The xint bundle

Jean-François Burnol
jfbu (at) free (dot) fr

Package version: 1.09kb (2014/02/13)


```
\input xintexpr.sty
% December 7, 2013. Expandably computing a big Fibonacci number
% using TeX+\numexpr+\xintexpr, (c) Jean-François Burnol
% January 17, 2014: algorithm modified to be more economical in computations.

\catcode`_ 11

\def\Fibonacci #1{%
    \expandafter\Fibonacci_a\expandafter
        {\the\numexpr #1\expandafter}\expandafter
        {\romannumeral0\xintiieval 1\expandafter\relax\expandafter}\expandafter
        {\romannumeral0\xintiieval 1\expandafter\relax\expandafter}\expandafter
        {\romannumeral0\xintiieval 1\expandafter\relax\expandafter}\expandafter
        {\romannumeral0\xintiieval 0\relax}}

\def\Fibonacci_a #1{%
    \ifcase #1
          \expandafter\Fibonacci_end_i
    \or
          \expandafter\Fibonacci_end_ii
    \else
          \ifodd #1
              \expandafter\expandafter\expandafter\Fibonacci_b_ii
          \else
              \expandafter\expandafter\expandafter\Fibonacci_b_i
          \fi
    \fi {#1}%
}%

\def\Fibonacci_b_i #1#2#3{\expandafter\Fibonacci_a\expandafter
  {\the\numexpr #1/2\expandafter}\expandafter
  {\romannumeral0\xintiieval sqr(#2)+sqr(#3)\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiieval (2*#2-#3)*#3\relax}%
}% end of Fibonacci_b_i

\def\Fibonacci_b_ii #1#2#3#4#5{\expandafter\Fibonacci_a\expandafter
  {\the\numexpr (#1-1)/2\expandafter}\expandafter
  {\romannumeral0\xintiieval sqr(#2)+sqr(#3)\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiieval (2*#2-#3)*#3\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiieval #2*#4+#3*#5\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiieval #2*#5+#3*(#4-#5)\relax}%
}% end of Fibonacci_b_ii

\def\Fibonacci_end_i  #1#2#3#4#5{\xintthe#5}
\def\Fibonacci_end_ii #1#2#3#4#5{\xinttheiiexpr #2*#5+#3*(#4-#5)\relax}

\catcode`_ 8

% This \Fibonacci macro is designed to compute *one* Fibonacci number, not a
% whole sequence of them. Let's reap the fruits of our work:

\message{F(1250)=\Fibonacci {1250}}
\bye % see subsection 23.22 for some explanations and more.
```

Documentation generated from the source file with timestamp "13-02-2014 at 22:56:36 CET".

Description of the packages

xinttools is loaded by **xint** (hence by all other packages of the bundle, too): it provides utilities of independent interest such as expandable and non-expandable loops.

xint implements with expandable TeX macros additions, subtractions, multiplications, divisions and powers with arbitrarily long numbers.

xintfrac extends the scope of **xint** to decimal numbers, to numbers in scientific notation and also to fractions with arbitrarily long such numerators and denominators separated by a forward slash.

xintexpr extends **xintfrac** with an expandable parser `\xintexpr . . . \relax` of expressions involving arithmetic operations in infix notation on decimal numbers, fractions, numbers in scientific notation, with parentheses, factorial symbol, function names, comparison operators, logic operators, twofold and threefold way conditionals, sub-expressions, macros expanding to the previous items.

Further modules:

xintbinhex is for conversions to and from binary and hexadecimal bases.

xintseries provides some basic functionality for computing in an expandable manner partial sums of series and power series with fractional coefficients.

xintgcd implements the Euclidean algorithm and its typesetting.

xintcfrac deals with the computation of continued fractions.

Most macros, and all of those doing computations, work purely by expansion without assignments, and may thus be used almost everywhere in TeX.

The packages may be used with any flavor of TeX supporting the ε-TeX extensions. LaTeX users will use `\usepackage` and others `\input` to load the package components.

Contents

1 Read me first

This section provides recommended reading on first discovering the package; complete details are given later in the manual.

1.1 Presentation of the package

The components of the **xint** bundle provide macros dedicated to *expandable* computations on numbers exceeding the TeX (and ε-TeX) limit of `2147483647`.

The ε-TeX extensions must be enabled; this is the case in modern distributions by default, except if TeX is invoked under the name `tex` in command line (`etex` should be used then, or `pdftex` in `DVI` output mode). All components may be used as regular LaTeX packages or, with any other format based on TeX, loaded directly via \input (e.g. `\input xint.sty\relax`). Each package automatically loads those not already loaded it depends on.

The **xint** bundle consists of the three principal components **xint**, **xintfrac** (which loads **xint**), and **xintexpr** (which loads **xintfrac**), and four additional modules. The macros of the **xint** bundle not dealing directly with the manipulation of big numbers belong to a package **xinttools** (automatically loaded by all others), which is of independent interest.

1.2 User interface

The user interface for executing operations on numbers is via macros such as `\xintAdd` or `\xintMul` which have two arguments, or via expressions `\xintexpr..\relax` which use infix notations such as `+`, `-`, `*`, `/`, and `^` for the basic operations, and recognize functions of one or more comma separated arguments (such as `max`, or `round`, or `sqrt`), parentheses, logic operators of conjunction `&`, disjunction `|`, as well as two-way `?` and three-way `:` conditionals and more.

In the latter case the contents are expanded completely from left to right until the ending `\relax` is found and swallowed, and spaces and even (to some extent) catcodes do not matter. In the former (macro) case the arguments are each subjected to the process of *f*-expansion: repeated expansion of the first token until finding something unexpandable (or being stopped by a space token).

Conversely this process of *f*-expansion always provokes the complete expansion of the package macros and `\xintexpr..\relax` also will expand completely under *f*-expansion, but to a private format; the `\xintthe` prefix allows the computation result either to be passed as argument to one of the package macros,[1] or also end up on the printed page (or in an auxiliary file).

To recapitulate: all macros dealing with computations (1.) *expand completely under the sole process of repeated expansion of the first token, (and two expansions suffice)*,[2] (2.) *apply this f-expansion to each one of their arguments.* Hence they can be nested one within the other up to arbitrary depths. Conditional evaluations either within the macro arguments themselves, or with branches defined in terms of these macros are made possible via macros such as as `\xintifSgn` or `\xintifCmp`.

There is no notion of *declaration of a variable* to **xint**, **xintfrac**, or **xintexpr**. The user employs the `\def`, `\edef`, or `\newcommand` (in LaTeX) as usual, for example:

```
\def\x{17} \def\y{35} \edef\z{\xintMul {\x}{\y}}
```

As a faster alternative to `\edef` (when hundreds of digits are involved), the package provides `\oodef` which only expands twice its argument.

The **xintexpr** package has a private internal representation for the evaluated computation result. With

```
\oodef\z {\xintexpr 3.141^17\relax}
```

the macro `\z` is already fully evaluated (two expansions were applied, and this is enough), and can be reused in other `\xintexpr`-essions, such as for example

```
\xintexpr \z+1/\z\relax
```

But to print it, or to use it as argument to one of the package macros, it must be prefixed by `\xintthe` (a synonym for `\xintthe\xintexpr` is `\xinttheexpr`). Application of this `\xintthe` prefix outputs the value in the **xintfrac** semi-private internal format `A/B[N]`,[3]

[1] the `\xintthe` prefix *f*-expands the `\xintexpr`-ession then unlocks it from its private format; it should not be used for sub-expressions inside a bigger one as its is more efficient for the expression parser to keep the result in the private format. [2] see in section 7 for more details. [3] there is also the notion of `\xintfloatexpr`, for which the output format after the action of `\xintthe` is a number in floating point scientific notation.

representing the fraction $(A/B)\times 10^N$. The example above produces a somewhat large output: 79489513238562599144638186610178199027369015027956794746351751450959463226307850357841486232421619170499633662459598361/281938846629127356485024692078690813942961005925167846878981[-51]

> By default, computations done by the macros of **xintfrac** or within `\xintexpr..\relax` are exact. Inputs containing decimal points or scientific parts do not make the package switch to a 'floating-point' mode. The inputs, however long, are converted into exact internal representations.

The `A/B[N]` shape is the output format of most **xintfrac** macros, it benefits from accelerated parsing when used on input, compared to the normal user syntax which has no `[N]` part. An example of valid user input for a fraction is

`-123.45602e78/+765.987e-123`

where both the decimal parts, the scientific exponent parts, and the whole denominator are optional components. The corresponding semi-private form in this case would be

`-12345602/765987[199]`

The optional forward slash / introducing a denominator is not an operation, but a denomination for a fractional input. Reduction to the irreducible form must be asked for explicitely via the `\xintIrr` macro or the `reduce` function within `\xintexpr..\relax`. Elementary operations on fractions work blindly (addition does not even check for equality of the denominators and multiply them automatically) and do none of the simplifications which could be obvious to (some) human beings.

1.3 Space and time, floating point macros

The size of the manipulated numbers is limited by two factors:[4] (1.) *the available memory as configured in the* `tex` *executable*, (2.) *the* time *necessary to fully expand the computations themselves*. The most limiting factor is the second one, the time needed (for multiplication and division, and even more for powers) explodes with increasing input sizes long before the computations could get limited by constraints on TeX's available memory: computations with 100 digits are still reasonably fast, but the situation then deteriorates swiftly, as it takes of the order of seconds (on my laptop) for the package to multiply exactly two numbers each of 1000 digits and it would take hours for numbers each of 20000 digits.[5]

To address this issue, floating point macros are provided to work with a given arbitrary precision. The default size for significands is 16 digits. Working with significands of 24, 32, 48, 64, or even 80 digits is well within the reach of the package. But routine multiplications and divisions will become too slow if the precision goes into the hundreds, although the syntax to set it (`\xintDigits:=P;`) allows values up to 32767.[6] The exponents may

[4] there is an intrinsic limit of 2147483647 on the number of digits, but it is irrelevant, in view of the other limiting factors. [5] Perhaps some faster routines could emerge from an approach which, while maintaining expandability would renounce at *f*-expandability (without impacting the input save stack). There is one such routine `\xintXTrunc` which is able to write to a file (or inside an `\edef`) tens of thousands of digits of a (reasonably-sized) fraction. [6] for a one-shot conversion of a fraction to float format, or one addition, a precision exceeding 32767 may be passed as optional argument to the used macro.

be as big as ±2147483647.[7]

Here is such a floating point computation:

```
\xintFloatPower [48] {1.1547}{\xintiiPow {2}{35}}
```

which thus computes $(1.1547)^{2^{35}} = (1.1547)^{34359738368}$ to be approximately

$2.785{,}837{,}382{,}571{,}371{,}438{,}495{,}789{,}880{,}733{,}698{,}213{,}205{,}183{,}990{,}48 \times 10^{2{,}146{,}424{,}193}$

Notice that `2^35` exceeds TEX's bound, but `\xintFloatPower` allows it, what counts is the exponent of the result which, while dangerously close to `2^31` is not quite there yet. The printing of the result was done via the `\numprint` command from the numprint package[8].

The same computation can be done via the non-expandable assignment `\xintDigits:=48;` and then

```
\xintthefloatexpr 1.1547^(2^35)\relax
```

Notice though that `2^35` will be evaluated as a floating point number, and if the floating point precision had been too low, this computation would have given an inexact value. It is safer, and also more efficient to code this as:

```
\xintthefloatexpr 1.1547^\xintiiexpr 2^35\relax\relax
```

The `\xintiiexpr` is a cousin of `\xintexpr` which is big integer-only and skips the overhead of fraction management. Notice on this example that being embedded inside the `floatexpr`-ession has nil influence on the `iiexpr`-ession: expansion proceeds in exactly the same way as if it had been at the 'top' level.

xintexpr provides *no* implementation of the `IEEE` standard: no `NaN`s, signed infinities, signed zeroes, error traps, ...; what is achieved though is exact rounding for the basic operations. The only non-algebraic operation currently implemented is square root extraction. The power functions (there are three of them: `\xintPow` to which `^` is mapped in `\xintexpr..\relax`, `\xintFloatPower` for `^` in `\xintfloatexpr..relax`, and `\xintFloatPow` which is slighty faster but limits the exponent to the TEX bound) allow only integral exponents.

1.4 Printing big numbers on the page

When producing very long numbers there is the question of printing them on the page, without going beyond the page limits. In this document, I have most of the time made use of these macros (not provided by the package:)

```
\def\allowsplits #1{\ifx #1\relax \else #1\hskip 0pt plus 1pt\relax
                    \expandafter\allowsplits\fi}%
\def\printnumber #1{\expandafter\allowsplits \romannumeral-`0#1\relax }%
% \printnumber thus first ``fully'' expands its argument.
```

An alternative (footnote 13) is to suitably configure the thousand separator with the numprint package (does not work in math mode; I also tried siunitx but even in text mode could not get it to break numbers accross lines). Recently I became aware of the seqsplit package[9] which can be used to achieve this splitting accross lines, and does work in inline math mode.

[7] almost... as inner manipulations may either add or subtract the precision value to the exponent, arithmetic overflow may occur if the exponents are a bit to close to the TEX bound ±2147483647.
[8] http://ctan.org/pkg/numprint [9] http://ctan.org/pkg/seqsplit

1.5 Expandable implementations of mathematical algorithms

Another use of the `\xintexpr`-essions is illustrated with the algorithm on the title page: it shows how one may chain expandable evaluations, almost as if one were using the `\numexpr` facilities.[10] Notice that the 47th Fibonacci number is 2971215073 thus already too big for TeX and ε-TeX, a difficulty which our front page showed how to overcome (see subsection 23.22 for more). The `\Fibonacci` macro is completely expandable hence can be used for example within `\message` to write to the log and terminal.

It is even *f*-expandable (although not in only two steps, this could be added but does not matter here), thus if we are interested in knowing how many digits `F(1250)` has, suffices to use `\xintLen {\Fibonacci {1250}}` (which expands to 261), or if we want to check the formula `gcd(F(1859),F(1573))=F(gcd(1859,1573))=F(143)`, we only need[11]

```
\xintGCD{\Fibonacci{1859}}{\Fibonacci{1573}}=\Fibonacci{\xintGCD{1859}{1573}}
```

343358302784187294870275058337=343358302784187294870275058337

The `\Fibonacci` macro expanded its `\xintGCD{1859}{1573}` argument via the services of `\numexpr`: this step allows only things obeying the TeX bound, naturally! (but `F(2147483648)` would be rather big anyhow...).

2 Recent changes

Releases `1.09kb` (`[2014/02/13]`) and `1.09ka` (`[2014/02/05]`):

- bug fix (**xintexpr**): an aloof modification done by `1.09i` to `\xintNewExpr` had resulted in a spurious trailing space present in the outputs of all macros created by `\xintNewExpr`, making nesting of such macros impossible.
- bug fix (**xinttools**): `\xintBreakFor` and `\xintBreakForAndDo` were buggy when used in the last iteration of an `\xintFor` loop.
- bug fix (**xinttools**): `\xintSeq` from `1.09k` needed a `\chardef` which was missing from `xinttools.sty`, it was in `xint.sty`.

Release `1.09k` (`[2014/01/21]`):

- inside `\xintexpr..\relax` (and its variants) tacit multiplication is implied when a number or operand is followed directly with an opening parenthesis,
- the `"` for denoting (arbitrarily big) hexadecimal numbers is recognized by `\xintexpr` and its variants (package **xintbinhex** is required); a fractional hexadecimal part introduced by a dot `.` is allowed.
- re-organization of the first sections of the user manual.
- bug fix: forgotten loading time `"` catcode sanity check has been added.

For a more detailed change history, see section 22. Main recent additions:

Release `1.09j` (`[2014/01/09]`):

- the core division routines have been re-written for some (limited) efficiency gain, more pronounced for small divisors. As a result the computation of one thousand digits of π is close to three times faster than with earlier releases.
- a new macro `\xintXTrunc` is designed to produce thousands or even tens of thousands of digits of the decimal expansion of a fraction.
- the tacit multiplication done in `\xintexpr..\relax` on encountering a count register or variable, or a `\numexpr`, while scanning a (decimal) number, is extended to the case of a sub `\xintexpr`-ession.
- `\xintexpr` can now be used in an `\edef` with no `\xintthe` prefix.

[10] The implementation uses the (already once-expanded) integer only variant `\xintiiexpr` as `\romannumeral0\xintiieval..\relax`. [11] The `\xintGCD` macro is provided by the **xintgcd** package.

Release 1.09i ([2013/12/18]):

- \xintiiexpr is a variant of \xintexpr which is optimized to deal only with (long) integers, / does a euclidean quotient.
- \xintnumexpr, \xintthenumexpr, \xintNewNumExpr are renamed, respectively, \xintiexpr, \xinttheiexpr, \xintNewIExpr. The earlier denominations are kept but to be removed at some point.
- it is now possible within \xintexpr...\relax and its variants to use count, dimen, and skip registers or variables without explicit \the/\number: the parser inserts automatically \number and a tacit multiplication is implied when a register or variable immediately follows a number or fraction.
- **xinttools** defines \odef, \oodef, \fdef. These tools are provided for the case one uses the package macros in a non-expandable context, particularly \oodef which expands twice the macro replacement text and is thus a faster alternative to \edef. This can be significant when repeatedly making \definitions expanding to hundreds of digits.

Release 1.09h ([2013/11/28]):

- all macros of **xinttools** for which it makes sense are now declared \long.

Release 1.09g ([2013/11/22]):

- package **xinttools** is detached from **xint**, to make tools such as \xintFor, \xintApplyUnbraced, and \xintiloop available without the **xint** overhead.
- new expandable nestable loops \xintloop and \xintiloop.

Release 1.09f ([2013/11/04]):

- new \xintZapFirstSpaces, \xintZapLastSpaces, \xintZapSpaces, \xintZapSpacesB, for expandably stripping away leading and/or ending spaces.
- \xintCSVtoList by default uses \xintZapSpacesB to strip away spaces around commas (or at the start and end of the comma separated list).
- also the \xintFor loop will strip out all spaces around commas and at the start and the end of its list argument; and similarly for \xintForpair, \xintForthree, \xintForfour.
- \xintFor *et al.* accept all macro parameters from #1 to #9.

Release 1.09e ([2013/10/29]):

- new \xintintegers, \xintdimensions, \xintrationals for infinite \xintFor loops, interrupted with \xintBreakFor and \xintBreakForAndDo.
- new \xintifForFirst, \xintifForLast for the \xintFor and \xintFor* loops,
- the \xintFor and \xintFor* loops are now \long, the replacement text and the items may contain explicit \par's.
- new conditionals \xintifCmp, \xintifInt, \xintifOdd.
- the documentation has been enriched with various additional examples, such as the the quick sort algorithm illustrated or the computation of prime numbers (subsection 23.11, subsection 23.14, subsection 23.21).

Release 1.09c ([2013/10/09]):

- added bool and togl to the \xintexpr syntax; also added \xintboolexpr and \xintifboolexpr.
- \xintFor is a new type of loop, whose replacement text inserts the comma separated values or list items via macro parameters, rather than encapsulated in macros; the loops are nestable up to four levels (nine levels since 1.09f), and their replacement texts are allowed to close groups as happens with the tabulation in alignments,
- \xintApplyInline has been enhanced in order to be usable for generating rows (partially or completely) in an alignment,
- new command \xintSeq to generate (expandably) arithmetic sequences of (short) integers,

Release 1.09a ([2013/09/24]):

- \xintexpr..\relax and \xintfloatexpr..\relax admit functions in their syntax, with comma separated values as arguments, among them reduce, sqr, sqrt, abs, sgn, floor, ceil, quo, rem, round, trunc, float, gcd, lcm, max, min, sum, prd, add, mul, not, all, any, xor.

- comparison (<, >, =) and logical (|, &) operators.
- \xintNewExpr now works with the standard macro parameter character #.
- both regular \xintexpr-essions and commands defined by \xintNewExpr will work with comma separated lists of expressions,
- new commands \xintFloor, \xintCeil, \xintMaxof, \xintMinof (package **xintfrac**), \xintGCDof, \xintLCM, \xintLCMof (package **xintgcd**), \xintifLt, \xintifGt, \xintifSgn, \xintANDof, ...
- The arithmetic macros from package **xint** now filter their operands via \xintNum which means that they may use directly count registers and \numexpr-essions without having to prefix them by \the. This is thus similar to the situation holding previously but with **xintfrac** loaded.

See section 22 for more.

3 Some examples

The main initial goal is to allow computations with integers and fractions of arbitrary sizes.

Here are some examples. The first one uses only the base module **xint**, the next two require the **xintfrac** package, which deals with fractions. Then two examples with the **xintgcd** package, one with the **xintseries** package, and finally a computation with a float. Some inputs are simplified by the use of the **xintexpr** package.

