  \MFP@Rcopyzx\MFP@Rcopy ty\MFP@Radd}%
```

It is impossible to get accuracy to the last digit when e^x is large. This is because an absolute error in x converts to a relative error in e^x, That is, knowing x only to 10^{-8} means e^x is off by (about) $e^x \cdot 10^{-8}$. Roughly speaking, this means only about 8 places of e^x are accurate, so if (for example) the integer part of e^x has six places then only two places after the decimal are significant.

The first issue with negative exponents is that it doesn't take much to produce a value of e^{-x} that rounds to 0. Any $x > 19.11382792$. So we start by comparing to that value and simply return zero if x is larger.

We perform exactly the same reductions as for positive exponents, handling the integer part and the first decimal separately. Then we call the power series program (not the same).

```
\def\MFP@Rexp@neg{%
  \MFP@Rload y1{19}{11382792}%
  \MFP@Rcmp
  \ifMFP@pos
    \MFP@Rloadz 000%
  \else
    \MFP@tempa\MFP@x@Int
    \edef\MFP@powerof@e{%
      \ifcase\MFP@tempa
```

```
    11{0}\or
    10{36787944}\or
    10{13533528}\or
    10{04978707}\or
    10{01831564}\or
    10{00673795}\or
    10{00247875}\or
    10{00091188}\or
    10{00033546}\or
    10{00012341}\or
    10{00004540}\or
    10{00001670}\or
    10{00000614}\or
    10{00000226}\or
    10{00000083}\or
    10{00000031}\or
    10{00000011}\or
    10{00000004}\or
    10{00000002}\or
    10{00000001}\else
    000\fi}%
  \@xp\MFP@Rloadz\MFP@powerof@e
  \ifnum\MFP@x@Frc=0
  \else
    \MFP@Rcopyz s%
    \MFP@tempa=\@xp\MFP@oneofmany\MFP@x@Frc\MFP@end
    \edef\MFP@powerof@e{%
    y1\ifcase\MFP@tempa
      10\or
      0{90483742}\or
      0{81873075}\or
      0{74081822}\or
      0{67032005}\or
      0{60653066}\or
      0{54881164}\or
      0{49658530}\or
      0{44932896}\or
      0{40656966}\else
      10\fi}%
    \edef\MFP@t@Frc{0\@xp\@gobble\MFP@x@Frc}%
    \MFP@Rcopy sx\@xp\MFP@Rload\MFP@powerof@e\MFP@Rmul
    \ifnum\MFP@t@Frc=0
    \else
      \MFP@Rcopyz s\MFP@Rload t10\MFP@t@Frc
      \MFP@Rexp@neg@prog
      \MFP@Rcopyzx\MFP@Rcopy sy\MFP@Rmul
    \fi
  \fi
\fi}%
```

Since x is now positive we calculate e^{-x}. Again we need only up to the 6th power, organized as follows

$$1 - x(1 - x/2(1 - x/3(1 - x/4(1 - x/5(1 - x/6)))))$$

Since this has exactly the same form as the the power series calculation for log and sin, we can reuse the code in `\MFP@com@iter`. We end with the final multiplication by x and the subtraction from 1 rather than call `\MFP@com@iter` with a useless multiplication by 1.

```
\def\MFP@Rexp@neg@prog{%
  \MFP@Rcopy tx\MFP@Rload y10{14285712}\MFP@Rsmul
  \MFP@com@iter{16666667}\MFP@com@iter{20000000}%
  \MFP@com@iter{25000000}\MFP@com@iter{33333333}%
  \MFP@com@iter{50000000}\MFP@flipz\MFP@Rcopyzx
  \MFP@Rcopy ty\MFP@Rsmul\MFP@flipz}%
```

The most efficient way to take an integer power of a number x is to scan the binary code for the exponent. Each digit 1 in this code corresponds to a 2^k power of x, which can be computed by repeatedly squaring x. These *dyadic* powers are mutiplied together. We can convert this idea to a simple loop illustrated by this example of finding x^{13} ($13 = 1101$ in base 2). Here p holds the current product and q holds the current dyadic power of x, initialized with $p = 1$ and $q = x$:

1. Rightmost digit 1: update $p \leftarrow pq = x$ and $q \leftarrow q^2 = x^2$.
2. Next digit 0: Just update $q \leftarrow q^2 = x^4$.
3. Next digit 1: update $p \leftarrow pq = x^5$ and $q \leftarrow q^2 = x^8$.
4. Next digit 1: update $p \leftarrow pq = x^{13}$, detect that we are at the end and skip the update of q. Return p.

Of course, this requires the binary digits of the exponent n. But the rightmost digit of n is 1 if and only if n is odd, and we can examine each digit in turn if we divide n by 2 (discarding the remainder) at each stage. We detect the end when n is reduced to 1.

Accuracy is partly a function of the number of multiplications. The above scheme requires at most $\lfloor \log_2 n \rfloor$ squarings and at most $\lceil \log_2 n \rceil$ multiplications for x^n, while directly multiplying $x \cdot x \cdots x$ would require $n - 1$ multiplications. I have tested with an exponent equal to 8000 and it takes only about 25 times as long as a single multiplication (rather than 7999 times).

For negative powers we can either find the positive power of x and take its reciprocal or take the reciprocal of x and find its positive power. We do the second so that overflow can be detected in `\MFP@@Rpow`.