123456^99:

\xintiPow{123456}{99}: 114738181166266556633273330008454586747025480423426102975889545437359089469703202762264705426632058346902708682211681334152500324038762776168953222117634295872033762216088606915850757168019716710712087697033536507377487778737784987816067499997983665812517232752154970541659566738491153332674854107560766971890623518995832377826369998110953239399323518999222056458781270149587767914316773543725385844594871559412151974163986661258969837372587167573949494355201709502618658016651990307184144322311696783769‌6

1234/56789 with 1500 digits after the decimal point:

\xintTrunc{1500}{1234/56789}\dots: 0.02172956030217119512581661941573191991406786525559527373258905774005529239817570304108189966366725950448150169927274648259346000105654264030005810984521650319604148690767578228177992216802549789571924140238426455827713113455070524221240028878832168201588335769251087358467308809804715701984539259363609149659264998503231259574917677719276620472274560214125975100811776928630544647731074679955625209107397559386500906865766257549877617144165243268942929088379791861099860888552360492348870379827079187870890489355332898976914543309443730299882019405166493511067284157143108700628642870978534575357903819401644684710067090457659053689975171247952948634418637412174893025057669619116378171829051400799450597827043969782880487418338058426808008593213474440472626741094225994470760182429695891810033633274049551849830072725351740653999894345735969994189015478349680395851309232421771822007783197450210428075859761573544172286886544929475778759971121167831798411664230748912641532691190195284298015460740636390850340735001496768740425082322280723379527725439785874024899188223071369455352268925320044374790892602440613499093134233742450122382855834756731057070911620208138900139111447639507651‌12

9620172920812129109510644667101023085456690556269700117980594833506488932715842856891299371357129021465424642096180598355315289932909542340946310024828752047051365581362587825106974942330380883621828170948599200549402172956030217119512581661941573191991406786525559527373258905774005529239817570304108189966366 7...

0.99^{-100} with 200 digits after the decimal point:

```
\xinttheexpr trunc(.99^-100,200)\relax\dots
```
: 2.73199902642902600384667172125783743550535164293857207083343057250824645551870534304481430137848061403680556247650192530703426968548915319461661227101592067191384034885148574794308647096392073177979303...

Computation of a Bezout identity with 7^200-3^200 and 2^200-1:

```
\xintAssign \xintBezout {\xinttheiexpr 7^200-3^200\relax}
                        {\xinttheiexpr 2^200-1\relax}\to\A\B\U\V\D
        \U$\times$(7^200-3^200)+\xintiOpp\V$\times$(2^200-1)=\D
```

-220045702773594816771390169652074193009609478853×(7^200-3^200)+14325894936276369318591306832683204654744168633877140891583816724789919211328201191274624371580391777549768571912876931442406050669914563361432056776967748 91×(2^200-1)=1803403947125

The Euclide algorithm applied to 22,206,980,239,027,589,097 and 8,169,486,210,102,119,256:[12]

```
\xintTypesetEuclideAlgorithm {22206980239027589097}{8169486210102119256}
```

$$
\begin{aligned}
22206980239027589097 &= 2 \times 8169486210102119256 + 5868007818823350585\\
8169486210102119256 &= 1 \times 5868007818823350585 + 2301478391278768671\\
5868007818823350585 &= 2 \times 2301478391278768671 + 1265051036265813243\\
2301478391278768671 &= 1 \times 1265051036265813243 + 1036427355012955428\\
1265051036265813243 &= 1 \times 1036427355012955428 + 228623681252857815\\
1036427355012955428 &= 4 \times 228623681252857815 + 121932630001524168\\
228623681252857815 &= 1 \times 121932630001524168 + 106691051251333647\\
121932630001524168 &= 1 \times 106691051251333647 + 15241578750190521\\
106691051251333647 &= 7 \times 15241578750190521 + 0
\end{aligned}
$$

$\sum_{n=1}^{500}(4n^2-9)^{-2}$ with each term rounded to twelve digits, and the sum to nine digits:

```
\def\coeff #1%
 {\xintiRound {12}{1/\xintiSqr{\the\numexpr 4*#1*#1-9\relax }[0]}}
\xintRound {9}{\xintiSeries {1}{500}{\coeff}[-12]}
```
: 0.062366080

The complete series, extended to infinity, has value $\frac{\pi^2}{144}-\frac{1}{162}$ = 0.062,366,079,945,836,595,346,844,45...[13] I also used (this is a lengthier computation than the one above) **xintseries** to evaluate the sum with 100,000 terms, obtaining 16 correct decimal digits for the complete sum. The coefficient macro must be redefined to avoid a \numexpr overflow, as \numexpr inputs must not exceed 2^31-1; my choice was:

```
\def\coeff #1%
{\xintiRound {22}{1/\xintiSqr{\xintiMul{\the\numexpr 2*#1-3\relax}
                                      {\the\numexpr 2*#1+3\relax}}[0]}}
```

[12] this example is computed tremendously faster than the other ones, but we had to limit the space taken by the output. [13] This number is typeset using the numprint package, with `\npthousandsep {,\hskip 1pt plus .5pt minus .5pt}`. But the breaking across lines works only in text mode. The number itself was (of course...) computed initially with **xint**, with 30 digits of π as input. See how **xint** may compute π from scratch.

Computation of $2^{999,999,999}$ with 24 significant figures:

`\numprint{\xintFloatPow [24] {2}{999999999}}` expands to:

$$2.306,488,000,584,534,696,558,06 \times 10^{301,029,995}$$

where the `\numprint` macro from the eponym package was used.

As an example of chaining package macros, let us consider the following code snippet within a file with filename `myfile.tex`:

```
\newwrite\outstream
\immediate\openout\outstream \jobname-out\relax
\immediate\write\outstream {\xintQuo{\xintPow{2}{1000}}{\xintFac{100}}}
% \immediate\closeout\outstream
```

The tex run creates a file `myfile-out.tex`, and then writes to it the quotient from the euclidean division of `2^{1000}` by `100!`. The number of digits is `\xintLen{\xintQuo{\xintPow{2}{1000}}{\xintFac{100}}}` which expands (in two steps) and tells us that `[2^{1000}/100!]` has 144 digits. This is not so many, let us print them here: 11481324 964150750548227839387255106625980551778418617288366347806582654189470473797041953579887663048435826506006150374953170779311862777482960 1.

For the sake of typesetting this documentation and not have big numbers extend into the margin and go beyond the page physical limits, I use these commands (not provided by the package):

```
\def\allowsplits #1{\ifx #1\relax \else #1\hskip 0pt plus 1pt \relax
                    \expandafter\allowsplits\fi}%
\def\printnumber #1% first ''fully'' expands its argument.
{\expandafter\allowsplits \romannumeral-`0#1\relax }
```

The `\printnumber` macro is not part of the package and would need additional thinking for more general use.[14] It may be used like this:

```
\printnumber {\xintQuo{\xintPow {2}{1000}}{\xintFac{100}}}
```

or as `\printnumber\mynumber` or `\printnumber{\mynumber}` if `\mynumber` was previously defined via a `\newcommand`, or a `\def`:

```
\def\mynumber {\xintQuo {\xintPow {2}{1000}}{\xintFac{100}}}
```

Just to show off (again), let's print 300 digits (after the decimal point) of the decimal expansion of `0.7^{-25}`:[15]

```
\np {\xinttheexpr trunc(.7^-25,300)\relax}\dots
```

7,456.739,985,837,358,837,609,119,727,341,853,488,853,339,101,579,533,
584,812,792,108,394,305,337,246,328,231,852,818,407,506,767,353,741,
490,769,900,570,763,145,015,081,436,139,227,188,742,972,826,645,967,
904,896,381,378,616,815,228,254,509,149,848,168,782,309,405,985,245,
368,923,678,816,256,779,083,136,938,645,362,240,130,036,489,416,562,
067,450,212,897,407,646,036,464,074,648,484,309,937,461,948,589...

This computation is with `\xinttheexpr` from package **xintexpr**, which allows to use standard infix notations and function names to access the package macros, such as here `trunc` which corresponds to the **xintfrac** macro `\xintTrunc`. The fraction `.7^-25` is first evaluated *exactly*; for some more complex inputs, such as `.7123045678952^-243`, the exact evaluation before truncation would be expensive, and (assuming one needs twenty

[14] as explained in a previous footnote, the `numprint` package may also be used, in text mode only (as the thousand separator seemingly ends up typeset in a `\hbox` when in math mode). [15] the `\np` typesetting macro is from the `numprint` package.

digits) one would rather use floating mode:

```
\xintDigits:=20; \np{\xintthefloatexpr .7123045678952^-243\relax}
```

`.7123045678952^-243` $\approx$ 6.342,022,117,488,416,127,3 $\times 10^{35}$

The exponent `-243` didn't have to be put inside parentheses, contrarily to what happens with some professional computational software.

4 Further illustrative examples within this document

The utilities provided by **xinttools** (section 23), some completely expandable, others not, are of independent interest. Their use is illustrated through various examples: among those, it is shown in subsection 23.28 how to implement in a completely expandable way the Quick Sort algorithm and also how to illustrate it graphically. Other examples include some dynamically constructed alignments with automatically computed prime number cells: one using a completely expandable prime test and `\xintApplyUnbraced` (subsection 23.11), another one with `\xintFor*` (subsection 23.21).

One has also a computation of primes within an `\edef` (subsection 23.13), with the help of `\xintiloop`. Also with `\xintiloop` an automatically generated table of factorizations (subsection 23.15).

The title page fun with Fibonacci numbers is continued in subsection 23.22 with `\xintFor*` joining the game.

The computations of π and $\log 2$ (subsection 29.11) using **xint** and the computation of the convergents of e with the further help of the **xintcfrac** package are among further examples. There is also an example of an interactive session, where results are output to the log or to a file.

Almost all of the computational results interspersed through the documentation are not hard-coded in the source of the document but just written there using the package macros, and were selected to not impact too much the compilation time.

5 General overview

The main characteristics are:

1. exact algebra on arbitrarily big numbers, integers as well as fractions,
2. floating point variants with user-chosen precision,
3. implemented via macros compatible with expansion-only context.

'Arbitrarily big': this means with less than `2^31-1=2147483647` digits, as most of the macros will have to compute the length of the inputs and these lengths must be treatable as TeX integers, which are at most `2147483647` in absolute value. This is a distant irrelevant upper bound, as no such thing can fit in TeX's memory! And besides, the true limitation is from the *time* taken by the expansion-compatible algorithms, as will be commented upon soon.

As just recalled, ten-digits numbers starting with a `3` already exceed the TeX bound on integers; and TeX does not have a native processing of floating point numbers (multiplication by a decimal number of a dimension register is allowed — this is used for example by the pgf basic math engine.)

TEX elementary operations on numbers are done via the non-expandable *advance, multiply,* and *divide* assignments. This was changed with ε-TEX's `\numexpr` which does expandable computations using standard infix notations with TEX integers. But ε-TEX did not modify the TEX bound on acceptable integers, and did not add floating point support.

The bigintcalc package by Heiko Oberdiek provided expandable operations (using some of `\numexpr` possibilities, when available) on arbitrarily big integers, beyond the TEX bound. The present package does this again, using more of `\numexpr` (**xint** requires the ε-TEX extensions) for higher speed, and also on fractions, not only integers. Arbitrary precision floating points operations are a derivative, and not the initial design goal.[16, 17]

The LATEX3 project has implemented expandably floating-point computations with 16 significant figures (l3fp), including special functions such as exp, log, sine and cosine.[18]

The **xint** package can be used for 24, 40, etc… significant figures but one rather quickly (not much beyond 100 figures) hits against a 'wall' created by the constraint of expandability: currently, multiplying out two one-hundred digits numbers takes circa 80 or 90 times longer than for two ten-digits numbers, which is reasonable, but multiplying out two one-thousand digits numbers takes more than 500 times longer than for two one hundred-digits numbers. This shows that the algorithm is drifting from quadratic to cubic in that range. On my laptop multiplication of two 1000-digits numbers takes some seconds, so it can not be done routinely in a document.[19]

The conclusion perhaps could be that it is in the end lucky that the speed gains brought by **xint** for expandable operations on big numbers do open some non-empty range of applicability in terms of the number of kept digits for routine floating point operations.

The second conclusion, somewhat depressing after all the hard work, is that if one really wants to do computations with *hundreds* of digits, one should drop the expandability requirement. And indeed, as clearly demonstrated long ago by the pi computing file by D. Roegel one can program TEX to compute with many digits at a much higher speed than what **xint** achieves: but, direct access to memory storage in one form or another seems a necessity for this kind of speed and one has to renounce at the complete expandability.[20] [21]

---

[16] currently (`v1.08`), the only non-elementary operation implemented for floating point numbers is the square-root extraction; no signed infinities, signed zeroes, `NaN`'s, error trapes…, have been implemented, only the notion of 'scientific notation with a given number of significant figures'. [17] multiplication of two floats with `P=\xinttheDigits` digits is first done exactly then rounded to `P` digits, rather than using a specially tailored multiplication for floating point numbers which would be more efficient (it is a waste to evaluate fully the multiplication result with `2P` or `2P-1` digits.) [18] at the time of writing the l3fp (exactly represented) floating point numbers have their exponents limited to ±9999. [19] without entering into too much technical details, the source of this 'wall' is that when dealing with two long operands, when one wants to pick some digits from the second one, one has to jump above all digits constituting the first one, which can not be stored away: expandability forbids assignments to memory storage. One may envision some sophisticated schemes, dealing with this problem in less naive ways, trying to move big chunks of data higher up in the input stream and come back to it later, etc...; but each 'better' algorithm adds overhead for the smaller inputs. For example, I have another version of addition which is twice faster on inputs with 500 digits or more, but it is slightly less efficient for 50 digits or less. This 'wall' dissuaded me to look into implementing 'intelligent' multiplication which would be sub-quadratic in a model where storing and retrieving from memory do not cost much. [20] I could, naturally, be proven wrong! [21] The LuaTEX project possibly makes endeavours such as **xint** appear even more insane that they are, in truth.

6 Origins of the package

Package `bigintcalc` by HEIKO OBERDIEK already provides expandable arithmetic operations on “big integers”, exceeding the TEX limits (of `2^{31}-1`), so why another[22] one?

I got started on this in early March 2013, via a thread on the `c.t.tex` usenet group, where ULRICH DIEZ used the previously cited package together with a macro (`\ReverseOrder`) which I had contributed to another thread.[23] What I had learned in this other thread thanks to interaction with ULRICH DIEZ and GL on expandable manipulations of tokens motivated me to try my hands at addition and multiplication.

I wrote macros `\bigMul` and `\bigAdd` which I posted to the newsgroup; they appeared to work comparatively fast. These first versions did not use the ε-TEX `\numexpr` primitive, they worked one digit at a time, having previously stored carry-arithmetic in 1200 macros.

I noticed that the `bigintcalc` package used`\numexpr` if available, but (as far as I could tell) not to do computations many digits at a time. Using `\numexpr` for one digit at a time for `\bigAdd` and `\bigMul` slowed them a tiny bit but avoided cluttering TEX memory with the 1200 macros storing pre-computed digit arithmetic. I wondered if some speed could be gained by using `\numexpr` to do four digits at a time for elementary multiplications (as the maximal admissible number for `\numexpr` has ten digits).

The present package is the result of this initial questioning.

7 Expansion matters

By convention in this manual f-expansion (“full expansion” or “full first expansion”) is the process of expanding repeatedly the first token seen until hitting against something not further expandable like an unexpandable TEX-primitive or an opening brace `{` or a character (inactive). For those familiar with LATEX3 (which is not used by **xint**) this is what is called in its documentation full expansion. Technically, macro arguments in **xint** which are submitted to such a f-expansion are so via prefixing them with `\romannumeral-`0`. An explicit or implicit space token stops such an expansion and is gobbled.

Most of the package macros, and all those dealing with computations, are expandable in the strong sense that they expand to their final result via this f-expansion. Again copied from LATEX3 documentation conventions, this will be signaled in the description of the

★ macro by a star in the margin. All[24] expandable macros of the **xint** packages completely expand in two steps.

Furthermore the macros dealing with computations, as well as many utilities from **xinttools**, apply this process of f-expansion to their arguments. Again from LATEX3's conventions this will be signaled by a margin annotation. Some additional parsing which is done by most macros of **xint** is indicated with a variant; and the extended fraction parsing done by most macros of **xintfrac** has its own symbol. When the argument has a priori to obey the TEX bound of `2147483647` it is systematically fed to a `\numexpr..\relax` hence the expansion is then a *complete* one, signaled with an x in the margin. This means not only complete expansion, but also that spaces are ignored, infix algebra is possible, count registers are allowed, etc…

f $\underset{f}{\text{Num}}$ $\underset{f}{\text{Frac}}$ $\underset{x}{\text{num}}$

[22] this section was written before the **xintfrac** package; the author is not aware of another package allowing expandable computations with arbitrarily big fractions. [23] the `\ReverseOrder` could be avoided in that circumstance, but it does play a crucial rôle here. [24] except `\xintloop` and `\xintiloop`.

$*f$ The \xintApplyInline and \xintFor* macros from **xinttools** apply a special iterated f-expansion, which gobbles spaces, to all those items which are found *unbraced* from left to right in the list argument; this is denoted specially as here in the margin. Some other $f \to \overset{\text{Frac}}{*f}$ macros such as \xintSum from **xintfrac** first do an f-expansion, then treat each found (braced or not) item (skipping spaces between such items) via the general fraction input parsing, this is signaled as here in the margin where the signification of the $*$ is thus a bit different from the previous case.

n, resp. o A few macros from **xinttools** do not expand, or expand only once their argument. This is also signaled in the margin with notations à la LaTeX3.

As the computations are done by f-expandable macros which f-expand their argument they may be chained up to arbitrary depths and still produce expandable macros.

Conversely, wherever the package expects on input a "big" integers, or a "fraction", f-expansion of the argument *must result in a complete expansion* for this argument to be acceptable.[25] The main exception is inside \xintexpr...\relax where everything will be expanded from left to right, completely.

Summary of important expansion aspects:

1. the macros f-expand their arguments, this means that they expand the first token seen (for each argument), then expand, etc..., until something un-expandable such as a digit or a brace is hit against. This example

   ```
   \def\x{98765}\def\y{43210}\xintAdd {\x}{\x\y}
   ```

   is *not* a legal construct, as the \y will remain untouched by expansion and not get converted into the digits which are expected by the sub-routines of \xintAdd. It is a \numexpr which will expand it and an arithmetic overflow will arise as 9876543210 exceeds the TeX bounds.

   With \xinttheexpr one could write \xinttheexpr \x+\x\y\relax, or \xintAdd \x{\xinttheexpr\x\y\relax}.

2. using \if...\fi constructs *inside* the package macro arguments requires suitably mastering TeXniques (\expandafter's and/or swapping techniques) to ensure that the f-expansion will indeed absorb the \else or closing \fi, else some error will arise in further processing. Therefore it is highly recommended to use the package provided conditionals such as \xintifEq, \xintifGt, \xintifSgn, \xintifOdd..., or, for LaTeX users and when dealing with short integers the etoolbox[26] expandable conditionals (for small integers only) such as \ifnumequal, \ifnumgreater, ... Use of *non-expandable* things such as \ifthenelse is impossible inside the arguments of **xint** macros.

   One can use naive \if..\fi things inside an \xinttheexpr-ession and cousins, as long as the test is expandable, for example

   ```
   \xinttheiexpr\ifnum3>2 143\else 33\fi 0^2\relax→2044900=1430^2
   ```

3. after the definition \def\x {12}, one can not use -\x as input to one of the package macros: the f-expansion will act only on the minus sign, hence do nothing. The only way is to use the \xintOpp macro, or perhaps here rather \xintiOpp which does maintains integer format on output, as they replace a number with its opposite.

[25] this is not quite as stringent as claimed here, see subsection 9.1 for more details.
[26] http://www.ctan.org/pkg/etoolbox

Again, this is otherwise inside an `\xinttheexpr`*-ession or* `\xintthefloatexpr`*-ession. There, the minus sign may prefix macros which will expand to numbers (or parentheses etc...)*

4. With the definition

   `\def\AplusBC #1#2#3{\xintAdd {#1}{\xintMul {#2}{#3}}}`

   one obtains an expandable macro producing the expected result, not in two, but rather in three steps: a first expansion is consumed by the macro expanding to its definition. As the package macros expand their arguments until no more is possible (regarding what comes first), this `\AplusBC` may be used inside them: `\xintAdd {\AplusBC {1}{2}{3}}{4}` does work and returns `11/1[0]`.

   If, for some reason, it is important to create a macro expanding in two steps to its final value, one may either do:

   `\def\AplusBC #1#2#3{\romannumeral-`0\xintAdd {#1}{\xintMul {#2}{#3}}}`

   or use the *lowercase* form of `\xintAdd`:

   `\def\AplusBC #1#2#3{\romannumeral0\xintadd {#1}{\xintMul {#2}{#3}}}`

   and then `\AplusBC` will share the same properties as do the other **xint** 'primitive' macros.

The `\romannumeral0` and `\romannumeral-`0` things above look like an invitation to hacker's territory; if it is not important that the macro expands in two steps only, there is no reason to follow these guidelines. Just chain arbitrarily the package macros, and the new ones will be completely expandable and usable one within the other.

Since release `1.07` the `\xintNewExpr` command automatizes the creation of such expandable macros:

`\xintNewExpr\AplusBC[3]{#1+#2*#3}`

creates the `\AplusBC` macro doing the above and expanding in two expansion steps.

8 User interface

Maintaining complete expandability is not for the faint of heart as it excludes doing macro definitions in the midst of the computation; in many cases, one does not need complete expandability, and definitions are allowed. In such contexts, there is no declaration for the user to be made to the package of a "typed variable" such as a long integer, or a (long) fraction, or possibly an `\xintexpr`-ession. Rather, the user has at its disposals the general tools of the TeX language: `\def` or (in LaTeX) `\newcommand`, and `\edef`.

The **xinttools** package provides `\oodef` which expands twice the replacement text, hence forces complete expansion when the top level of this replacement text is a call to one of the **xint** bundle macros, its arguments being themselves chains of such macros. There is also `\fdef` which will apply f-expansion to the replacement text. Both are in such uses faster alternatives to `\edef`.

This section will explain the various inputs which are recognized by the package macros and the format for their outputs. Inputs have mainly five possible shapes:

1. expressions which will end up inside a `\numexpr..\relax`,

2. long integers in the strict format (no +, no leading zeroes, a count register or variable must be prefixed by `\the` or `\number`)
3. long integers in the general format allowing both `-` and + signs, then leading zeroes, and a count register or variable without prefix is allowed,
4. fractions with numerators and denominators as in the previous item, or also decimal numbers, possibly in scientific notation (with a lowercase `e`), and also optionally the semi-private `A/B[N]` format,
5. and finally expandable material understood by the `\xintexpr` parser.

Outputs are mostly of the following types:

1. long integers in the strict format,
2. fractions in the `A/B[N]` format where `A` and `B` are both strict long integers, and `B` is positive,
3. numbers in scientific format (with a lowercase `e`),
4. the private `\xintexpr` format which needs the `\xintthe` prefix in order to end up on the printed page (or get expanded in the log) or be used as argument to the package macros.

8.1 Input formats

num
x Some macro arguments are by nature 'short' integers, *i.e.* less than (or equal to) in absolute value 2,147,483,647. This is generally the case for arguments which serve to count or index something. They will be embedded in a `\numexpr..\relax` hence on input one may even use count registers or variables and expressions with infix operators. Notice though that `-(..stuff..)` is surprisingly not legal in the `\numexpr` syntax!

But **xint** is mainly devoted to big numbers; the allowed input formats for 'long numbers' and 'fractions' are:

f 1. the strict format is for some macros of **xint** which only f-expand their arguments. After this f-expansion the input should be a string of digits, optionally preceded by a unique minus sign. The first digit can be zero only if the number is zero. A plus sign is not accepted. `-0` is not legal in the strict format. A count register can serve as argument of such a macro only if prefixed by `\the` or `\number`. Most macros of **xint** are like `\xintAdd` and accept the extended format described in the next item; they may have a 'strict' variant such as `\xintiiAdd` which remains available even with **xintfrac** loaded, for optimization purposes.

2. the macro \xintNum normalizes into strict format an input having arbitrarily many minus and plus signs, followed by a string of zeroes, then digits:
   `\xintNum {+-+-+----++-++----00000000009876543210}=-9876543210`
   Num *f* The extended integer format is thus for the arithmetic macros of **xint** which automatically parse their arguments via this \xintNum.[27]

Frac *f*

3. the fraction format is what is expected by the macros of **xintfrac**: a fraction is constituted of a numerator A and optionally a denominator B, separated by a forward slash / and A and B may be macros which will be automatically given to \xintNum. Each of A and B may be decimal numbers (the decimal mark must be a .). Here is an example:[28]
   `\xintAdd {+--0367.8920280/-++278.289287}{-109.2882/+270.12898}`
   Scientific notation is accepted for both numerator and denominator of a fraction, and is produced on output by \xintFloat:
   `\xintAdd{10.1e1}{101.010e3}=101111/1[0]`
   `\xintFloatAdd{10.1e1}{101.010e3}=1.011110000000000e5`
   `\xintPow {2}{100}=1267650600228229401496703205376/1[0]`
   `\xintFloat{\xintPow {2}{100}}=1.267650600228229e30`
   `\xintFloatPow {2}{100}=1.267650600228229e30`
   Produced fractions having a denominator equal to one are, as a general rule, nevertheless printed as fractions. In math mode \xintFrac will remove such dummy denominators, and in inline text mode one has \xintPRaw with the similar effect.
   `\xintPRaw{\xintAdd{10.1e1}{101.010e3}}=101111`
   `\xintRaw{1.234e5/6.789e3}=1234/6789[2]`

4. the expression format is for inclusion in an \xintexpr...\relax, it uses infix notations, function names, complete expansion, and is described in its devoted section (section 21).

Generally speaking, there should be no spaces among the digits in the inputs (in arguments to the package macros). Although most would be harmless in most macros, there are some cases where spaces could break havoc. So the best is to avoid them entirely.

This is entirely otherwise inside an \xintexpr-ession, where spaces are ignored (except when they occur inside arguments to some macros, thus escaping the \xintexpr parser). See the documentation.

Even with **xintfrac** loaded, some macros by their nature can not accept fractions on input. Those parsing their inputs through \xintNum will accept a fraction reducing to an integer. For example `\xintQuo {100/2}{12/3}` works, because its arguments are, after simplification, integers.

With **xintfrac** loaded, a number may be empty or start directly with a decimal point:
`\xintRaw{}=\xintRaw{.}=0/1[0]`
`\xintPow{-.3/.7}{11}=-177147/1977326743[0]`
`\xinttheexpr (-.3/.7)^11\relax=-177147/1977326743[0]`
It is also licit to use \A/\B as input if each of \A and \B expands (in the sense previously described) to a “decimal number” as examplified above by the numerators and denominators (thus, possibly with a ‘scientific’ exponent part, with a lowercase ‘e’). Or one may

[27] A LaTeX \value{countername} is accepted as macro argument. [28] the square brackets one sees in various outputs are explained near the end of this section.

have just one macro `\C` which expands to such a "fraction with optional decimal points", or mixed things such as `\A 245/7.77`, where the numerator will be the concatenation of the expansion of `\A` and `245`. But, as explained already `123\A` is a no-go, *except inside an* `\xintexpr`*-ession*!

The scientific notation is necessarily (except in `\xintexpr..\relax`) with a lowercase `e`. It may appear both at the numerator and at the denominator of a fraction.

`\xintRaw {+--+1253.2782e++--3/---0087.123e---5}=-12532782/87123[7]`

Num f Arithmetic macros of **xint** which parse their arguments automatically through `\xintNum` are signaled by a special symbol in the margin. This symbol also means that these arguments may contain to some extent infix algebra with count registers, see the section Use of count registers.

With **xintfrac** loaded the symbol Num f means that a fraction is accepted if it is a whole number in disguise; and for macros accepting the full fraction format with no restriction there is the corresponding symbol in the margin. Frac f

The **xintfrac** macros generally output their result in `A/B[n]` format, representing the fraction `A/B` times `10^n`.

This format with a trailing `[n]` (possibly, `n=0`) is accepted on input but it presupposes that the numerator and denominator `A` and `B` are in the strict integer format described above. So `16000/289072[17]` or `3[-4]` are authorized and it is even possible to use `\A/\B[17]` if `\A` expands to `16000` and `\B` to `289072`, or `\A` if `\A` expands to `3[-4]`. However, NEITHER the numerator NOR the denominator may then have a decimal point. And, for this format, ONLY the numerator may carry a UNIQUE minus sign (and no superfluous leading zeroes; and NO plus sign). IMPORTANT!

It is allowed for user input but the parsing is minimal and it is mandatory to follow the above rules. This reduced flexibility, compared to the format without the square brackets, allows nesting package macros without too much speed impact.

8.2 Output formats

With package **xintfrac** loaded, the routines `\xintAdd`, `\xintSub`, `\xintMul`, `\xintPow`, `\xintSum`, `\xintPrd` are modified to allow fractions on input,[29] [30] [31] [32] and produce on output a fractional number `f=A/B[n]` where `A` and `B` are integers, with `B` positive, and `n` is a "short" integer. This represents `(A/B)` times `10^n`. The fraction `f` may be, and generally is, reducible, and `A` and `B` may well end up with zeroes (*i.e.* `n` does not contain all powers of 10). Conversely, this format is accepted on input (and is parsed more quickly than fractions containing decimal points; the input may be a number without denominator).[33]

---

[29] the power function does not accept a fractional exponent. Or rather, does not expect, and errors will result if one is provided. [30] macros `\xintiAdd`, `\xintiSub`, `\xintiMul`, `\xintiPow`, are the original ones dealing only with integers. They are available as synonyms, also when **xintfrac** is not loaded. With **xintfrac** loaded they accept on input also fractions, if these fractions reduce to integers, and then the output format is the original **xint**'s one. The macros `\xintiiAdd`, `\xintiiSub`, `\xintiiMul`, `\xintiiPow`, `\xintiiSum`, `\xintiiPrd` are strictly integer-only: they skip the overhead of parsing their arguments via `\xintNum`. [31] also `\xintCmp`, `\xintSgn`, `\xintGeq`, `\xintOpp`, `\xintAbs`, `\xintMax`, `\xintMin` are extended to fractions; and the last four have the integer-only variants `\xintiOpp`, `\xintiAbs`, `\xintiMax`, `\xintiMin`. [32] and `\xintFac`, `\xintQuo`, `\xintRem`, `\xintDivision`, `\xintFDg`, `\xintLDg`, `\xintOdd`, `\xintMON`, `\xintMMON` all accept a fractional input as long as it reduces to an integer. [33] at each stage of the computations, the sum of `n` and the length of `A`, or of the absolute value of `n` and the length of `B`, must be kept less than `2^{31}-9`.

Thus loading **xintfrac** not only relaxes the format of the inputs; it also modifies the format of the outputs: except when a fraction is filtered on output by \xintIrr or \xint-RawWithZeros, or \xintPRaw, or by the truncation or rounding macros, or is given as argument in math mode to \xintFrac, the output format is normally of the A/B[n] form (which stands for (A/B)×10^n). The A and B may end in zeroes (*i.e*, n does not represent all powers of ten), and will generally have a common factor. The denominator B is always strictly positive.

A macro \xintFrac is provided for the typesetting (math-mode only) of such a 'raw' output. The command \xintFrac is not accepted as input to the package macros, it is for typesetting only (in math mode).

The macro \xintRaw prints the fraction directly from its internal representation in A/B[n] form. The macro \xintPRaw does the same but without printing the [n] if n=0 and without printing /1 if B=1.

The macro \xintIrr reduces the fraction to its irreducible form C/D (without a trailing [0]), and it prints the D even if D=1.

The macro \xintNum from package **xint** is extended: it now does like \xintIrr, raises an error if the fraction did not reduce to an integer, and outputs the numerator. This macro should be used when one knows that necessarily the result of a computation is an integer, and one wants to get rid of its denominator /1 which would be left by \xintIrr (or one can use \xintPRaw on top of \xintIrr).

See also the documentations of \xintTrunc, \xintiTrunc, \xintXTrunc, \xint-Round, \xintiRound and \xintFloat.

The \xintiAdd, \xintiSub, \xintiMul, \xintiPow, and some others accept fractions on input under the condition that they are (big) integers in disguise and then output a (possibly big) integer, without fraction slash nor trailing [n].

The \xintiiAdd, \xintiiSub, \xintiiMul, \xintiiPow, and some others with 'ii' in their names accept on input only integers in strict format (skipping the overhead of the \xintNum parsing) and output naturally a (possibly big) integer, without fraction slash nor trailing [n].

8.3 Multiple outputs

Some macros have an output consisting of more than one number or fraction, each one is then returned within braces. Examples of multiple-output macros are \xintDivision which gives first the quotient and then the remainder of euclidean division, \xintBezout from the **xintgcd** package which outputs five numbers, \xintFtoCv from the **xintcfrac** package which returns the list of the convergents of a fraction, ... section 11 and section 12 mention utilities, expandable or not, to cope with such outputs.

Another type of multiple outputs is when using commas inside \xintexpr..\relax:

\xinttheiexpr 10!,2^20,lcm(1000,725)\relax→3628800,1048576,29000

9 Use of TEX registers and variables

9.1 Use of count registers

Inside `\xintexpr..\relax` and its variants, a count register or count control sequence is automatically unpacked using `\number`, with tacit multiplication: `1.23\counta` is like `1.23*\number\counta`. There is a subtle difference between count *registers* and count *variables*. In `1.23*\counta` the unpacked `\counta` variable defines a complete operand thus `1.23*\counta 7` is a syntax error. But `1.23*\count0` just replaces `\count0` by `\number\count0` hence `1.23*\count0 7` is like `1.23*57` if `\count0` contains the integer value `5`.

Regarding now the package macros, there is first the case of arguments having to be short integers: this means that they are fed to a `\numexpr...\relax`, hence submitted to a *complete expansion* which must deliver an integer, and count registers and even algebraic expressions with them like `\mycountA+\mycountB*17-\mycountC/12+\mycountD` are admissible arguments (the slash stands here for the rounded integer division done by `\numexpr`). This applies in particular to the number of digits to truncate or round with, to the indices of a series partial sum, ...

The macros allowing the extended format for long numbers or dealing with fractions will *to some extent* allow the direct use of count registers and even infix algebra inside their arguments: a count register `\mycountA` or `\count 255` is admissible as numerator or also as denominator, with no need to be prefixed by `\the` or `\number`. It is possible to have as argument an algebraic expression as would be acceptable by a `\numexpr...\relax`, under this condition: *each of the numerator and denominator is expressed with at most* eight *tokens*.[34] The slash for rounded division in a `\numexpr` should be written with braces `{/}` to not be confused with the **xintfrac** delimiter between numerator and denominator (braces will be removed internally). Example: `\mycountA+\mycountB{/}17/1+\mycountA*\mycountB`, or `\count 0+\count 2{/}17/1+\count 0*\count 2`, but in the latter case the numerator has the maximal allowed number of tokens (the braced slash counts for only one).

`\cnta 10 \cntb 35 \xintRaw {\cnta+\cntb{/}17/1+\cnta*\cntb}`->12/351[0]

For longer algebraic expressions using count registers, there are two possibilities:

1. encompass each of the numerator and denominator in `\the\numexpr...\relax`,
2. encompass each of the numerator and denominator in `\numexpr {...}\relax`.

```
\cnta 100 \cntb 10 \cntc 1
\xintPRaw {\numexpr {\cnta*\cnta+\cntb*\cntb+\cntc*\cntc+
                    2*\cnta*\cntb+2*\cnta*\cntc+2*\cntb*\cntc}\relax/%
          \numexpr {\cnta*\cnta+\cntb*\cntb+\cntc*\cntc}\relax }
                                        12321/10101
```

The braces would not be accepted as regular `\numexpr`-syntax: and indeed, they are removed at some point in the processing.

IMPORTANT! [34] Attention! there is no problem with a LaTeX `\value{countername}` if if comes first, but if it comes later in the input it will not get expanded, and braces around the name will be removed and chaos will ensues inside a `\numexpr`. One should enclose the whole input in `\the\numexpr...\relax` in such cases.

9.2 Dimensions

⟨*dimen*⟩ variables can be converted into (short) integers suitable for the **xint** macros by prefixing them with `\number`. This transforms a dimension into an explicit short integer which is its value in terms of the `sp` unit (`1/65536 pt`). When `\number` is applied to a ⟨*glue*⟩ variable, the stretch and shrink components are lost.

For LaTeX users: a length is a ⟨*glue*⟩ variable, prefixing a length command defined by `\newlength` with `\number` will thus discard the `plus` and `minus` glue components and return the dimension component as described above, and usable in the **xint** bundle macros.

This conversion is done automatically inside an `\xintexpr`-essions, with tacit multiplication implied if prefixed by some (integral or decimal) number.

One may thus compute areas or volumes with no limitations, in units of `sp^2` respectively `sp^3`, do arithmetic with them, compare them, etc..., and possibly express some final result back in another unit, with the suitable conversion factor and a rounding to a given number of decimal places.

A table of dimensions illustrates that the internal values used by TEX do not correspond always to the closest rounding. For example a millimeter exact value in terms of `sp` units is `72.27/10/2.54*65536=186467.981...` and TEX uses internally `186467sp` (it thus appears that TEX truncates to get an integral multiple of the `sp` unit).

Unit	definition	Exact value in sp units	TEX's value in sp units	Relative error
cm	0.01 m	236814336/127 = 1864679.811...	1864679	-0.0000%
mm	0.001 m	118407168/635 = 186467.981...	186467	-0.0005%
in	2.54 cm	118407168/25 = 4736286.720...	4736286	-0.0000%
pc	12 pt	786432/1 = 786432.000...	786432	0%
pt	1/72.27 in	65536/1 = 65536.000...	65536	0%
bp	1/72 in	1644544/25 = 65781.760...	65781	-0.0012%
3bp	1/24 in	4933632/25 = 197345.280...	197345	-0.0001%
12bp	1/6 in	19734528/25 = 789381.120...	789381	-0.0000%
72bp	1 in	118407168/25 = 4736286.720...	4736286	-0.0000%
dd	1238/1157 pt	81133568/1157 = 70124.086...	70124	-0.0001%
11dd	11*1238/1157 pt	892469248/1157 = 771364.950...	771364	-0.0001%
12dd	12*1238/1157 pt	973602816/1157 = 841489.037...	841489	-0.0000%
sp	1/65536 pt	1/1 = 1.000...	1	0%

TEX dimensions

There is something quite amusing with the Didot point. According to the TEXBook, `1157` dd=`1238` pt. The actual internal value of `1` dd in TEX is `70124` sp. We can use **xintcfrac** to display the list of centered convergents of the fraction `70124/65536`:

`\xintListWithSep{, }{\xintFtoCCv{70124/65536}}`

`1/1, 15/14, 61/57, 107/100, 1452/1357, 17531/16384`, and we don't find `1238/1157` therein, but another approximant `1452/1357`!

And indeed multiplying `70124/65536` by `1157`, and respectively `1357`, we find the approximations (wait for more, later):

"`1157` dd"=`1237.998474121093...pt`

"`1357` dd"=`1451.999938964843...pt`

and we seemingly discover that `1357` dd=`1452` pt is *far more accurate* than the TEXBook formula `1157 dd=1238 pt` ! The formula to compute `N dd` was

`\xinttheexpr trunc(N\dimexpr 1dd\relax/\dimexpr 1pt\relax,12)\relax}`

What's the catch? The catch is that TeX *does not* compute `1157 dd` like we just did:

`1157 dd=\number\dimexpr 1157dd\relax/65536=1238.000000000000...pt`

`1357 dd=\number\dimexpr 1357dd\relax/65536=1452.001724243164...pt`

We thus discover that TeX (or rather here, e-TeX, but one can check that this works the same in TeX82), uses indeed `1238/1157` as a conversion factor, and necessarily intermediate computations are done with more precision than is possible with only integers less than `2^31` (or `2^30` for dimensions). Hence the `1452/1357` ratio is irrelevant, a misleading artefact of the necessary rounding (or, as we see, truncating) for one `dd` as an integral number of `sp`'s.

Let us now use `\xintexpr` to compute the value of the Didot point in millimeters, if the above rule is exactly verified:

`\xinttheexpr trunc(1238/1157*25.4/72.27,12)\relax=0.376065027442...mm`

This fits very well with the possible values of the Didot point as listed in the Wikipedia Article. The value `0.376065 mm` is said to be the *the traditional value in European printers' offices*. So the `1157 dd=1238 pt` rule refers to this Didot point, or more precisely to the *conversion factor* to be used between this Didot and TeX points.

The actual value in millimeters of exactly one Didot point as implemented in TeX is

`\xinttheexpr trunc(\dimexpr 1dd\relax/65536/72.27*25.4,12)\relax`
`=0.376064563929...mm`

The difference of circa 5Å is arguably tiny!

By the way the *European printers' offices* (dixit Wikipedia) *Didot* is thus exactly

`\xinttheexpr reduce(.376065/(25.4/72.27))\relax=543564351/508000000 pt`

and the centered convergents of this fraction are `1/1`, `15/14`, `61/57`, `107/100`, `1238/1157`, `11249/10513`, `23736/22183`, `296081/276709`, `615898/575601`, `11382245/10637527`, `22148592/20699453`, `188570981/176233151`, `543564351/508000000`. We do recover the `1238/1157` therein!