```
\def\MFP@Rpow{%
  \ifnum\MFP@y@Frc>0
    \MFP@warn{The "pow" function requires an integer power.
      \MFP@msgbreak The fractional part will be ignored}%
  \fi
  \MFP@loopctr=\MFP@y@Int\relax
  \ifnum\MFP@loopctr=0
    \MFP@Rloadz 110%
```

```
  \else
    \ifnum\MFP@x@Sgn=0
      \ifnum\MFP@y@Sgn>0
        \MFP@Rloadz 000%
      \else
        \MFP@badpower@err
        \MFP@Rloadz 1\xOverZeroInt\xOverZeroFrac
      \fi
    \else
      \ifnum\MFP@x@Sgn>0
        \def\MFP@power@Sgn{1}%
      \else
        \edef\MFP@power@Sgn{\ifodd\MFP@loopctr -\fi 1}%
      \fi
      \ifnum\MFP@y@Sgn<0 \MFP@Rinv \MFP@Rcopyzx\fi
      \ifnum\MFP@loopctr=1
        \MFP@Rloadz \MFP@power@Sgn\MFP@x@Int\MFP@x@Frc
      \else
        \MFP@@Rpow
      \fi
    \fi
  \fi}%
```

This implements the algorithm discussed above. We save x in register q, initialize the starting value of 1 in p and then run the loop. If the binary digit just read is a 1 (i.e., `\ifodd` is true), it multiplies p and q. It also saves the last product (copies z to p). This need not be done on the last iteration, but must not be moved out of the `\ifodd` conditional because intervening computations modify z. If there are more iterations to do (i.e., the `\ifnum` is true), this squares q and reduces the counter. Note that the exponents 0 and 1 do not occur since we have handled them separately.

In case of overflow (either the multiplication or the squaring) we break the loop and return $\pm\infty$.

```
\def\MFP@@Rpow{%
  \MFP@Rcopy xq%
  \MFP@Rload p110%
  \MFP@Rpow@loop}%
\def\MFP@Rpow@loop{%
  \ifodd\MFP@loopctr
    \MFP@Rcopy px\MFP@Rcopy qy\MFP@Rmul
    \ifnum \MFP@z@Ovr>0 \MFP@handle@expoverflow
    \else
      \ifnum\MFP@loopctr>1 \MFP@Rcopyz p\fi
    \fi
  \fi
  \ifnum\MFP@loopctr>1
    \MFP@Rcopy qx\MFP@Rsq
    \ifnum \MFP@z@Ovr>0 \MFP@handle@expoverflow
    \else
```