10 \ifcase, \ifnum, ... constructs

When using things such as `\ifcase \xintSgn{\A}` one has to make sure to leave a space after the closing brace for TeX to stop its scanning for a number: once TeX has finished expanding `\xintSgn{\A}` and has so far obtained either `1`, `0`, or `-1`, a space (or something 'unexpandable') must stop it looking for more digits. Using `\ifcase\xintSgn\A` without the braces is very dangerous, because the blanks (including the end of line) following `\A` will be skipped and not serve to stop the number which `\ifcase` is looking for. With `\def \A{1}`:

```
\ifcase \xintSgn\A       0\or OK\else ERROR\fi   ---> gives ERROR
\ifcase \xintSgn\A\space 0\or OK\else ERROR\fi   ---> gives OK
\ifcase \xintSgn{\A}     0\or OK\else ERROR\fi   ---> gives OK
```

In order to use successfully `\if...\fi` constructions either as arguments to the **xint** bundle expandable macros, or when building up a completely expandable macro of one's own, one needs some TeXnical expertise (see also item 2 on page 15).

It is thus much to be recommended to opt rather for already existing expandable branching macros, such as the ones which are provided by **xint**: `\xintSgnFork`, `\xintifSgn`,

\xintifZero, \xintifOne, \xintifNotZero, \xintifTrueAelseB, \xintifCmp, \xintifGt, \xintifLt, \xintifEq, \xintifOdd, and \xintifInt. See their respective documentations. All these conditionals always have either two or three branches, and empty brace pairs {} for unused branches should not be forgotten.

If these tests are to be applied to standard TEX short integers, it is more efficient to use (under LATEX) the equivalent conditional tests from the etoolbox[35] package.

11 Assignments

It might not be necessary to maintain at all times complete expandability. A devoted syntax is provided to make these things more efficient, for example when using the \xintDivision macro which computes both quotient and remainder at the same time:

```
\xintAssign\xintDivision{100}{3}\to\A\B
\xintAssign\xintDivision{\xintiPow {2}{1000}}{\xintFac{100}}\to\A\B
```

gives \meaning\A: macro:->114813249641507505482278393872551066259805517784186172883663478065826541894704737970419535798876630484358265060061503749531707793118627774829601 and \meaning\B: macro:->549362945213398322513812878622391280734105004984760505953218996123132766490228838813287870244458207512960315204105480496462508313856765262438683720568069376.

Another example (which uses \xintBezout from the **xintgcd** package):

```
\xintAssign\xintBezout{357}{323}\to\A\B\U\V\D
```

is equivalent to setting \A to 357, \B to 323, \U to -9, \V to -10, and \D to 17. And indeed (-9)×357-(-10)×323=17 is a Bezout Identity.

Thus, what \xintAssign does is to first apply an *f*-expansion to what comes next; it then defines one after the other (using \def; an optional argument allows to modify the expansion type, see subsection 23.24 for details), the macros found after \to to correspond to the successive braced contents (or single tokens) located prior to \to.

```
\xintAssign\xintBezout{3570902836026}{200467139463}\to\A\B\U\V\D
```

gives then \U: macro:->5812117166, \V: macro:->103530711951 and \D=3.

In situations when one does not know in advance the number of items, one has \xintAssignArray or its synonym \xintDigitsOf:

```
\xintDigitsOf\xintiPow{2}{100}\to\DIGITS
```

This defines \DIGITS to be macro with one parameter, \DIGITS{0} gives the size N of the array and \DIGITS{n}, for n from 1 to N then gives the nth element of the array, here the nth digit of 2^{100}, from the most significant to the least significant. As usual, the generated macro \DIGITS is completely expandable (in two steps). As it wouldn't make much sense to allow indices exceeding the TEX bounds, the macros created by \xintAssignArray put their argument inside a \numexpr, so it is completely expanded and may be a count register, not necessarily prefixed by \the or \number. Consider the following code snippet:

```
\newcount\cnta
\newcount\cntb
\begingroup
\xintDigitsOf\xintiPow{2}{100}\to\DIGITS
```

[35] http://www.ctan.org/pkg/etoolbox

```
\cnta = 1
\cntb = 0
\loop
\advance \cntb \xintiSqr{\DIGITS{\cnta}}
\ifnum \cnta < \DIGITS{0}
\advance\cnta 1
\repeat

|2^{100}| (=\xintiPow {2}{100}) has \DIGITS{0} digits and the sum of
their squares is \the\cntb. These digits are, from the least to
the most significant: \cnta = \DIGITS{0}
\loop \DIGITS{\cnta}\ifnum \cnta > 1 \advance\cnta -1 , \repeat.
\endgroup
```

`2^{100}` (=1267650600228229401496703205376) has 31 digits and the sum of their squares is 679. These digits are, from the least to the most significant: 6, 7, 3, 5, 0, 2, 3, 0, 7, 6, 9, 4, 1, 0, 4, 9, 2, 2, 8, 2, 2, 0, 0, 6, 0, 5, 6, 7, 6, 2, 1.

Warning: `\xintAssign`, `\xintAssignArray` and `\xintDigitsOf` *do not do any check* on whether the macros they define are already defined.

12 Utilities for expandable manipulations

The package now has more utilities to deal expandably with 'lists of things', which were treated un-expandably in the previous section with `\xintAssign` and `\xintAssignArray`: `\xintReverseOrder` and `\xintLength` since the first release, `\xintApply` and `\xintListWithSep` since `1.04`, `\xintRevWithBraces`, `\xintCSVtoList`, `\xintNthElt` since `1.06`, `\xintApplyUnbraced`, since `1.06b`, `\xintloop` and `\xintiloop` since `1.09g`.[36]

As an example the following code uses only expandable operations:

```
|2^{100}| (=\xintiPow {2}{100}) has \xintLen{\xintiPow {2}{100}} digits
and the sum of their squares is
\xintiiSum{\xintApply {\xintiSqr}{\xintiPow {2}{100}}}.
These digits are, from the least to the most significant:
\xintListWithSep {, }{\xintRev{\xintiPow {2}{100}}}. The thirteenth most
significant digit is \xintNthElt{13}{\xintiPow {2}{100}}. The seventh
least significant one is \xintNthElt{7}{\xintRev{\xintiPow {2}{100}}}.
```

`2^{100}` (=1267650600228229401496703205376) has 31 digits and the sum of their squares is 679. These digits are, from the least to the most significant: 6, 7, 3, 5, 0, 2, 3, 0, 7, 6, 9, 4, 1, 0, 4, 9, 2, 2, 8, 2, 2, 0, 0, 6, 0, 5, 6, 7, 6, 2, 1. The thirteenth most significant digit is 8. The seventh least significant one is 3.

It would be more efficient to do once and for all `\oodef\z{\xintiPow {2}{100}}`, and then use `\z` in place of `\xintiPow {2}{100}` everywhere as this would spare the CPU some repetitions.

Expandably computing primes is done in subsection 23.10.

[36] All these utilities, as well as `\xintAssign`, `\xintAssignArray` and the `\xintFor` loops are now available from the **`xinttools`** package, independently of the big integers facilities of **`xint`**.

13 A new kind of for loop

As part of the utilities coming with the **xinttools** package, there is a new kind of for loop, `\xintFor`. Check it out (subsection 23.17).

14 A new kind of expandable loop

Also included in **xinttools**, `\xintiloop` is an expandable loop giving access to an iteration index, without using count registers which would break expandability. Check it out (subsection 23.13).

15 Exceptions (error messages)

In situations such as division by zero, the package will insert in the TEX processing an undefined control sequence (we copy this method from the `bigintcalc` package). This will trigger the writing to the log of a message signaling an undefined control sequence. The name of the control sequence is the message. The error is raised *before* the end of the expansion so as to not disturb further processing of the token stream, after completion of the operation. Generally the problematic operation will output a zero. Possible such error message control sequences:

```
\xintError:ArrayIndexIsNegative
\xintError:ArrayIndexBeyondLimit
\xintError:FactorialOfNegativeNumber
\xintError:FactorialOfTooBigNumber
\xintError:DivisionByZero
\xintError:NaN
\xintError:FractionRoundedToZero
\xintError:NotAnInteger
\xintError:ExponentTooBig
\xintError:TooBigDecimalShift
\xintError:TooBigDecimalSplit
\xintError:RootOfNegative
\xintError:NoBezoutForZeros
\xintError:ignored
\xintError:removed
\xintError:inserted
\xintError:bigtroubleahead
\xintError:unknownfunction
```

16 Common input errors when using the package macros

Here is a list of common input errors. Some will cause compilation errors, others are more annoying as they may pass through unsignaled.

- using - to prefix some macro: `-\xintiSqr{35}/271`.[37]

[37] to the contrary, this *is* allowed inside an `\xintexpr`-ession.

- using one pair of braces too many \xintIrr{{\xintiPow {3}{13}}/243} (the computation goes through with no error signaled, but the result is completely wrong).
- using [] and decimal points at the same time 1.5/3.5[2], or with a sign in the denominator 3/-5[7]. The scientific notation has no such restriction, the two inputs 1.5/-3.5e-2 and -1.5e2/3.5 are equivalent: \xintRaw{1.5/-3.5e-2} =-15/35[2], \xintRaw{-1.5e2/3.5}=-15/35[2].
- specifying numerators and denominators with macros producing fractions when **xintfrac** is loaded: \edef\x{\xintMul {3}{5}/\xintMul{7}{9}}. This expands to 15/1[0]/63/1[0] which is invalid on input. Using this \x in a fraction macro will most certainly cause a compilation error, with its usual arcane and undecipherable accompanying message. The fix here would be to use \xintiMul. The simpler alternative with package **xintexpr**: \xinttheexpr 3*5/(7*9)\relax.
- generally speaking, using in a context expecting an integer (possibly restricted to the TeX bound) a macro or expression which returns a fraction: \xinttheexpr 4/2 \relax outputs 4/2[0], not 2. Use \xintNum {\xinttheexpr 4/2\relax} or \xinttheiexpr 4/2\relax (which rounds the result to the nearest integer, here, the result is already an integer) or \xinttheiiexpr 4/2\relax (but / therein is euclidean quotient, which on positive operands is like truncating to the integer part, not rounding).

17 Package namespace

Inner macros of **xinttools**, **xint**, **xintfrac**, **xintexpr**, **xintbinhex**, **xintgcd**, **xintseries**, and **xintcfrac** all begin either with \XINT_ or with \xint_.[38] The package public commands all start with \xint. Some other control sequences are used only as delimiters, and left undefined, they may have been defined elsewhere, their meaning doesn't matter and is not touched.

xinttools defines \odef, \oodef, \fdef, but only if macros with these names do not already exist (\xintoodef etc... are defined anyhow for use in \xintAssign and \xintAssignArray).

The **xint** packages presuppose that the \space, \empty, \m@ne, \z@ and \@ne control sequences have their meanings as in Plain TeX or LaTeX2e.

18 Loading and usage

```
Usage with LaTeX: \usepackage{xinttools}
                  \usepackage{xint}       % (loads xinttools)
                  \usepackage{xintfrac}   % (loads xint)
                  \usepackage{xintexpr}   % (loads xintfrac)

                  \usepackage{xintbinhex} % (loads xint)
```

[38] starting with release 1.06b the style files use for macro names a more modern underscore _ rather than the @ sign. A handful of private macros starting with \XINT do not have the underscore for technical reasons: \XINTsetupcatcodes, \XINTdigits and macros with names starting with XINTinFloat or XINTinfloat.

```
                    \usepackage{xintgcd}    % (loads xint)
                    \usepackage{xintseries} % (loads xintfrac)
                    \usepackage{xintcfrac}  % (loads xintfrac)

Usage with TeX:     \input xinttools.sty\relax
                    \input xint.sty\relax       % (loads xinttools)
                    \input xintfrac.sty\relax   % (loads xint)
                    \input xintexpr.sty\relax   % (loads xintfrac)

                    \input xintbinhex.sty\relax % (loads xint)
                    \input xintgcd.sty\relax    % (loads xint)
                    \input xintseries.sty\relax % (loads xintfrac)
                    \input xintcfrac.sty\relax  % (loads xintfrac)
```

We have added, directly copied from packages by Heiko Oberdiek, a mechanism of reload and ε-TEX detection, especially for Plain TEX. As ε-TEX is required, the executable `tex` can not be used, `etex` or `pdftex` (version `1.40` or later) or ..., must be invoked. Each package refuses to be loaded twice and automatically loads the other components on which it has dependencies.[39]

Also initially inspired from the Heiko Oberdiek packages we have included a complete catcode protection mecanism. The packages may be loaded in any catcode configuration satisfying these requirements: the percent is of category code comment character, the backslash is of category code escape character, digits have category code other and letters have category code letter. Nothing else is assumed, and the previous configuration is restored after the loading of each one of the packages.

This is for the loading of the packages.

For the input of numbers as macro arguments the minus sign must have its standard category code (“*other*”). Similarly the slash used for fractions must have its standard category code. And the square brackets, if made use of in the input, also must be of category code *other*. The ‘e’ of the scientific notation must be of category code *letter*.

All these requirements (which are anyhow satisfied by default) are relaxed for the contents of an `\xintexpr`-ession: spaces are gobbled, catcodes mostly do not matter, the `e` of scientific notation may be `E` (on input) ...

19 Installation

```
A. Installation using xint.tds.zip:
-----------------------------------

obtain xint.tds.zip from CTAN:
  http://mirror.ctan.org/install/macros/generic/xint.tds.zip

cd to the download repertory and issue
  unzip xint.tds.zip -d <TEXMF>
for example: (assuming standard access rights, so sudo needed)
  sudo unzip xint.tds.zip -d /usr/local/texlive/texmf-local
  sudo mktexlsr
```

[39] exception: **xintexpr** needs the user to explicitely load **xintgcd**, resp. **xintbinhex**, if use is to be made in `\xintexpr` of the `lcm` and `gcd` functions, and, resp., hexadecimal numbers.

```
On Mac OS X, installation into user home folder:
  unzip xint.tds.zip -d  ~/Library/texmf

B. Installation after file extractions:
---------------------------------------

obtain xint.dtx, xint.ins and the README from CTAN:
   http://www.ctan.org/pkg/xint

- "tex xint.ins" generates the style files
(pre-existing files in the same repertory will be overwritten).

- without xint.ins: "tex or latex or pdflatex or xelatex xint.dtx"
will also generate the style files (and xint.ins).

xint.tex is also extracted, use it for the documentation:

- with latex+dvipdfmx: latex xint.tex thrice then dvipdfmx xint.dvi
Ignore dvipdfmx warnings, but if the pdf file has problems with fonts
(possibly from an old dvipdfmx), use then rather pdflatex or xelatex.

- with pdflatex or xelatex: run it directly thrice on xint.dtx, or run
it on xint.tex after having edited the suitable toggle therein.

When compiling xint.tex, the documentation is by default produced
with the source code included. See instructions in the file for
changing this default.

When compiling directly xint.dtx, the documentation is produced
without the source code (latex+dvips or pdflatex or xelatex).

Finishing the installation: (on first installation the destination
repertories may need to be created)

      xinttools.sty |
           xint.sty |
       xintfrac.sty |
       xintexpr.sty | --> TDS:tex/generic/xint/
     xintbinhex.sty |
        xintgcd.sty |
     xintseries.sty |
      xintcfrac.sty |

           xint.dtx   --> TDS:source/generic/xint/
           xint.ins   --> TDS:source/generic/xint/
           xint.tex   --> TDS:source/generic/xint/

           xint.pdf   --> TDS:doc/generic/xint/
             README   --> TDS:doc/generic/xint/

Depending on the TDS destination and the TeX installation, it may be
necessary to refresh the TeX installation filename database (mktexlsr)
```

20 The \xintexpr math parser (I)

Here is some random formula, defining a LaTeX command with three parameters,

```
\newcommand\formula[3]
{\xinttheexpr round((#1 & (#2 | #3)) * (355/113*#3 - (#1 - #2/2)^2), 8) \relax}
```

Let a=#1, b=#2, c=#3 be the parameters. The first term is the logical operation a and (b or c) where a number or fraction has truth value 1 if it is non-zero, and 0 otherwise. So here it means that a must be non-zero as well as b or c, for this first operand to be 1, else the formula returns 0. This multiplies a second term which is algebraic. Finally the result (where all intermediate computations are done *exactly*) is rounded to a value with 8 digits after the decimal mark, and printed.

\formula {771.3/9.1}{1.51e2}{37.73} expands to 32.81726043

Note that #1, #2, and #3 are not protected by parentheses in the definition of \formula, this is something to keep in mind if for example we want to use 2+5 as third argument: it should be (2+5) then.

- as everything gets expanded, the characters +,-,*,/,^,!,&,|,?,:,<,>,=,(,)," and the comma, which may appear in the infix syntax, should not (if actually used in the expression) be active (for example from serving as shorthands for some language in the Babel system). The command \xintexprSafeCatcodes resets these characters to their standard catcodes and \xintexprRestoreCatcodes restores the status prevailing at the time of the previous \xintexprSafeCatcodes.

- many expressions have equivalent macro formulations written without \xinttheexpr.[40] Here for \formula we could have used:

```
\xintRound {8}{\xintMul {\xintAND {#1}{\xintOR {#2}{#3}}}{\xintSub
{\xintMul {355/113}{#3}}{\xintPow {\xintSub {#1}{\xintDiv {#2}{2}}}{2}}}}
```

with the inherent difficulty of keeping up with braces and everything...

- if such a formula is used thousands of times in a document (for plots?), this could impact some parts of the TeX program memory (for technical reasons explained in section 26). So, a utility \xintNewExpr is provided to do the work of translating an \xintexpr-ession with parameters into a chain of macro evaluations.[41] With

```
\xintNewExpr\formula[3]
{ round((#1 & (#2 | #3)) * (355/113*#3 - (#1 - #2/2)^2), 8) }
```

one gets a command \formula with three parameters and meaning:

```
macro:#1#2#3->\romannumeral -`0\xintRound {\xintNum {8}}{\xintMul
{\xintAND {#1}{\xintOR {#2}{#3}}}{\xintSub {\xintMul {\xintDiv
{355}{113}}{#3}}{\xintPow {\xintSub {#1}{\xintDiv {#2}{2}}}{2}}}}
```

This does the same thing as the hand-written version from the previous item (but expands in only two steps).[42] The use even thousands of times of such an \xintNewExpr-generated \formula has no memory impact.

[40] Not everything allows a straightforward reformulation because the package macros only *f*-expand their arguments while \xintexpr expands everything from left to right. [41] As its makes some macro definitions, it is not an expandable command. It does not need protection against active characters as it does it itself. [42] But the hand-written version as well as the \xintNewExpr generated one differ from the original \formula command which allowed each of its argument to use all the operators and functions recognized by \xintexpr, and this aspect is lost. To recover it the arguments themselves should be passed as \xinttheexpr..\relax to the defined macro.

• count registers and \numexpr-essions are accepted (LaTeX's counters can be inserted using \value) without needing \the or \number as prefix. Also dimen registers and control sequences, skip registers and control sequences (LaTeX's lengths), \dimexpr-essions, \glueexpr-essions are automatically unpacked using \number, discarding the stretch and shrink components and giving the dimension value in sp units (1/65536th of a TeX point). Furthermore, tacit multiplication is implied, when the register, variable, or expression if immediately prefixed by a (decimal) number.

• tacit multiplication (the parser inserts a *) applies when the parser is currently scanning the digits of a number (or its decimal part), or is looking for an infix operator, and:
v1.09i → (1.) *encounters a register, variable or ε-TeX expression (as described in the previous item)*,
v1.09j → (2.) *encounters a sub-*\xintexpr*-ession*, or (3.) *encounters an opening parenthesis.*
v1.09k

• so far only \xinttheexpr was mentioned: there is also \xintexpr which, like a \numexpr, needs a prefix which is called \xintthe. Thus \xinttheexpr as was done in the definition of \formula is equivalent to \xintthe\xintexpr.

• This latter form is convenient when one has defined for example:
\def\x {\xintexpr \a + \b \relax} or \edef\x {\xintexpr \a+\b\relax}
One may then do \xintthe\x, either for printing the result on the page or use it in some other package macros. The \edef does the computation but keeps it in an internal private format. Naturally, the \edef is only possible if \a and \b are already defined.

• in both cases (the 'yet-to-be computed' and the 'already computed') \x can then be inserted in other expressions, as for example

\edef\y {\xintexpr \x^3\relax}

This would have worked also with \x defined as \def\x {(\a+\b)} but \edef\x would not have been an option then, and \x could have been used only inside an \xintexpr-ession, whereas the previous \x can also be used as \xintthe\x in any context triggering the expansion of \xintthe.

• sometimes one needs an integer, not a fraction or decimal number. The round function rounds to the nearest integer, and \xintexpr round(...)\relax has an alternative and equivalent syntax as \xintiexpr ... \relax. There is also \xinttheiexpr. The rounding is applied to the final result only, intermediate computations are not rounded.

• \xintiiexpr..\relax and \xinttheiiexpr..\relax deal only with (long) integers and skip the overhead of the fraction internal format. The infix operator / does euclidean division, thus 2+5/3 will not be treated exactly but be like 2+1.

• there is also \xintboolexpr ... \relax and \xinttheboolexpr ... \relax. Same as \xintexpr with the final result converted to 1 if it is not zero. See also \xintifboolexpr (subsection 26.11) and the discussion of the bool and togl functions in section 20. Here is an example:

```
\xintNewBoolExpr \AssertionA[3]{ #1 & (#2|#3) }
\xintNewBoolExpr \AssertionB[3]{ #1 | (#2&#3) }
\xintNewBoolExpr \AssertionC[3]{ xor(#1,#2,#3) }
\xintFor #1 in {0,1} \do {%
  \xintFor #2 in {0,1} \do {%
    \xintFor #3 in {0,1} \do {%
```

```
\centerline{#1 AND (#2 OR #3) is \AssertionA {#1}{#2}{#3}\hfil
            #1 OR (#2 AND #3) is \AssertionB {#1}{#2}{#3}\hfil
            #1 XOR #2 XOR #3  is \AssertionC {#1}{#2}{#3}}}}}
```

0 AND (0 OR 0) is 0	0 OR (0 AND 0) is 0	0 XOR 0 XOR 0 is 0
0 AND (0 OR 1) is 0	0 OR (0 AND 1) is 0	0 XOR 0 XOR 1 is 1
0 AND (1 OR 0) is 0	0 OR (1 AND 0) is 0	0 XOR 1 XOR 0 is 1
0 AND (1 OR 1) is 0	0 OR (1 AND 1) is 1	0 XOR 1 XOR 1 is 0
1 AND (0 OR 0) is 0	1 OR (0 AND 0) is 1	1 XOR 0 XOR 0 is 1
1 AND (0 OR 1) is 1	1 OR (0 AND 1) is 1	1 XOR 0 XOR 1 is 0
1 AND (1 OR 0) is 1	1 OR (1 AND 0) is 1	1 XOR 1 XOR 0 is 0
1 AND (1 OR 1) is 1	1 OR (1 AND 1) is 1	1 XOR 1 XOR 1 is 1

- there is also `\xintfloatexpr ... \relax` where the algebra is done in floating point approximation (also for each intermediate result). Use the syntax `\xintDigits:=N;` to set the precision. Default: 16 digits.

  `\xintthefloatexpr 2^100000\relax:` 9.990020930143845e30102

  The square-root operation can be used in `\xintexpr`, it is computed as a float with the precision set by `\xintDigits` or by the optional second argument:

  `\xinttheexpr sqrt(2,60)\relax:`

  141421356237309504880168872420969807856967187537694807317668[-59]

  Notice the `a/b[n]` notation: usually the denominator `b` even if `1` gets printed; it does not show here because the square root is computed by a version of `\xintFloatSqrt` which for efficiency when used in such expressions outputs the result in a format `d_1 d_2 .... d_P [N]` equivalent to the usual float output format `d_1.d_2...d_P e<expon.>`. To get a float format, it is easier to use an `\xintfloatexpr`, but the precision must be set using the non expandable `\xintDigits:=60;` assignment, there is no optional parameter possible currently to `\xintfloatexpr`:

  `\xintDigits:=60;\xintthefloatexpr sqrt(2)\relax`

  1.41421356237309504880168872420969807856967187537694807317668e0

  Or, without manipulating `\xintDigits`, another option to convert to float a computation done by an `\xintexpr`:

  `\xintFloat[60]{\xinttheexpr sqrt(2,60)\relax}`

  1.41421356237309504880168872420969807856967187537694807317668e0

  Floats are quickly indispensable when using the power function (which can only have an integer exponent), as exact results will easily have hundreds, if not thousands, of digits.

  `\xintDigits:=48; \xintthefloatexpr 2^100000\relax:`

  9.99002093014384507944032764330033590980429139054e30102

New with 1.09k! →

- hexadecimal TEX number denotations (*i.e.*, with a " prefix) are recognized by the `\xintexpr` parser and its variants. Except in `\xintiiexpr`, a (possibly empty) fractional part with the dot `.` as “hexadecimal” mark is allowed.

  `\xinttheexpr "FEDCBA9876543210\relax`→18364758544493064720

  `\xinttheiexpr 16^5-("F75DE.0A8B9+"8A21.F5746+16^-5)\relax`→0

  Letters must be uppercased, as with standard TEX hexadecimal denotations. Loading the **xintbinhex** package is required for this functionality.

21 The \xintexpr math parser (II)

An expression is built with infix operators (including comparison and boolean operators), parentheses, functions, and the two branching operators ? and :. The parser expands everything from the left to the right and everything may thus be revealed step by step by expansion of macros. Spaces anywhere are allowed.

Note that `2^-10` is perfectly accepted input, no need for parentheses; operators of power `^`, division `/`, and subtraction `-` are all left-associative: `2^4^8` is evaluated as `(2^4)^8`. The minus sign as prefix has various precedence levels: `\xintexpr -3-4*-5^-7\relax` evaluates as `(-3)-(4*(-(5^(-7))))` and `-3^-4*-5-7` as `(-((3^(-4))*(-5)))-7`.

If one uses directly macros within `\xintexpr..\relax`, rather than the operators or the functions which are described next, one should take into account that:

1. the parser will not see the macro arguments, (but they may themselves be set-up as `\xinttheexpr...\relax`),
2. the output format of most **xintfrac** macros is `A/B[N]`, and square brackets are *not understood by the parser*. One *must* enclose the macro and its arguments inside a brace pair `{..}`, which will be recognized and treated specially,
3. a macro outputting numbers in scientific notation `x.yEz` (either with a lowercase `e` or uppercase `E`), must *not* be enclosed in a brace pair, this is the exact opposite of the `A/B[N]` case; scientific numbers, explicit or implicit, should just be inserted directly in the expression.

One may insert a sub-`\xintexpr`-expression within a larger one. Each one of `\xintexpr`, `\xintiexpr`, `\xintfloatexpr`, `\xintboolexpr` may be inserted in another one. On the other hand the integer only `\xintiiexpr` will generally choke on a sub-`\xintexpr` as the latter (except if it did not do any operation or had an overall top level `round` or `trunc` or `?(..)` or...) produces (in internal format) an `A/B[N]` which the strictly integer only `\xintiiexpr` does not understand. See subsection 26.8 for more information.

Here is, listed from the highest priority to the lowest, the complete list of operators and functions. Functions are at the top level of priority. Next are the postfix operators: `!` for the factorial, ? and `:` are two-fold way and three-fold way branching constructs. Also at the top level of priority the `e` and `E` of the scientific notation and the `"` for hexadecimal numbers, then power, multiplication/division, addition/subtraction, comparison, and logical operators. At the lowest level: commas then parentheses.

" is new in 1.09k →

The `\relax` at the end of an expression is *mandatory*.

- Functions are at the same top level of priority. All functions even `?` and `!` (as prefix) require parentheses around their argument (possibly a comma separated list).

> **floor, ceil, frac, reduce, sqr, abs, sgn, ?, !, not, bool, togl, round, trunc, float, sqrt, quo, rem, if, ifsgn, all, any, xor, add (=sum), mul (=prd), max, min, gcd, lcm.**
>
> `quo` and `rem` operate only on integers; `gcd` and `lcm` also and require **xintgcd** loaded; `togl` requires the `etoolbox` package; `all`, `any`, `xor`, `add`, `mul`, `max` and `min` are functions with arbitrarily many comma separated arguments.

functions with one (numeric) argument (numeric: any expression leading to an integer, decimal number, fraction, or floating number in scientific notation) `floor, ceil, frac, reduce, sqr, abs, sgn, ?, !, not`. The `?(x)` function returns the truth value, `1` if `x<>0`, `0` if `x=0`. The `!(x)` is the logical not. The `reduce` function puts the fraction in irreducible form. The `frac` function is fractional part, same sign as the number:

```
\xinttheexpr frac(-3.57)\relax→-57/1[-2]
\xinttheexpr trunc(frac(-3.57),2)\relax→-0.57
\xintthefloatexpr frac(-3.57)\relax→-5.700000000000000e-1.
```

Like the other functions `!` and `?` *must* use parentheses.

functions with one (alphabetical) argument `bool,togl`. `bool(name)` returns `1` if the TeX conditional `\ifname` would act as `\iftrue` and `0` otherwise. This works with conditionals defined by `\newif` (in TeX or LaTeX) or with primitive conditionals such as `\ifmmode`. For example:

```
\xintifboolexpr{25*4-if(bool(mmode),100,75)}{YES}{NO}
```

will return *NO* if executed in math mode (the computation is then $100 - 100 = 0$) and YES if not (the `if` conditional is described below; the `\xintifboolexpr` test automatically encapsulates its first argument in an `\xintexpr` and follows the first branch if the result is non-zero (see subsection 26.11)).

The alternative syntax `25*4-\ifmmode100\else75\fi` could have been used here, the usefulness of `bool(name)` lies in the availability in the `\xintexpr` syntax of the logic operators of conjunction `&`, inclusive disjunction `|`, negation `!` (or `not`), of the multi-operands functions `all`, `any`, `xor`, of the two branching operators `if` and `ifsgn` (see also `?` and `:`), which allow arbitrarily complicated combinations of various `bool(name)`.

Similarly `togl(name)` returns `1` if the LaTeX package etoolbox[43] has been used to define a toggle named `name`, and this toggle is currently set to `true`. Using `togl` in an `\xintexpr..\relax` without having loaded etoolbox will result in an error from `\iftoggle` being a non-defined macro. If `etoolbox` is loaded but `togl` is used on a name not recognized by `etoolbox` the error message will be of the type “ERROR: Missing `\endcsname` inserted.”, with further information saying that `\protect` should have not been encountered (this `\protect` comes from the expansion of the non-expandable `etoolbox` error message).

When `bool` or `togl` is encountered by the `\xintexpr` parser, the argument enclosed in a parenthesis pair is expanded as usual from left to right, token by token, until the closing parenthesis is found, but everything is taken literally, no computations are performed. For example `togl(2+3)` will test the value of a toggle declared to `etoolbox` with name `2+3`, and not `5`. Spaces are gobbled in this process. It is impossible to use `togl` on such names containing spaces, but `\iftoggle{name with spaces}{1}{0}` will work, naturally, as its expansion will pre-empt the `\xintexpr` scanner.

There isn’t in `\xintexpr...` a `test` function available analogous to the `test{\ifsometest}` construct from the `etoolbox` package; but any *expandable* `\ifsometest` can be inserted directly in an `\xintexpr`-ession as `\ifsometest10` (or `\ifsometest{1}{0}`), for example `if(\ifsometest{1}{0},YES,NO)` (see the `if` operator below) works.

43 http://www.ctan.org/pkg/etoolbox

A straight `\ifsometest{YES}{NO}` would do the same more efficiently, the point of `\ifsometest10` is to allow arbitrary boolean combinations using the (described later) `&` and `|` logic operators: `\ifsometest10 & \ifsomeothertest10 | \ifsomethirdtest10`, etc... `YES` or `NO` above stand for material compatible with the `\xintexpr` parser syntax.

See also `\xintifboolexpr`, in this context.

functions with one mandatory and a second optional argument `round, trunc, float, sqrt`. For example `round(2^9/3^5,12)=2.106995884774.` The `sqrt` is available also in `\xintexpr`, not only in `\xintfloatexpr`. The second optional argument is the required float precision.

functions with two arguments `quo, rem`. These functions are integer only, they give the quotient and remainder in Euclidean division (more generally one can use the `floor` function; related: the `frac` function).

the if conditional (twofold way) `if(cond,yes,no)` checks if `cond` is true or false and takes the corresponding branch. Any non zero number or fraction is logical true. The zero value is logical false. Both "branches" are evaluated (they are not really branches but just numbers). See also the `?` operator.

the ifsgn conditional (threefold way) `ifsgn(cond,<0,=0,>0)` checks the sign of `cond` and proceeds correspondingly. All three are evaluated. See also the `:` operator.

functions with an arbitrary number of arguments `all, any, xor, add (=sum), mul (=prd), max, min, gcd, lcm`: `gcd` and `lcm` are integer-only and require the **xintgcd** package. Currently, the `and` and `or` keywords are left undefined by the package, which uses rather `all` and `any`. They must have at least one argument.

- The three postfix operators `!, ?, :`.

`!` computes the factorial of an integer. `sqrt(36)!` evaluates to `6!` (`=720`) and not to the square root of `36!` ($\approx$`6.099,125,566,750,542` $\times 10^{20}$). This is the exact factorial even when used inside `\xintfloatexpr`.

`?` is used as `(cond)?{yes}{no}`. It evaluates the (numerical) condition (any non-zero value counts as `true`, zero counts as `false`). It then acts as a macro with two mandatory arguments within braces (hence this escapes from the parser scope, the braces can not be hidden in a macro), chooses the correct branch *without evaluating the wrong one*. Once the braces are removed, the parser scans and expands the uncovered material so for example

```
\xinttheiexpr (3>2)?{5+6}{7-1}2^3\relax
```

is legal and computes `5+62^3=238333`. Note though that it would be better practice to include here the `2^3` inside the branches. The contents of the branches may be arbitrary as long as once glued to what is next the syntax is respected: `\xintexpr (3>2)?{5+(6}{7-(1}2^3)\relax` also works. Differs thus from the `if` conditional in two ways: the false branch is not at all computed, and the number scanner is still active on exit, more digits may follow.

`:` is used as `(cond):{<0}{=0}{>0}`. `cond` is anything, its sign is evaluated (it is not necessary to use `sgn(cond):{<}{=}{>}`) and depending on the sign the correct branch is un-braced, the two others are swallowed. The un-braced branch will then be parsed as usual. Differs from the `ifsgn` conditional as the two false branches are not evaluated and furthermore the number scanner is still active on exit.

```
\def\x{0.33}\def\y{1/3}
\xinttheexpr (\x-\y):{sqrt}{0}{1/}(\y-\x)\relax=5773502691896258[-17]
```

- The `.` as decimal mark; the number scanner treats it as an inherent, optional and unique component of a being formed number. One can do things such as `\xinttheexpr .^2+2^. \relax`→`1/1[0]` (which is `0^2+2^0`).

- The `"` for hexadecimal numbers: it is treated with highest priority, allowed only at locations where the parser expects to start forming a numeric operand, once encountered it triggers the hexadecimal scanner which looks for successive hexadecimal digits (as usual skipping spaces and expanding forward everything) possibly a unique optional dot (allowed directly in front) and then an optional (possibly empty) fractional part. The dot and fractional part are not allowed in `\xintiiexpr..\relax`. The `"` functionality requires that the user loaded **xintbinhex** (there is no warning, but an "undefined control sequence" error will naturally results if the package has not been loaded).

- The `e` and `E` for scientific notation. They are treated as infix operators of highest priority: this means that they serve as an end marker (possibly arising from macro expansion) for the scanned number, and then will pre-empt the number coming next, either explicit, or arising from expansion, from parenthesized material, from a sub-expression etc..., to serve as exponent. From the rules above, inside `\xintexpr`, `1e3-1` is `999/1[0]`, `1e3^2` is `1/1[6]`, and `"Ae("A+"F)^"A` is `10000000000/1[250]`.

- The power operator `^`. It is left associative: `\xinttheiexpr 2^2^3\relax` evaluates to 64, not 256. Note that if the float precision is too low, iterated powers withing `\xintfloatexpr..\relax` may fail: for example with the default setting `(1+1e-8)^(12^16)` will be computed with `12^16` approximated from its `16` most significant digits but it has `18` digits (`=184884258895036416`), hence the result is wrong:
$$1.879{,}985{,}375{,}897{,}266\times 10^{802{,}942{,}130}$$
One should code
```
\xintthe\xintfloatexpr (1+1e-8)^\xintiiexpr 12^20\relax \relax
```
to obtain the correct floating point evaluation
$$1.000{,}000{,}01^{12^{16}}\approx 1.879{,}985{,}676{,}694{,}948\times 10^{802{,}942{,}130}$$

- Multiplication and division `*`, `/`. The division is left associative, too: `\xinttheiexpr 100/50/2\relax` evaluates to 1, not 4.

- Addition and subtraction `+`, `-`. Again, `-` is left associative: `\xinttheiexpr 100-50-2 \relax` evaluates to 48, not 52.

- Comparison operators <, >, = (currently, no <=, >=, ...).

- Conjunction (logical and): `&`. (no `&&`)

- Inclusive disjunction (logical or): `|`. (no `||`)

- The comma `,`. With `\xinttheiexpr 2^3, 3^4, 5^6\relax` one obtains as output 8,81,15625 (no space after the commas on output).

- The parentheses.

See subsection 26.2 for count and dimen registers and variables.

22 Change log for earlier releases

Release 1.09j ([2014/01/09]):

- the core division routines have been re-written for some (limited) efficiency gain, more pronounced for small divisors. As a result the computation of one thousand digits of π is close to three times faster than with earlier releases.
- some various other small improvements, particularly in the power routines.
- a new macro \xintXTrunc is designed to produce thousands or even tens of thousands of digits of the decimal expansion of a fraction. Although completely expandable it has its use limited to inside an \edef, \write, \message, ... It can thus not be nested as argument to another package macro.
- the tacit multiplication done in \xintexpr..\relax on encountering a count register or variable, or a \numexpr, while scanning a (decimal) number, is extended to the case of a sub \xintexpr-ession.
- \xintexpr can now be used in an \edef with no \xintthe prefix; it will execute completely the computation, and the error message about a missing \xintthe will be inhibited. Previously, in the absence of \xintthe, expansion could only be a full one (with \romannumeral-`0), not a complete one (with \edef). Note that this differs from the behavior of the non-expandable \numexpr: \the or \number are needed not only to print but also to trigger the computation, whereas \xintthe is mandatory only for the printing step.
- the default behavior of \xintAssign is changed, it now does not do any further expansion beyond the initial full-expansion which provided the list of items to be assigned to macros.
- bug-fix: 1.09i did an unexplainable change to \XINT_infloat_zero which broke the floating point routines for vanishing operands =:(((
- dtx bug-fix: the 1.09i .ins file produced a buggy .tex file.

Release 1.09i ([2013/12/18]):

- \xintiiexpr is a variant of \xintexpr which is optimized to deal only with (long) integers, / does a euclidean quotient.
- \xintnumexpr, \xintthenumexpr, \xintNewNumExpr are renamed, respectively, \xintiexpr, \xinttheiexpr, \xintNewIExpr. The earlier denominations are kept but to be removed at some point.
- it is now possible within \xintexpr...\relax and its variants to use count, dimen, and skip registers or variables without explicit \the/\number: the parser inserts automatically \number and a tacit multiplication is implied when a register or variable immediately follows a number or fraction. Regarding dimensions and \number, see the further discussion in subsection 9.2.
- new conditional \xintifOne; \xintifTrueFalse renamed to \xintifTrueAelseB; new macros \xintTFrac ('fractional part', mapped to function frac in \xintexpr-essions), \xintFloatE.
- \xintAssign admits an optional argument to specify the expansion type to be used: [] (none, default), [o] (once), [oo] (twice), [f] (full), [e] (\edef),... to define the macros
- related to the previous item, **xinttools** defines \odef, \oodef, \fdef (if the names have already been assigned, it uses \xintoodef etc...). These tools are provided for the case one uses the package macros in a non-expandable context, particularly \oodef which expands twice the macro replacement text and is thus a faster alternative to \edef taking into account that the **xint** bundle macros expand already completely in only two steps. This can be significant when repeatedly making \def-initions expanding to hundreds of digits.
- some across the board slight efficiency improvement as a result of modifications of various types to "fork" macros and "branching conditionals" which are used internally.
- bug-fix: \xintAND and \xintOR inserted a space token in some cases and did not expand as promised in two steps (bug dating back to 1.09a I think; this bug was without consequences when using & and | in \xintexpr-essions, it affected only the macro form) :-((.

- bug-fix: \xintFtoCCv still ended fractions with the [0]'s which were supposed to have been removed since release 1.09b.

Release 1.09h ([2013/11/28]):

- parts of the documentation have been re-written or re-organized, particularly the discussion of expansion issues and of input and output formats.
- the expansion types of macro arguments are documented in the margin of the macro descriptions, with conventions mainly taken over from those in the LaTeX3 documentation.
- a dependency of **xinttools** on **xint** (inside \xintSeq) has been removed.
- \xintTypesetEuclideAlgorithm and \xintTypesetBezoutAlgorithm have been slightly modified (regarding indentation).
- macros \xintiSum and \xintiPrd are renamed to \xintiiSum and \xintiiPrd.
- a count register used in 1.09g in the \xintFor loops for parsing purposes has been removed and replaced by use of a \numexpr.
- the few uses of \loop have been replaced by \xintloop/\xintiloop.
- all macros of **xinttools** for which it makes sense are now declared \long.