```
      \MFP@Rcopyz q%
      \divide\MFP@loopctr 2
      \@XP\MFP@Rpow@loop
    \fi
  \fi}%
\def\MFP@handle@expoverflow{%
  \MFP@expoverflow@err
  \MFP@loopctr=0
  \MFP@Rloadz\MFP@power@Sgn\MaxRealInt\MaxRealFrac}%
```

4.7 The square root

One can combine logarithms and exponentials to get any power: to get x^y, compute $e^{y \ln x}$. This has the disadvantage that it doesn't work if x is negative. Most powers of negative numbers are not defined, but certainly integer powers are. Thus we have defined `\MFPpow` and `\Rpow` for that case.

If we enforce a positive x, then y can have any value. However, the computation of $e^{.5 \ln x}$ cannot give a result as good as one can get from a special purpose algorithm for the square root. For example, the inaccuracies in computing $\ln x$ will make $e^{.5 \ln 9}$ inexact, while the square root function we implement below will produce exactly $\sqrt{9} = 3$. In fact, if a square root can be expressed exactly within our 8-digit precision, our code will find it.

For the square root we return zero if x is not positive. If the integer part of x is 0, we copy the fractional part to the integer part (that is, we multiply by 10^8, remembering to multiply by 10^{-4} later). This makes the square root of such numbers rather more accurate. (To get around some other rare but annoying inaccuracies, we go through a similar process when the integer part of x is at most 4 digits, multiplying by 10^4 before and by 10^{-2} after.)

We then compute the square root using an algorithm that will be exact whenever possible. We perform one additional processing step. To explain it, note that our algorithm actually produces the largest number s with four digits right of the decimal place that satisfies $s^2 \le x$. That is

$$s^2 \le x < \left(s + 10^{-4}\right)^2$$

From this it follows that $x = (s + \epsilon)^2 = s^2 + 2s\epsilon + \epsilon^2$ with $\epsilon < 10^{-4}$ (and so $\epsilon^2 < 10^{-8}$). We estimate this ϵ and add that estimate to s. The estimate we use is obtained by discarding the very small ϵ^2 and solving for the remaining ϵ get

$$\epsilon \approx \bar{\epsilon} = \frac{x - s^2}{2s}$$

With this value, $s + \bar{\epsilon}$ misses the exact square root by at most $\epsilon^2/(2s) < .5 \cdot 10^{-8}$, because $s \ge 1$. The final result $s + \bar{\epsilon}$ is equivalent to computing the average s and x/s. This, possibly divided by 10^4 or 10^2 is the returned value.

By tests, with rare exceptions, our computations produces a result correct in all eight decimal places. In the rare exception, the last place is within 1 of the correct value.

```
\def\MFP@Rsqrt{%
  \ifcase\MFP@x@Sgn\relax
    \MFP@Rzero
  \or
    \ifnum\MFP@x@Int=0
      \def\MFP@sqrt@reduce{2}%
      \edef\MFP@x@Int{\number\MFP@x@Frc}%
      \edef\MFP@x@Frc{00000000}%
    \else\ifnum\MFP@x@Int<10000
      \def\MFP@sqrt@reduce{1}%
      \edef\MFP@x@Int{\MFP@x@Int\@xp\MFP@fourofmany\MFP@x@Frc\MFP@end}%
      \edef\MFP@x@Frc{\@xp\MFP@gobblefour\MFP@x@Frc0000}%
    \else
      \def\MFP@sqrt@reduce{0}%
    \fi\fi
    \MFP@Rcopy xt%
    \MFP@Isqrt
    \MFP@Rcopyz s\MFP@Rcopyzy
    \MFP@Rcopy tx\MFP@Rdiv
    \MFP@Rcopy sx\MFP@Rcopyzy\MFP@Radd
    \MFP@Rcopyzx\MFP@Rhalve
    \ifcase \MFP@sqrt@reduce\relax
    \or
      \MFP@Rcopyzx\MFP@Rload y10{01000000}\MFP@Rmul
    \or
      \MFP@Rcopyzx\MFP@Rload y10{00010000}\MFP@Rmul
    \fi
  \else
    \MFP@warn{Square root of a negative number. Zero will be returned.}%
    \MFP@Rzero
  \fi}%
\def\MFP@fourofmany#1#2#3#4#5\MFP@end{#1#2#3#4}%
\def\MFP@gobblefour#1#2#3#4{}%
```

There is a rather straightforward pencil and paper algorithm that provides the square root digit by digit, and it produces an exact answer when that is possible. Unfortunately, the decimal version is not easy to code. Fortunately the same algorithm works in any number base and it is rather simple to code the binary version (because we only need to decide at each stage whether the "next digit" is 0 or 1. This produces a square root in binary digits, from which it is easy to compute the number itself. The result is exact if the answer would be a finite number of binary digits. We apply it to the integer 10^8x. While this number is too large for TEX to handle as an integer, it is not that hard to convert it to a string of binary digits stored in a macro.

The process turns out to be simpler if we convert 10^8x to base 4 rather than binary. Also, instead of producing the square root encoded in a string of binary digits, we simply build the numerical result as we discover the binary digits (multiply previous value by two and add the new digit.) Fortunately, the square root of 10^8x (and the temporary scratch registers used in the code) will never exceed

TEX's limit for integers.

The macro `\MFP@ItoQ` implements the conversion to base-4 digits. The two arguments are the integer and fractional part of x. The result is stored in `\MFP@ItoQ@Tmp`, which is so far only used by the square root code.

The test `\ifodd\MFP@tempb` is used to get the binary digits. Combining two of them yields the quadrenary digits. The `\ifodd\MFP@tempa` tests are there to check whether there will be a remainder after division by 2, which should then be inserted at the front of `\MFP@tempb` before division by 2. Two divisions by 2 each iteration amounts to division by 4. This is slightly more efficient than dividing by 4 and determining the remainder.