Release 1.09g ([2013/11/22]):

- package **xinttools** is detached from **xint**, to make tools such as \xintFor, \xintApplyUnbraced, and \xintiloop available without the **xint** overhead.
- new expandable nestable loops \xintloop and \xintiloop.
- bugfix: \xintFor and \xintFor* do not modify anymore the value of \count 255.

Release 1.09f ([2013/11/04]):

- new \xintZapFirstSpaces, \xintZapLastSpaces, \xintZapSpaces, \xintZapSpacesB, for expandably stripping away leading and/or ending spaces.
- \xintCSVtoList by default uses \xintZapSpacesB to strip away spaces around commas (or at the start and end of the comma separated list).
- also the \xintFor loop will strip out all spaces around commas and at the start and the end of its list argument; and similarly for \xintForpair, \xintForthree, \xintForfour.
- \xintFor *et al.* accept all macro parameters from #1 to #9.
- for reasons of inner coherence some macros previously with one extra 'i' in their names (e.g. \xintiMON) now have a doubled 'ii' (\xintiiMON) to indicate that they skip the overhead of parsing their inputs via \xintNum. Macros with a *single* 'i' such as \xintiAdd are those which maintain the non-**xintfrac** output format for big integers, but do parse their inputs via \xintNum (since release 1.09a). They too may have doubled-i variants for matters of programming optimization when working only with (big) integers and not fractions or decimal numbers.

Release 1.09e ([2013/10/29]):

- new \xintintegers, \xintdimensions, \xintrationals for infinite \xintFor loops, interrupted with \xintBreakFor and \xintBreakForAndDo.
- new \xintifForFirst, \xintifForLast for the \xintFor and \xintFor* loops,
- the \xintFor and \xintFor* loops are now \long, the replacement text and the items may contain explicit \par's.
- bug fix, the \xintFor loop (not \xintFor*) did not correctly detect an empty list.
- new conditionals \xintifCmp, \xintifInt, \xintifOdd.
- bug fix, \xintiSqrt {0} crashed. :-((
- the documentation has been enriched with various additional examples, such as the the quick sort algorithm illustrated or the computation of prime numbers (subsection 23.11, subsection 23.14, subsection 23.21).
- the documentation explains with more details various expansion related issues, particularly in relation to conditionals.

Release 1.09d ([2013/10/22]):

- \xintFor* is modified to gracefully handle a space token (or more than one) located at the very end of its list argument (as in for example \xintFor* #1 in {{a}{b}{c}<space>} \do {stuff}; spaces at other locations were already harmless). Furthermore this new version f-expands the un-braced list items. After \def\x{{1}{2}} and \def\y{{a}\x {b}{c}\x }, \y will appear to \xintFor* exactly as if it had been defined as \def\y{{a}{1}{2}{b}{c}{1}{2}}.
- same bug fix in \xintApplyInline.

Release 1.09c ([2013/10/09]):

- added bool and togl to the \xintexpr syntax; also added \xintboolexpr and \xintifboolexpr.
- added \xintNewNumExpr (now \xintNewIExpr and \xintNewBoolExpr,
- \xintFor is a new type of loop, whose replacement text inserts the comma separated values or list items via macro parameters, rather than encapsulated in macros; the loops are nestable up to four levels (nine levels since 1.09f) and their replacement texts are allowed to close groups as happens with the tabulation in alignments,
- \xintForpair, \xintForthree, \xintForfour are experimental variants of \xintFor,
- \xintApplyInline has been enhanced in order to be usable for generating rows (partially or completely) in an alignment,
- new command \xintSeq to generate (expandably) arithmetic sequences of (short) integers,
- the factorial ! and branching ?, :, operators (in \xintexpr...\relax) have now less precedence than a function name located just before: func(x)! is the factorial of func(x), not func(x!),
- again various improvements and changes in the documentation.

Release 1.09b ([2013/10/03]):

- various improvements in the documentation,
- more economical catcode management and re-loading handling,
- removal of all those [0]'s previously forcefully added at the end of fractions by various macros of **xintcfrac**,
- \xintNthElt with a negative index returns from the tail of the list,
- new macro \xintPRaw to have something like what \xintFrac does in math mode; i.e. a \xintRaw which does not print the denominator if it is one.

Release 1.09a ([2013/09/24]):

- \xintexpr..\relax and \xintfloatexpr..\relax admit functions in their syntax, with comma separated values as arguments, among them reduce, sqr, sqrt, abs, sgn, floor, ceil, quo, rem, round, trunc, float, gcd, lcm, max, min, sum, prd, add, mul, not, all, any, xor.
- comparison (<, >, =) and logical (|, &) operators.
- the command \xintthe which converts \xintexpressions into printable format (like \the with \numexpr) is more efficient, for example one can do \xintthe\x if \x was def'ined to be an \xintexpr..\relax:

```
\def\x{\xintexpr 3^57\relax}\def\y{\xintexpr \x^(-2)\relax}
\def\z{\xintexpr \y-3^-114\relax}        \xintthe\z=0/1[0]
```

- \xintnumexpr .. \relax (now renamed \xintiexpr) is \xintexpr round(..) \relax.
- \xintNewExpr now works with the standard macro parameter character #.
- both regular \xintexpr-essions and commands defined by \xintNewExpr will work with comma separated lists of expressions,
- new commands \xintFloor, \xintCeil, \xintMaxof, \xintMinof (package **xintfrac**), \xintGCDof, \xintLCM, \xintLCMof (package **xintgcd**), \xintifLt, \xintifGt, \xintifSgn, \xintANDof, ...
- The arithmetic macros from package **xint** now filter their operands via \xintNum which means that they may use directly count registers and \numexpr-essions without having to prefix them by \the. This is thus similar to the situation holding previously but with **xintfrac** loaded.

- a bug introduced in 1.08b made \xintCmp crash when one of its arguments was zero. :-((

Release 1.08b ([2013/06/14]):

- Correction of a problem with spaces inside \xintexpr-essions.
- Additional improvements to the handling of floating point numbers.
- The macros of **xintfrac** allow to use count registers in their arguments in ways which were not previously documented. See Use of count registers.

Release 1.08a ([2013/06/11]):

- Improved efficiency of the basic conversion from exact fractions to floating point numbers, with ensuing speed gains especially for the power function macros \xintFloatPow and \xintFloatPower,
- Better management by the **xintfrac** macros \xintCmp, \xintMax, \xintMin and \xintGeq of inputs having big powers of ten in them.
- Macros for floating point numbers added to the **xintseries** package.

Release 1.08 ([2013/06/07]):

- Extraction of square roots, for floating point numbers (\xintFloatSqrt), and also in a version adapted to integers (\xintiSqrt).
- New package **xintbinhex** providing conversion routines to and from binary and hexadecimal bases.

Release 1.07 ([2013/05/25)]):

- The **xintfrac** macros accept numbers written in scientific notation, the \xintFloat command serves to output its argument with a given number D of significant figures. The value of D is either given as optional argument to \xintFloat or set with \xintDigits := D;. The default value is 16.
- The **xintexpr** package is a new core constituent (which loads automatically **xintfrac** and **xint**) and implements the expandable expanding parsers
  \xintexpr . . . \relax, and its variant \xintfloatexpr . . . \relax
  allowing on input formulas using the standard form with infix operators +, -, *, /, and ^, and arbitrary levels of parenthesizing. Within a float expression the operations are executed according to the current value of \xintDigits. Within an \xintexpr-ession the binary operators are computed exactly.
- The floating point precision D is set (this is a local assignment to a \mathchar variable) with \xintDigits := D; and queried with \xinttheDigits. It may be set to anything up to 32767.[44] The macro incarnations of the binary operations admit an optional argument which will replace pointwise D; this argument may exceed the 32767 bound.
- To write the \xintexpr parser I benefited from the commented source of the LaTeX3 parser; the \xintexpr parser has its own features and peculiarities. See its documentation.

Initial release 1.0 was on 2013/03/28.

23 Commands of the **xinttools** package

These utilities used to be provided within the **xint** package; since 1.09g they have been moved to an independently usable package **xinttools**, which has none of the **xint** facilities regarding big numbers. Whenever relevant release 1.09h has made the macros \long so they accept \par tokens on input.

First the completely expandable utilities up to \xintiloop, then the non expandable utilities.

This section contains various concrete examples and ends with a completely expandable implementation of the Quick Sort algorithm together with a graphical illustration of its action.

[44] but values higher than 100 or 200 will presumably give too slow evaluations.

Contents

23.1 \xintReverseOrder

$n\star$ \xintReverseOrder{⟨*list*⟩} does not do any expansion of its argument and just reverses the order of the tokens in the ⟨*list*⟩. Braces are removed once and the enclosed material, now unbraced, does not get reverted. Unprotected spaces (of any character code) are gobbled.

```
\xintReverseOrder{\xintDigitsOf\xintiPow {2}{100}\to\Stuff}
```

gives: `\Stuff\to1002\xintiPow\xintDigitsOf`

23.2 \xintRevWithBraces

$f\star$ \xintRevWithBraces{⟨*list*⟩} first does the f-expansion of its argument then it reverses the order of the tokens, or braced material, it encounters, adding a pair of braces to each (thus, maintaining brace pairs already existing). Spaces (in-between external brace pairs) are gobbled. This macro is mainly thought out for use on a ⟨*list*⟩ of such braced material; with such a list as argument the f-expansion will only hit against the first opening brace, hence do nothing, and the braced stuff may thus be macros one does not want to expand.

```
\edef\x{\xintRevWithBraces{12345}}
\meaning\x:macro:->{5}{4}{3}{2}{1}
\edef\y{\xintRevWithBraces\x}
\meaning\y:macro:->{1}{2}{3}{4}{5}
```

The examples above could be defined with \edef's because the braced material did not contain macros. Alternatively:

```
\expandafter\def\expandafter\w\expandafter
{\romannumeral0\xintrevwithbraces{{\A}{\B}{\C}{\D}{\E}}}
\meaning\w:macro:->{\E }{\D }{\C }{\B }{\A }
```

n ★ The macro `\xintReverseWithBracesNoExpand` does the same job without the initial expansion of its argument.

23.3 `\xintLength`

n ★ `\xintLength{⟨list⟩}` does not do *any* expansion of its argument and just counts how many tokens there are (possibly none). So to use it to count things in the replacement text of a macro one should do `\expandafter\xintLength\expandafter{\x}`. One may also use it inside macros as `\xintLength{#1}`. Things enclosed in braces count as one. Blanks between tokens are not counted. See `\xintNthElt{0}` for a variant which first *f*-expands its argument.

```
\xintLength {\xintiPow {2}{100}}=3
≠ \xintLen {\xintiPow {2}{100}}=31
```

23.4 `\xintZapFirstSpaces`, `\xintZapLastSpaces`, `\xintZapSpaces`, `\xintZapSpacesB`

n ★ `\xintZapFirstSpaces{⟨stuff⟩}` does not do *any* expansion of its argument, nor brace removal of any sort, nor does it alter ⟨*stuff*⟩ in anyway apart from stripping away all *leading* spaces.

This macro will be mostly of interest to programmers who will know what I will now be talking about. *The essential points, naturally, are the complete expandability and the fact that no brace removal nor any other alteration is done to the input.*

TeX's input scanner already converts consecutive blanks into single space tokens, but `\xintZapFirstSpaces` handles successfully also inputs with consecutive multiple space tokens. However, it is assumed that ⟨*stuff*⟩ does not contain (except inside braced sub-material) space tokens of character code distinct from 32.

It expands in two steps, and if the goal is to apply it to the expansion text of `\x` to define `\y`, then one should do: `\expandafter\def\expandafter\y\expandafter {\romannumeral0\expandafter\xintzapfirstspaces\expandafter{\x}}`.

Other use case: inside a macro as `\edef\x{\xintZapFirstSpaces {#1}}` assuming naturally that `#1` is compatible with such an `\edef` once the leading spaces have been stripped.

```
\xintZapFirstSpaces { \a { \X } { \b \Y } }->\a { \X } { \b \Y } +++
```

n ★ `\xintZapLastSpaces{⟨stuff⟩}` does not do *any* expansion of its argument, nor brace removal of any sort, nor does it alter ⟨*stuff*⟩ in anyway apart from stripping away all *ending* spaces. The same remarks as for `\xintZapFirstSpaces` apply.

```
\xintZapLastSpaces { \a { \X } { \b \Y } }-> \a { \X } { \b \Y }+++
```

n ★ `\xintZapSpaces{⟨stuff⟩}` does not do *any* expansion of its argument, nor brace removal of any sort, nor does it alter ⟨*stuff*⟩ in anyway apart from stripping away all *leading* and all *ending* spaces. The same remarks as for `\xintZapFirstSpaces` apply.

```
\xintZapSpaces { \a { \X } { \b \Y } }->\a { \X } { \b \Y }+++
```

n ★ `\xintZapSpacesB{⟨stuff⟩}` does not do *any* expansion of its argument, nor does it alter

⟨*stuff*⟩ in anyway apart from stripping away all leading and all ending spaces and possibly removing one level of braces if ⟨*stuff*⟩ had the shape `<spaces>{braced}<spaces>`. The same remarks as for `\xintZapFirstSpaces` apply.

`\xintZapSpacesB { \a { \X } { \b \Y } }->\a { \X } { \b \Y }+++`
`\xintZapSpacesB { { \a { \X } { \b \Y } } }-> \a { \X } { \b \Y } +++`

The spaces here at the start and end of the output come from the braced material, and are not removed (one would need a second application for that; recall though that the **xint** zapping macros do not expand their argument).

23.5 `\xintCSVtoList`

f ★ `\xintCSVtoList{a,b,c...,z}` returns `{a}{b}{c}...{z}`. A *list* is by convention in this manual simply a succession of tokens, where each braced thing will count as one item ("items" are defined according to the rules of TeX for fetching undelimited parameters of a macro, which are exactly the same rules as for LaTeX and command arguments [they are the same things]). The word 'list' in 'comma separated list of items' has its usual linguistic meaning, and then an "item" is what is delimited by commas.

So `\xintCSVtoList` takes on input a 'comma separated list of items' and converts it into a 'TeX list of braced items'. The argument to `\xintCSVtoList` may be a macro: it will first be *f*-expanded. Hence the item before the first comma, if it is itself a macro, will be expanded which may or may not be a good thing. A space inserted at the start of the first
n ★ item serves to stop that expansion (and disappears). The macro `\xintCSVtoListNoExpand` does the same job without the initial expansion of the list argument.

Apart from that no expansion of the items is done and the list items may thus be completely arbitrary (and even contain perilous stuff such as unmatched `\if` and `\fi` tokens).

Contiguous spaces and tab characters, are collapsed by TeX into single spaces. All such spaces around commas[45] $\boxed{\text{are removed}}$, as well as the spaces at the start and the spaces at the end of the list.[46] The items may contain explicit `\par`'s or empty lines (converted by the TeX input parsing into `\par` tokens).

```
\xintCSVtoList { 1 ,{ 2 , 3 , 4 , 5 }, a , {b,T} U , { c , d } , { {x , y} } }
        ->{1}{2 , 3 , 4 , 5}{a}{{b,T} U}{ c , d }{ {x , y} }
```

One sees on this example how braces protect commas from sub-lists to be perceived as delimiters of the top list. Braces around an entire item are removed, even when surrounded by spaces before and/or after. Braces for sub-parts of an item are not removed.

We observe also that there is a slight difference regarding the brace stripping of an item: if the braces were not surrounded by spaces, also the initial and final (but no other) spaces of the *enclosed* material are removed. This is the only situation where spaces protected by braces are nevertheless removed.

From the rules above: for an empty argument (only spaces, no braces, no comma) the output is `{}` (a list with one empty item), for "`<opt. spaces>{}<opt. spaces>`" the output is `{}` (again a list with one empty item, the braces were removed), for "`{ }`" the output is `{}` (again a list with one empty item, the braces were removed and then the inner

[45] and multiple space tokens are not a problem; but those at the top level (not hidden inside braces) *must* be of character code 32. [46] let us recall that this is all done completely expandably... There is absolutely no alteration of any sort of the item apart from the stripping of initial and final space tokens (of character code 32) and brace removal if and only if the item apart from intial and final spaces (or more generally multiple `char 32` space tokens) is braced.

space was removed), for "` { }`" the output is `{}` (again a list with one empty item, the initial space served only to stop the expansion, so this was like "`{ }`" as input, the braces were removed and the inner space was stripped), for "` {  } `" the output is `{ }` (this time the ending space of the first item meant that after brace removal the inner spaces were kept; recall though that TeX collapses on input consecutive blanks into one space token), for "`,`" the output consists of two consecutive empty items `{}{}`. Recall that on output everything is braced, a `{}` is an "empty" item. Most of the above is mainly irrelevant for every day use, apart perhaps from the fact to be noted that an empty input does not give an empty output but a one-empty-item list (it is as if an ending comma was always added at the end of the input).

```
\def\y{ \a,\b,\c,\d,\e} \xintCSVtoList\y->{\a }{\b }{\c }{\d }{\e }
\def\t {{\if},\ifnum,\ifx,\ifdim,\ifcat,\ifmmode}
\xintCSVtoList\t->{\if }{\ifnum }{\ifx }{\ifdim }{\ifcat }{\ifmmode }
```

The results above were automatically displayed using TeX's primitive `\meaning`, which adds a space after each control sequence name. These spaces are not in the actual braced items of the produced lists. The first items `\a` and `\if` were either preceded by a space or braced to prevent expansion. The macro `\xintCSVtoListNoExpand` would have done the same job without the initial expansion of the list argument, hence no need for such protection but if `\y` is defined as `\def\y{\a,\b,\c,\d,\e}` we then must do:

```
\expandafter\xintCSVtoListNoExpand\expandafter {\y}
```

Else, we may have direct use:

```
\xintCSVtoListNoExpand {\if,\ifnum,\ifx,\ifdim,\ifcat,\ifmmode}
    ->{\if }{\ifnum }{\ifx }{\ifdim }{\ifcat }{\ifmmode }
```

Again these spaces are an artefact from the use in the source of the document of `\meaning` (or rather here, `\detokenize`) to display the result of using `\xintCSVtoListNoExpand` (which is done for real in this document source).

For the similar conversion from comma separated list to braced items list, but with-
$f\star$ out removal of spaces around the commas, there is `\xintCSVtoListNonStripped` and
$n\star$ `\xintCSVtoListNonStrippedNoExpand`.

23.6 \xintNthElt

$\overset{\text{num}}{x} f\star$ `\xintNthElt{x}{⟨list⟩}` gets (expandably) the xth braced item of the ⟨*list*⟩. An unbraced item token will be returned as is. The list itself may be a macro which is first *f*-expanded.

`\xintNthElt {3}{{agh}\u{zzz}\v{Z}}` is `zzz`

`\xintNthElt {3}{{agh}\u{{zzz}}\v{Z}}` is `{zzz}`

`\xintNthElt {2}{{agh}\u{{zzz}}\v{Z}}` is `\u`

`\xintNthElt {37}{\xintFac {100}}=9` is the thirty-seventh digit of `100!`.

`\xintNthElt {10}{\xintFtoCv {566827/208524}}=1457/536`

is the tenth convergent of `566827/208524` (uses **xintcfrac** package).

```
\xintNthElt {7}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=7
\xintNthElt {0}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=9
\xintNthElt {-3}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=7
```

If `x=0`, the macro returns the *length* of the expanded list: this is not equivalent to `\xintLength` which does no pre-expansion. And it is different from `\xintLen` which is to be used only on integers or fractions.

If `x<0`, the macro returns the xth element from the end of the list.

\xintNthElt {-5}{{{agh}}\u{zzz}\v{Z}} is {agh}

num x n ★ The macro \xintNthEltNoExpand does the same job but without first expanding the list argument: \xintNthEltNoExpand {-4}{\u\v\w T\x\y\z} is T.

In cases where x is larger (in absolute value) than the length of the list then \xintNthElt returns nothing.

23.7 \xintListWithSep

nf ★ \xintListWithSep{sep}{⟨*list*⟩} inserts the given separator sep in-between all items of the given list of braced items: this separator may be a macro (or multiple tokens) but will not be expanded. The second argument also may be itself a macro: it is *f*-expanded. Applying \xintListWithSep removes the braces from the list items (for example {1}{2}{3} turns into 1,2,3 via \xintListWithSep{,}{{1}{2}{3}}). An empty input gives an empty output, a singleton gives a singleton, the separator is used starting with at least two elements. Using an empty separator has the net effect of unbracing the braced items constituting the ⟨*list*⟩ (in such cases the new list may thus be longer than the original).

\xintListWithSep{:}{\xintFac {20}}=2:4:3:2:9:0:2:0:0:8:1:7:6:6:4:0:0:0:0

nn ★ The macro \xintListWithSepNoExpand does the same job without the initial expansion.

23.8 \xintApply

ff ★ \xintApply{\macro}{⟨*list*⟩} expandably applies the one parameter command \macro to each item in the ⟨*list*⟩ given as second argument and returns a new list with these outputs: each item is given one after the other as parameter to \macro which is expanded at that time (as usual, *i.e.* fully for what comes first), the results are braced and output together as a succession of braced items (if \macro is defined to start with a space, the space will be gobbled and the \macro will not be expanded; it is allowed to have its own arguments, the list items serve as last arguments to \macro). Hence \xintApply{\macro}{{1}{2}{3}} returns {\macro{1}}{\macro{2}}{\macro{3}} where all instances of \macro have been already *f*-expanded.

Being expandable, \xintApply is useful for example inside alignments where implicit groups make standard loops constructs usually fail. In such situation it is often not wished that the new list elements be braced, see \xintApplyUnbraced. The \macro does not have to be expandable: \xintApply will try to expand it, the expansion may remain partial.

The ⟨*list*⟩ may itself be some macro expanding (in the previously described way) to the list of tokens to which the command \macro will be applied. For example, if the ⟨*list*⟩ expands to some positive number, then each digit will be replaced by the result of applying \macro on it.

\def\macro #1{\the\numexpr 9-#1\relax}

\xintApply\macro{\xintFac {20}}=7567097991823359999

fn ★ The macro \xintApplyNoExpand does the same job without the first initial expansion which gave the ⟨*list*⟩ of braced tokens to which \macro is applied.

23.9 \xintApplyUnbraced

ff ★ \xintApplyUnbraced{\macro}{⟨*list*⟩} is like \xintApply. The difference is that after having expanded its list argument, and applied `\macro` in turn to each item from the list, it reassembles the outputs without enclosing them in braces. The net effect is the same as doing

```
\xintListWithSep {}{\xintApply {\macro}{⟨list⟩}}
```

This is useful for preparing a macro which will itself define some other macros or make assignments, as the scope will not be limited by brace pairs.

```
\def\macro #1{\expandafter\def\csname myself#1\endcsname {#1}}
\xintApplyUnbraced\macro{{elta}{eltb}{eltc}}
\meaning\myselfelta:  macro:->elta
\meaning\myselfeltb:  macro:->eltb
\meaning\myselfeltc:  macro:->eltc
```

fn ★ The macro `\xintApplyUnbracedNoExpand` does the same job without the first initial expansion which gave the ⟨*list*⟩ of braced tokens to which `\macro` is applied.

23.10 \xintSeq

[num x] num x num x ★ `\xintSeq[d]{x}{y}` generates expandably `{x}{x+d}...` up to and possibly including `{y}` if `d>0` or down to and including `{y}` if `d<0`. Naturally `{y}` is omitted if `y-x` is not a multiple of `d`. If `d=0` the macro returns `{x}`. If `y-x` and `d` have opposite signs, the macro returns nothing. If the optional argument `d` is omitted it is taken to be the sign of `y-x`.

The current implementation is only for (short) integers; possibly, a future variant could allow big integers and fractions, although one already has access to similar functionality using `\xintApply` to get any arithmetic sequence of long integers. Currently thus, `x` and `y` are expanded inside a `\numexpr` so they may be count registers or a LaTeX `\value {countername}`, or arithmetic with such things.

```
\xintListWithSep{,\hskip2pt plus 1pt minus 1pt }{\xintSeq {12}{-25}}
12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8, -9, -10,
-11, -12, -13, -14, -15, -16, -17, -18, -19, -20, -21, -22, -23, -24, -25
\xintiiSum{\xintSeq [3]{1}{1000}}=167167
```

Important: for reasons of efficiency, this macro, when not given the optional argument `d`, works backwards, leaving in the token stream the already constructed integers, from the
IMPORTANT! tail down (or up). But this will provoke a failure of the `tex` run if the number of such items exceeds the input stack limit; on my installation this limit is at 5000.

However, when given the optional argument `d` (which may be `+1` or `-1`), the macro proceeds differently and does not put stress on the input stack (but is significantly slower for sequences with thousands of integers, especially if they are somewhat big). For example: `\xintSeq [1]{0}{5000}` works and `\xintiiSum{\xintSeq [1]{0}{5000}}` returns the correct value 12502500.

The produced integers are with explicit litteral digits, so if used in `\ifnum` or other tests they should be properly terminated[47].

23.11 Completely expandable prime test

Let us now construct a completely expandable macro which returns 1 if its given input is prime and 0 if not:

[47] a `\space` will stop the TeX scanning of a number and be gobbled in the process, maintaining expandability if this is required; the `\relax` stops the scanning but is not gobbled and remains afterwards as a token.

```
\def\remainder #1#2{\the\numexpr #1-(#1/#2)*#2\relax }
\def\IsPrime #1%
 {\xintANDof {\xintApply {\remainder {#1}}{\xintSeq {2}{\xintiSqrt{#1}}}}}
```

This uses `\xintiSqrt` and assumes its input is at least 5. Rather than **xint**'s own `\xintRem` we used a quicker `\numexpr` expression as we are dealing with short integers. Also we used `\xintANDof` which will return 1 only if all the items are non-zero. The macro is a bit silly with an even input, ok, let's enhance it to detect an even input:

```
\def\IsPrime #1%
   {\xintifOdd {#1}
        {\xintANDof % odd case
            {\xintApply {\remainder {#1}}
                        {\xintSeq [2]{3}{\xintiSqrt{#1}}}%
            }%
        }
        {\xintifEq {#1}{2}{1}{0}}%
   }
```

We used the **xint** provided expandable tests (on big integers or fractions) in oder for `\IsPrime` to be *f*-expandable.

Our integers are short, but without `\expandafter`'s with `\@firstoftwo`, or some other related techniques, direct use of `\ifnum..\fi` tests is dangerous. So to make the macro more efficient we are going to use the expandable tests provided by the package etoolbox[48]. The macro becomes:

```
\def\IsPrime #1%
   {\ifnumodd {#1}
    {\xintANDof % odd case
     {\xintApply {\remainder {#1}}{\xintSeq [2]{3}{\xintiSqrt{#1}}}}}
    {\ifnumequal {#1}{2}{1}{0}}}
```

In the odd case however we have to assume the integer is at least 7, as `\xintSeq` generates an empty list if `#1=3` or 5, and `\xintANDof` returns 1 when supplied an empty list. Let us ease up a bit `\xintANDof`'s work by letting it work on only 0's and 1's. We could use:

```
\def\IsNotDivisibleBy #1#2%
  {\ifnum\numexpr #1-(#1/#2)*#2=0 \expandafter 0\else \expandafter1\fi}
```

where the `\expandafter`'s are crucial for this macro to be *f*-expandable and hence work within the applied `\xintANDof`. Anyhow, now that we have loaded etoolbox, we might as well use:

```
\newcommand{\IsNotDivisibleBy}[2]{\ifnumequal{#1-(#1/#2)*#2}{0}{0}{1}}
```

Let us enhance our prime macro to work also on the small primes:

```
\newcommand{\IsPrime}[1] % returns 1 if #1 is prime, and 0 if not
  {\ifnumodd {#1}
    {\ifnumless {#1}{8}
      {\ifnumequal{#1}{1}{0}{1}}% 3,5,7 are primes
      {\xintANDof
         {\xintApply
        { \IsNotDivisibleBy {#1}}{\xintSeq [2]{3}{\xintiSqrt{#1}}}}%
        }}% END OF THE ODD BRANCH
    {\ifnumequal {#1}{2}{1}{0}}% EVEN BRANCH
}
```

48 http://ctan.org/pkg/etoolbox

The input is still assumed positive. There is a deliberate blank before \IsNotDivisibleBy to use this feature of \xintApply: a space stops the expansion of the applied macro (and disappears). This expansion will be done by \xintANDof, which has been designed to skip everything as soon as it finds a false (i.e. zero) input. This way, the efficiency is considerably improved.

We did generate via the \xintSeq too many potential divisors though. Later sections give two variants: one with \xintiloop (subsection 23.14) which is still expandable and another one (subsection 23.21) which is a close variant of the \IsPrime code above but with the \xintFor loop, thus breaking expandability. The xintiloop variant does not first evaluate the integer square root, the xintFor variant still does. I did not compare their efficiencies.

Let us construct with this expandable primality test a table of the prime numbers up to 1000. We need to count how many we have in order to know how many tab stops one shoud add in the last row.[49] There is some subtlety for this last row. Turns out to be better to insert a \\ only when we know for sure we are starting a new row; this is how we have designed the \OneCell macro. And for the last row, there are many ways, we use again \xintApplyUnbraced but with a macro which gobbles its argument and replaces it with a tabulation character. The \xintFor* macro would be more elegant here.

```
\newcounter{primecount}
\newcounter{cellcount}
\newcommand{\NbOfColumns}{13}
\newcommand{\OneCell}[1]{%
    \ifnumequal{\IsPrime{#1}}{1}
     {\stepcounter{primecount}
      \ifnumequal{\value{cellcount}}{\NbOfColumns}
       {\\\setcounter{cellcount}{1}#1}
       {&\stepcounter{cellcount}#1}%
     } % was prime
  {}% not a prime, nothing to do
}
\newcommand{\OneTab}[1]{&}
\begin{tabular}{|*{\NbOfColumns}{r}|}
\hline
2  \setcounter{cellcount}{1}\setcounter{primecount}{1}%
   \xintApplyUnbraced \OneCell {\xintSeq [2]{3}{999}}%
   \xintApplyUnbraced \OneTab
      {\xintSeq [1]{1}{\the\numexpr\NbOfColumns-\value{cellcount}\relax}}%
    \\
\hline
\end{tabular}
There are \arabic{primecount} prime numbers up to 1000.
```

The table has been put in float which appears on the following page. We had to be careful to use in the last row \xintSeq with its optional argument [1] so as to not generate a decreasing sequence from 1 to 0, but really an empty sequence in case the row turns out to already have all its cells (which doesn't happen here but would with a number of columns dividing 168).

[49] although a tabular row may have less tabs than in the preamble, there is a problem with the | vertical rule, if one does that.

2	3	5	7	11	13	17	19	23	29	31	37	41
43	47	53	59	61	67	71	73	79	83	89	97	101
103	107	109	113	127	131	137	139	149	151	157	163	167
173	179	181	191	193	197	199	211	223	227	229	233	239
241	251	257	263	269	271	277	281	283	293	307	311	313
317	331	337	347	349	353	359	367	373	379	383	389	397
401	409	419	421	431	433	439	443	449	457	461	463	467
479	487	491	499	503	509	521	523	541	547	557	563	569
571	577	587	593	599	601	607	613	617	619	631	641	643
647	653	659	661	673	677	683	691	701	709	719	727	733
739	743	751	757	761	769	773	787	797	809	811	821	823
827	829	839	853	857	859	863	877	881	883	887	907	911
919	929	937	941	947	953	967	971	977	983	991	997	

There are 168 prime numbers up to 1000.

23.12 \xintloop, \xintbreakloop, \xintbreakloopanddo, \xintloopskiptonext

☆ \xintloop⟨*stuff*⟩\if<test>...\repeat is an expandable loop compatible with nesting. However to break out of the loop one almost always need some un-expandable step. The cousin \xintiloop is \xintloop with an embedded expandable mechanism allowing to exit from the loop. The iterated commands may contain \par tokens or empty lines.

If a sub-loop is to be used all the material from the start of the main loop and up to the end of the entire subloop should be braced; these braces will be removed and do not create a group. The simplest to allow the nesting of one or more sub-loops is to brace everything between \xintloop and \repeat, being careful not to leave a space between the closing brace and \repeat.

As this loop and \xintiloop will primarily be of interest to experienced TeX macro programmers, my description will assume that the user is knowledgeable enough. Some examples in this document will be perhaps more illustrative than my attemps at explanation of use.

One can abort the loop with \xintbreakloop; this should not be used inside the final test, and one should expand the \fi from the corresponding test before. One has also \xintbreakloopanddo whose first argument will be inserted in the token stream after the loop; one may need a macro such as \xint_afterfi to move the whole thing after the \fi, as a simple \expandafter will not be enough.

One will usually employ some count registers to manage the exit test from the loop; this breaks expandability, see \xintiloop for an expandable integer indexed loop. Use in alignments will be complicated by the fact that cells create groups, and also from the fact that any encountered unexpandable material will cause the TeX input scanner to insert \endtemplate on each encountered & or \cr; thus \xintbreakloop may not work as expected, but the situation can be resolved via \xint_firstofone{&} or use of \TAB with \def\TAB{&}. It is thus simpler for alignments to use rather than \xintloop either the expandable \xintApplyUnbraced or the non-expandable but alignment compatible \xintApplyInline, \xintFor or \xintFor*.

As an example, let us suppose we have two macros `\A{⟨i⟩}{⟨j⟩}` and `\B{⟨i⟩}{⟨j⟩}` behaving like (small) integer valued matrix entries, and we want to define a macro `\C{⟨i⟩}{⟨j⟩}` giving the matrix product (`i` and `j` may be count registers). We will assume that `\A[I]` expands to the number of rows, `\A[J]` to the number of columns and want the produced `\C` to act in the same manner. The code is very dispendious in use of `\count` registers, not optimized in any way, not made very robust (the defined macro can not have the same name as the first two matrices for example), we just wanted to quickly illustrate use of the nesting capabilities of `\xintloop`.[50]

```
\newcount\rowmax   \newcount\colmax   \newcount\summax
\newcount\rowindex \newcount\colindex \newcount\sumindex
\newcount\tmpcount
\makeatletter
\def\MatrixMultiplication #1#2#3{%
    \rowmax #1[I]\relax
    \colmax #2[J]\relax
    \summax #1[J]\relax
    \rowindex 1
    \xintloop % loop over row index i
    {\colindex 1
     \xintloop % loop over col index k
     {\tmpcount 0
      \sumindex 1
      \xintloop % loop over intermediate index j
      \advance\tmpcount \numexpr #1\rowindex\sumindex*#2\sumindex\colindex\relax
      \ifnum\sumindex<\summax
         \advance\sumindex 1
      \repeat }%
     \expandafter\edef\csname\string#3{\the\rowindex.\the\colindex}\endcsname
      {\the\tmpcount}%
     \ifnum\colindex<\colmax
         \advance\colindex 1
     \repeat }%
    \ifnum\rowindex<\rowmax
    \advance\rowindex 1
    \repeat
    \expandafter\edef\csname\string#3{I}\endcsname{\the\rowmax}%
    \expandafter\edef\csname\string#3{J}\endcsname{\the\colmax}%
    \def #3##1{\ifx[##1\expandafter\Matrix@helper@size
                    \else\expandafter\Matrix@helper@entry\fi #3{##1}}%
}%
\def\Matrix@helper@size #1#2#3]{\csname\string#1{#3}\endcsname }%
\def\Matrix@helper@entry #1#2#3%
   {\csname\string#1{\the\numexpr#2.\the\numexpr#3}\endcsname }%
\def\A #1{\ifx[#1\expandafter\A@size
           \else\expandafter\A@entry\fi {#1}}%
\def\A@size #1#2]{\ifx I#23\else4\fi}% 3rows, 4columns
\def\A@entry #1#2{\the\numexpr #1+#2-1\relax}% not pre-computed...
\def\B #1{\ifx[#1\expandafter\B@size
```

[50] for a more sophisticated implementation of matrix multiplication, inclusive of determinants, inverses, and display utilities, with entries big integers or decimal numbers or even fractions see http://tex.stackexchange.com/a/143035/4686 from November 11, 2013.

```
            \else\expandafter\B@entry\fi {#1}}%
\def\B@size #1#2]{\ifx I#24\else3\fi}% 4rows, 3columns
\def\B@entry #1#2{\the\numexpr #1-#2\relax}% not pre-computed...
\makeatother
\MatrixMultiplication\A\B\C \MatrixMultiplication\C\C\D % etc...
\[\begin{pmatrix}
    \A11&\A12&\A13&\A14\\
    \A21&\A22&\A23&\A24\\
    \A31&\A32&\A33&\A34
  \end{pmatrix}
\times
  \begin{pmatrix}
    \B11&\B12&\B13\\
    \B21&\B22&\B23\\
    \B31&\B32&\B33\\
    \B41&\B42&\B43
  \end{pmatrix}
=
\begin{pmatrix}
    \C11&\C12&\C13\\
    \C21&\C22&\C23\\
    \C31&\C32&\C33
\end{pmatrix}\]
\[\begin{pmatrix}
    \C11&\C12&\C13\\
    \C21&\C22&\C23\\
    \C31&\C32&\C33
\end{pmatrix}^2 = \begin{pmatrix}
    \D11&\D12&\D13\\
    \D21&\D22&\D23\\
    \D31&\D32&\D33
\end{pmatrix}\]
```

$$\begin{pmatrix}1&2&3&4\\2&3&4&5\\3&4&5&6\end{pmatrix}\times\begin{pmatrix}0&-1&-2\\1&0&-1\\2&1&0\\3&2&1\end{pmatrix}=\begin{pmatrix}20&10&0\\26&12&-2\\32&14&-4\end{pmatrix}$$

$$\begin{pmatrix}20&10&0\\26&12&-2\\32&14&-4\end{pmatrix}^2=\begin{pmatrix}660&320&-20\\768&376&-16\\876&432&-12\end{pmatrix}$$

$$\begin{pmatrix}20&10&0\\26&12&-2\\32&14&-4\end{pmatrix}^3=\begin{pmatrix}20880&10160&-560\\24624&11968&-688\\28368&13776&-816\end{pmatrix}$$

$$\begin{pmatrix}20&10&0\\26&12&-2\\32&14&-4\end{pmatrix}^4=\begin{pmatrix}663840&322880&-18080\\781632&380224&-21184\\899424&437568&-24288\end{pmatrix}$$

23.13 \xintiloop, \xintiloopindex, \xintouteriloopindex, \xintbreakiloop, \xintbreakiloopanddo, \xintiloopskiptonext, \xintiloopskipandredo

☆ `\xintiloop[start+delta]`⟨*stuff*⟩`\if<test> ... \repeat` is a completely expandable nestable loop. complete expandability depends naturally on the actual iterated contents, and complete expansion will not be achievable under a sole f-expansion, as is indicated by the hollow star in the margin; thus the loop can be used inside an `\edef` but not inside arguments to the package macros. It can be used inside an `\xintexpr..\relax`.

This loop benefits via `\xintiloopindex` to (a limited access to) the integer index of the iteration. The starting value `start` (which may be a `\count`) and increment `delta` (*id.*) are mandatory arguments. A space after the closing square bracket is not significant, it will be ignored. Spaces inside the square brackets will also be ignored as the two arguments are first given to a `\numexpr...\relax`. Empty lines and explicit `\par` tokens are accepted.

As with `\xintloop`, this tool will mostly be of interest to advanced users. For nesting, one puts inside braces all the material from the start (immediately after `[start+delta]`) and up to and inclusive of the inner loop, these braces will be removed and do not create a loop. In case of nesting, `\xintouteriloopindex` gives access to the index of the outer loop. If needed one could write on its model a macro giving access to the index of the outer outer loop (or even to the nth outer loop).

The \xintiloopindex and \xintouteriloopindex can not be used inside braces, and generally speaking this means they should be expanded first when given as argument to a macro, and that this macro receives them as delimited arguments, not braced ones. Or, but naturally this will break expandability, one can assign the value of \xintiloopindex to some \count. Both \xintiloopindex and \xintouteriloopindex extend to the litteral representation of the index, thus in \ifnum tests, if it comes last one has to correctly end the macro with a \space, or encapsulate it in a \numexpr..\relax.

When the repeat-test of the loop is, for example, \ifnum\xintiloopindex<10 \repeat, this means that the last iteration will be with \xintiloopindex=10 (assuming delta=1). There is also \ifnum\xintiloopindex=10 \else\repeat to get the last iteration to be the one with \xintiloopindex=10.

One has \xintbreakiloop and \xintbreakiloopanddo to abort the loop. The syntax of \xintbreakiloopanddo is a bit surprising, the sequence of tokens to be executed after breaking the loop is not within braces but is delimited by a dot as in:

```
\xintbreakiloopanddo <afterloop>.etc.. etc... \repeat
```

The reason is that one may wish to use the then current value of \xintiloopindex in <afterloop> but it can't be within braces at the time it is evaluated. However, it is not that easy as \xintiloopindex must be expanded before, so one ends up with code like this:

```
\expandafter\xintbreakiloopanddo\expandafter\macro\xintiloopindex.%
                        etc.. etc.. \repeat
```

As moreover the \fi from the test leading to the decision of breaking out of the loop must be cleared out of the way, the above should be a branch of an expandable conditional test, else one needs something such as:

```
\xint_afterfi{\expandafter\xintbreakiloopanddo\expandafter\macro\xintiloopindex.}%
                        \fi etc..etc.. \repeat
```

There is \xintiloopskiptonext to abort the current iteration and skip to the next, \xintiloopskipandredo to skip to the end of the current iteration and redo it with the same value of the index (something else will have to change for this not to become an eternal loop...).