```
\def\MFP@ItoQ#1#2{%
  \MFP@tempa#1\relax\MFP@tempb#2\relax
  \def\MFP@ItoQ@Tmp{}\MFP@ItoQ@loop}%
\def\MFP@ItoQ@loop{%
  \ifodd\MFP@tempb
    \ifodd\MFP@tempa \advance\MFP@tempb \MFP@ttteight\relax\fi
    \divide\MFP@tempa2 \divide\MFP@tempb2
    \edef\MFP@ItoQ@Tmp{\ifodd\MFP@tempb 3\else 1\fi\MFP@ItoQ@Tmp}%
  \else
    \ifodd\MFP@tempa \advance\MFP@tempb \MFP@ttteight\relax\fi
    \divide\MFP@tempa2 \divide\MFP@tempb2
    \edef\MFP@ItoQ@Tmp{\ifodd\MFP@tempb 2\else 0\fi\MFP@ItoQ@Tmp}%
  \fi
  \ifodd\MFP@tempa \advance\MFP@tempb \MFP@ttteight\relax\fi
  \divide\MFP@tempa 2 \divide\MFP@tempb 2
  \ifnum\MFP@tempa>0
    \@xp\MFP@ItoQ@loop
  \else\ifnum\MFP@tempb>0
    \@XP\MFP@ItoQ@loop
  \fi\fi}%
```

This integer square root n is 10^4 times the largest number y satisfying $y^2 \leq x$ and having at most four decimal places. The rest of the code after the `\MFP@Isqrt@loop` is intended to divide n (returned in `\MFP@tempc`) by 10^4 in order to get the number y itself.

```
\def\MFP@Isqrt{%
  \MFP@ItoQ\MFP@x@Int\MFP@x@Frc
  \MFP@tempa=0 \MFP@tempb=0 \MFP@tempc=0
  \expandafter\MFP@Isqrt@loop\MFP@ItoQ@Tmp\MFP@end
  \MFP@tempa=\MFP@tempc
  \divide\MFP@tempc\MFP@tttfour
  \edef\MFP@z@Int{\number\MFP@tempc}%
  \multiply\MFP@tempc \MFP@tttfour
  \advance\MFP@tempa -\MFP@tempc
  \edef\MFP@z@Frc{\number\MFP@tempa}%
  \makeMFP@fourdigits\MFP@z@Frc
  \edef\MFP@z@Frc{\MFP@z@Frc0000}%
  \def\MFP@z@Sgn{1}}%
```

The following is a loop that essentially performs a base-2 version of the base-10

algorithm that I learned at age 12 from my father (apparently it was taught in eighth or ninth grade in his day). Seeing it written out, I am surprise at how concise and elegant it is!

```
\def\MFP@Isqrt@loop#1{%
  \ifx\MFP@end #1%
  \else
    \multiply\MFP@tempa 2 \multiply\MFP@tempb 4 \multiply\MFP@tempc 2
    \advance \MFP@tempb#1\relax
    \ifnum\MFP@tempa<\MFP@tempb
       \advance\MFP@tempc 1 \advance\MFP@tempa 1
       \advance\MFP@tempb -\MFP@tempa
       \advance\MFP@tempa 1
    \fi
    \expandafter\MFP@Isqrt@loop
  \fi}%
\MFP@xfinish
⟨/extra⟩
```

For my own benefit: the above code finds the next binary digit and updates the square root (in `\MFP@tempc`) by appending that digit. The new digit is also appended to the end of `\MFP@tempa`. This is subtracted from `\MFP@tempb`, but only if the last digit is a 1. Then the next quadrenary digit is appended to `\MFP@tempb`. Finally, the last binary digit found is added (not appended) to `\MFP@tempa`. The "appending" of a digit means a multiplication by 2 (or 4) and the addition of the digit. We perform such additions only if the digit is a 1, and we determine if the digit is 1 or 0 by the `\ifnum` test.

Index

Numbers refer to the page(s) where the corresponding entry is described.

R

S

X

Z