Inside alignments, if the looped-over text contains a & or a \cr, any un-expandable material before a \xintiloopindex will make it fail because of \endtemplate; in such cases one can always either replace & by a macro expanding to it or replace it by a suitable \firstofone{&}, and similarly for \cr.

As an example, let us construct an \edef\z{...} which will define \z to be a list of prime numbers:

```
\edef\z
{\xintiloop [10001+2]
  {\xintiloop [3+2]
   \ifnum\xintouteriloopindex<\numexpr\xintiloopindex*\xintiloopindex\relax
          \xintouteriloopindex,
          \expandafter\xintbreakiloop
   \fi
   \ifnum\xintouteriloopindex=\numexpr
        (\xintouteriloopindex/\xintiloopindex)*\xintiloopindex\relax
   \else
```

```
   \repeat
  }% no space here
 \ifnum \xintiloopindex < 10999 \repeat }%
```

`\meaning\z` macro:->10007, 10009, 10037, 10039, 10061, 10067, 10069, 10079, 10091, 10093, 10099, 10103, 10111, 10133, 10139, 10141, 10151, 10159, 10163, 10169, 10177, 10181, 10193, 10211, 10223, 10243, 10247, 10253, 10259, 10267, 10271, 10273, 10289, 10301, 10303, 10313, 10321, 10331, 10333, 10337, 10343, 10357, 10369, 10391, 10399, 10427, 10429, 10433, 10453, 10457, 10459, 10463, 10477, 10487, 10499, 10501, 10513, 10529, 10531, 10559, 10567, 10589, 10597, 10601, 10607, 10613, 10627, 10631, 10639, 10651, 10657, 10663, 10667, 10687, 10691, 10709, 10711, 10723, 10729, 10733, 10739, 10753, 10771, 10781, 10789, 10799, 10831, 10837, 10847, 10853, 10859, 10861, 10867, 10883, 10889, 10891, 10903, 10909, 10937, 10939, 10949, 10957, 10973, 10979, 10987, 10993, and we should have taken some steps to not have a trailing comma, but the point was to show that one can do that in an `\edef`! See also subsection 23.14 which extracts from this code its way of testing primality.

Let us create an alignment where each row will contain all divisors of its first entry.

```
\tabskip1ex
\halign{&\hfil#\hfil\cr
    \xintiloop [1+1]
    {\expandafter\bfseries\xintiloopindex &
     \xintiloop [1+1]
     \ifnum\xintouteriloopindex=\numexpr
           (\xintouteriloopindex/\xintiloopindex)*\xintiloopindex\relax
     \xintiloopindex&\fi
     \ifnum\xintiloopindex<\xintouteriloopindex\space % \space is CRUCIAL
     \repeat \cr }%
    \ifnum\xintiloopindex<30
    \repeat }
```

We wanted this first entry in bold face, but `\bfseries` leads to unexpandable tokens, so the `\expandafter` was necessary for `\xintiloopindex` and `\xintouteriloopindex` not to be confronted with a hard to digest `\endtemplate`. An alternative way of coding is:

```
\def\firstofone #1{#1}%
\halign{&\hfil#\hfil\cr
  \xintiloop [1+1]
    {\bfseries\xintiloopindex\firstofone{&}%
    \xintiloop [1+1] \ifnum\xintouteriloopindex=\numexpr
    (\xintouteriloopindex/\xintiloopindex)*\xintiloopindex\relax
    \xintiloopindex\firstofone{&}\fi
    \ifnum\xintiloopindex<\xintouteriloopindex\space % \space is CRUCIAL
    \repeat \firstofone{\cr}}%
  \ifnum\xintiloopindex<30 \repeat }
```

Here is the output, thus obtained without any count register:

1	1					
2	1	2				
3	1	3				
4	1	2	4			
5	1	5				
6	1	2	3	6		
7	1	7				
8	1	2	4	8		
9	1	3	9			
10	1	2	5	10		
11	1	11				
12	1	2	3	4	6	12

13	1 13
14	1 2 7 14
15	1 3 5 15
16	1 2 4 8 16
17	1 17
18	1 2 3 6 9 18
19	1 19
20	1 2 4 5 10 20
21	1 3 7 21
22	1 2 11 22
23	1 23
24	1 2 3 4 6 8 12 24
25	1 5 25
26	1 2 13 26
27	1 3 9 27
28	1 2 4 7 14 28
29	1 29
30	1 2 3 5 6 10 15 30

23.14 Another completely expandable prime test

The `\IsPrime` macro from subsection 23.11 checked expandably if a (short) integer was prime, here is a partial rewrite using `\xintiloop`. We use the `etoolbox` expandable conditionals for convenience, but not everywhere as `\xintiloopindex` can not be evaluated while being braced. This is also the reason why `\xintbreakiloopanddo` is delimited, and the next macro `\SmallestFactor` which returns the smallest prime factor examplifies that. One could write more efficient completely expandable routines, the aim here was only to illustrate use of the general purpose `\xintiloop`. A little table giving the first values of `\SmallestFactor` follows, its coding uses `\xintFor`, which is described later; none of this uses count registers.

```
\newcommand{\IsPrime}[1] % returns 1 if #1 is prime, and 0 if not
  {\ifnumodd {#1}
    {\ifnumless {#1}{8}
      {\ifnumequal{#1}{1}{0}{1}}% 3,5,7 are primes
      {\if
       \xintiloop [3+2]
       \ifnum#1<\numexpr\xintiloopindex*\xintiloopindex\relax
           \expandafter\xintbreakiloopanddo\expandafter1\expandafter.%
       \fi
       \ifnum#1=\numexpr (#1/\xintiloopindex)*\xintiloopindex\relax
       \else
       \repeat 00\expandafter0\else\expandafter1\fi
      }%
    }% END OF THE ODD BRANCH
    {\ifnumequal {#1}{2}{1}{0}}% EVEN BRANCH
}%
\catcode`_ 11
\newcommand{\SmallestFactor}[1] % returns the smallest prime factor of #1>1
  {\ifnumodd {#1}
    {\ifnumless {#1}{8}
      {#1}% 3,5,7 are primes
      {\xintiloop [3+2]
       \ifnum#1<\numexpr\xintiloopindex*\xintiloopindex\relax
           \xint_afterfi{\xintbreakiloopanddo#1.}%
       \fi
       \ifnum#1=\numexpr (#1/\xintiloopindex)*\xintiloopindex\relax
           \xint_afterfi{\expandafter\xintbreakiloopanddo\xintiloopindex.}%
       \fi
       \iftrue\repeat
```

```
      }%
     }% END OF THE ODD BRANCH
   {2}% EVEN BRANCH
}%
\catcode`_ 8
  \begin{tabular}{|c|*{10}c|}
    \hline
    \xintFor #1 in {0,1,2,3,4,5,6,7,8,9}\do {&\bfseries #1}\\
    \hline
    \bfseries 0&--&--&2&3&2&5&2&7&2&3\\
    \xintFor #1 in {1,2,3,4,5,6,7,8,9}\do
    {\bfseries #1%
      \xintFor #2 in {0,1,2,3,4,5,6,7,8,9}\do
      {&\SmallestFactor{#1#2}}\\}%
    \hline
  \end{tabular}
```

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	–	–	2	3	2	5	2	7	2	3
1	2	11	2	13	2	3	2	17	2	19
2	2	3	2	23	2	5	2	3	2	29
3	2	31	2	3	2	5	2	37	2	3
4	2	41	2	43	2	3	2	47	2	7
5	2	3	2	53	2	5	2	3	2	59
6	2	61	2	3	2	5	2	67	2	3
7	2	71	2	73	2	3	2	7	2	79
8	2	3	2	83	2	5	2	3	2	89
9	2	7	2	3	2	5	2	97	2	3

23.15 A table of factorizations

As one more example with `\xintiloop` let us use an alignment to display the factorization of some numbers. The loop will actually only play a minor rôle here, just handling the row index, the row contents being almost entirely produced via a macro `\factorize`. The factorizing macro does not use `\xintiloop` as it didn't appear to be the convenient tool. As `\factorize` will have to be used on `\xintiloopindex`, it has been defined as a delimited macro.

To spare some fractions of a second in the compilation time of this document (which has many many other things to do), 2147483629 and 2147483647, which turn out to be prime numbers, are not given to `factorize` but just typeset directly; this illustrates use of `\xintiloopskiptonext`.

```
\tabskip1ex
\halign {\hfil\strut#\hfil&&\hfil#\hfil\cr\noalign{\hrule}
         \xintiloop ["7FFFFFE0+1]
         \expandafter\bfseries\xintiloopindex &
         \ifnum\xintiloopindex="7FFFFFED
              \number"7FFFFFED\cr\noalign{\hrule}
              \expandafter\xintiloopskiptonext
         \fi
         \expandafter\factorize\xintiloopindex.\cr\noalign{\hrule}
```

```
        \ifnum\xintiloopindex<"7FFFFFFE
        \repeat
        \bfseries \number"7FFFFFFF&\number "7FFFFFFF\cr\noalign{\hrule}
}
```

The table has been made into a float which appears on the next page. Here is now the code for factorization; the conditionals use the package provided `\xint_firstoftwo` and `\xint_secondoftwo`, one could have employed rather LaTeX's own `\@firstoftwo` and `\@secondoftwo`, or, simpler still in LaTeX context, the `\ifnumequal`, `\ifnumless ...`, utilities from the package `etoolbox` which do exactly that under the hood. Only TeX acceptable numbers are treated here, but it would be easy to make a translation and use the **xint** macros, thus extending the scope to big numbers; naturally up to a cost in speed.

The reason for some strange looking expressions is to avoid arithmetic overflow.

```
\catcode`_ 11
\def\abortfactorize #1\xint_secondoftwo\fi #2#3{\fi}

\def\factorize #1.{\ifnum#1=1 \abortfactorize\fi
          % avoid overflow if #1="7FFFFFFF
          \ifnum\numexpr #1-2=\numexpr ((#1/2)-1)*2\relax
               \expandafter\xint_firstoftwo
          \else\expandafter\xint_secondoftwo
          \fi
         {2&\expandafter\factorize\the\numexpr#1/2.}%
         {\factorize_b #1.3.}}%

\def\factorize_b #1.#2.{\ifnum#1=1 \abortfactorize\fi
         % this will avoid overflow which could result from #2*#2
         \ifnum\numexpr #1-(#2-1)*#2<#2
                 #1\abortfactorize % this #1 is prime
         \fi
         % again, avoiding overflow as \numexpr integer division
         % rounds rather than truncates.
         \ifnum \numexpr #1-#2=\numexpr ((#1/#2)-1)*#2\relax
              \expandafter\xint_firstoftwo
         \else\expandafter\xint_secondoftwo
         \fi
         {#2&\expandafter\factorize_b\the\numexpr#1/#2.#2.}%
         {\expandafter\factorize_b\the\numexpr #1\expandafter.%
                                    \the\numexpr #2+2.}}%
\catcode`_ 8
```

The next utilities are not compatible with expansion-only context.

23.16 \xintApplyInline

*o *f* `\xintApplyInline{\macro}{⟨list⟩}` works non expandably. It applies the one-parameter `\macro` to the first element of the expanded list (`\macro` may have itself some arguments, the list item will be appended as last argument), and is then re-inserted in the input stream after the tokens resulting from this first expansion of `\macro`. The next item is then handled.

2147483616	2	2	2	2	2	3	2731	8191	
2147483617	6733	318949							
2147483618	2	7	367	417961					
2147483619	3	3	23	353	29389				
2147483620	2	2	5	4603	23327				
2147483621	14741	145681							
2147483622	2	3	17	467	45083				
2147483623	79	967	28111						
2147483624	2	2	2	11	13	1877171			
2147483625	3	5	5	5	7	199	4111		
2147483626	2	19	37	1527371					
2147483627	47	53	862097						
2147483628	2	2	3	3	59652323				
2147483629	2147483629								
2147483630	2	5	6553	32771					
2147483631	3	137	263	19867					
2147483632	2	2	2	2	7	73	262657		
2147483633	5843	367531							
2147483634	2	3	12097	29587					
2147483635	5	11	337	115861					
2147483636	2	2	536870909						
2147483637	3	3	3	13	6118187				
2147483638	2	2969	361651						
2147483639	7	17	18046081						
2147483640	2	2	2	3	5	29	43	113	127
2147483641	2699	795659							
2147483642	2	23	46684427						
2147483643	3	715827881							
2147483644	2	2	233	1103	2089				
2147483645	5	19	22605091						
2147483646	2	3	3	7	11	31	151	331	
2147483647	2147483647								

A table of factorizations

This is to be used in situations where one needs to do some repetitive things. It is not expandable and can not be completely expanded inside a macro definition, to prepare material for later execution, contrarily to what `\xintApply` or `\xintApplyUnbraced` achieve.

```
\def\Macro #1{\advance\cnta #1 , \the\cnta}
\cnta 0
0\xintApplyInline\Macro {3141592653}.
```
Output: 0, 3, 4, 8, 9, 14, 23, 25, 31, 36, 39.

The first argument `\macro` does not have to be an expandable macro.

`\xintApplyInline` submits its second, token list parameter to an *f*-expansion. Then, each *unbraced* item will also be *f*-expanded. This provides an easy way to insert one list inside another. *Braced* items are not expanded. Spaces in-between items are gobbled (as well as those at the start or the end of the list), but not the spaces *inside* the braced items.

\xintApplyInline, despite being non-expandable, does survive to contexts where the executed \macro closes groups, as happens inside alignments with the tabulation character &. This tabular for example:

N	N^2	N^3
17	289	4913
28	784	21952
39	1521	59319
50	2500	125000
61	3721	226981

was obtained from the following input:

```
\begin{tabular}{ccc}
     $N$ & $N^2$ & $N^3$ \\ \hline
     \def\Row #1{ #1 & \xintiSqr {#1} & \xintiPow {#1}{3} \\ \hline }%
     \xintApplyInline \Row {\xintCSVtoList{17,28,39,50,61}}
\end{tabular}
```

Despite the fact that the first encountered tabulation character in the first row close a group and thus erases \Row from TeX's memory, \xintApplyInline knows how to deal with this.

Using \xintApplyUnbraced is an alternative: the difference is that this would have prepared all rows first and only put them back into the token stream once they are all assembled, whereas with \xintApplyInline each row is constructed and immediately fed back into the token stream: when one does things with numbers having hundreds of digits, one learns that keeping on hold and shuffling around hundreds of tokens has an impact on TeX's speed (make this "thousands of tokens" for the impact to be noticeable).

One may nest various \xintApplyInline's. For example (see the table on this page):

```
\def\Row #1{#1:\xintApplyInline {\Item {#1}}{0123456789}\\ }%
\def\Item #1#2{&\xintiPow {#1}{#2}}%
\begin{tabular}{ccccccccccc}
     &0&1&2&3&4&5&6&7&8&9\\ \hline
     \xintApplyInline \Row {0123456789}
\end{tabular}
```

	0	1	2	3	4	5	6	7	8	9
0:	1	0	0	0	0	0	0	0	0	0
1:	1	1	1	1	1	1	1	1	1	1
2:	1	2	4	8	16	32	64	128	256	512
3:	1	3	9	27	81	243	729	2187	6561	19683
4:	1	4	16	64	256	1024	4096	16384	65536	262144
5:	1	5	25	125	625	3125	15625	78125	390625	1953125
6:	1	6	36	216	1296	7776	46656	279936	1679616	10077696
7:	1	7	49	343	2401	16807	117649	823543	5764801	40353607
8:	1	8	64	512	4096	32768	262144	2097152	16777216	134217728
9:	1	9	81	729	6561	59049	531441	4782969	43046721	387420489

One could not move the definition of \Item inside the tabular, as it would get lost after the first &. But this works:

```
\begin{tabular}{ccccccccccc}
    &0&1&2&3&4&5&6&7&8&9\\ \hline
```

```
    \def\Row #1{#1:\xintApplyInline {&\xintiPow {#1}}{0123456789}\\ }%
    \xintApplyInline \Row {0123456789}
\end{tabular}
```

A limitation is that, contrarily to what one may have expected, the \macro for an \xintApplyInline can not be used to define the \macro for a nested sub-\xintApplyInline. For example, this does not work:

```
  \def\Row #1{#1:\def\Item ##1{&\xintiPow {#1}{##1}}%
               \xintApplyInline \Item {0123456789}\\ }%
  \xintApplyInline \Row {0123456789} % does not work
```

But see \xintFor.

23.17 \xintFor, \xintFor*

on \xintFor is a new kind of for loop. Rather than using macros for encapsulating list items, its behavior is more like a macro with parameters: #1, #2, ..., #9 are used to represent the items for up to nine levels of nested loops. Here is an example:

```
\xintFor #9 in {1,2,3} \do {%
  \xintFor #1 in {4,5,6} \do {%
    \xintFor #3 in {7,8,9} \do {%
      \xintFor #2 in {10,11,12} \do {%
      $$#9\times#1\times#3\times#2=\xintiiPrd{{#1}{#2}{#3}{#9}}$$}}}}
```

This example illustrates that one does not have to use #1 as the first one: the order is arbitrary. But each level of nesting should have its specific macro parameter. Nine levels of nesting is presumably overkill, but I did not know where it was reasonable to stop. \par tokens are accepted in both the comma separated list and the replacement text.

> A macro \macro whose definition uses internally an \xintFor loop may be used inside another \xintFor loop even if the two loops both use the same macro parameter. Note: the loop definition inside \macro must double the character # as is the general rule in TeX with definitions done inside macros.
>
> The macros \xintFor and \xintFor* are not expandable, one can not use them inside an \edef. But they may be used inside alignments (such as a LaTeX tabular), as will be shown in examples.

The spaces between the various declarative elements are all optional; furthermore spaces around the commas or at the start and end of the list argument are allowed, they will be removed. If an item must contain itself commas, it should be braced to prevent these commas from being misinterpreted as list separator. These braces will be removed during processing. The list argument may be a macro \MyList expanding in one step to the comma separated list (if it has no arguments, it does not have to be braced). It will be expanded (only once) to reveal its comma separated items for processing, comma separated items will not be expanded before being fed into the replacement text as #1, or #2, etc..., only leading and trailing spaces are removed.

**fn* A starred variant \xintFor* deals with lists of braced items, rather than comma separated items. It has also a distinct expansion policy, which is detailed below.

Contrarily to what happens in loops where the item is represented by a macro, here it is truly exactly as when defining (in LaTeX) a "command" with parameters #1, etc... This may avoid the user quite a few troubles with `\expandafters` or other `\edef/\noexpands` which one encounters at times when trying to do things with LaTeX's `\@for` or other loops which encapsulate the item in a macro expanding to that item.

The non-starred variant `\xintFor` deals with comma separated values (*spaces before and after the commas are removed*) and the comma separated list may be a macro which is only expanded once (to prevent expansion of the first item `\x` in a list directly input as `\x,\y,...` it should be input as `{\x},\y,..` or `<space>\x,\y,..`, naturally all of that within the mandatory braces of the `\xintFor #n in {list}` syntax). The items are not expanded, if the input is `<stuff>,\x,<stuff>` then `#1` will be at some point `\x` not its expansion (and not either a macro with `\x` as replacement text, just the token `\x`). Input such as `<stuff>,,<stuff>` creates an empty `#1`, the iteration is not skipped. An empty list does lead to the use of the replacement text, once, with an empty `#1` (or `#n`). Except if the entire list is represented as a single macro with no parameters, it must be braced.

The starred variant `\xintFor*` deals with token lists (*spaces between braced items or single tokens are not significant*) and *f*-expands each *unbraced* list item. This makes it easy to simulate concatenation of various list macros `\x`, `\y`, ... If `\x` expands to `{1}{2}{3}` and `\y` expands to `{4}{5}{6}` then `{\x\y}` as argument to `\xintFor*` has the same effect as `{{1}{2}{3}{4}{5}{6}}`[51]. Spaces at the start, end, or in-between items are gobbled (but naturally not the spaces which may be inside *braced* items). Except if the list argument is a single macro with no parameters, it must be braced. Each item which is not braced will be fully expanded (as the `\x` and `\y` in the example above). An empty list leads to an empty result.

The macro `\xintSeq` which generates arithmetic sequences may only be used with `\xintFor*` (numbers from output of `\xintSeq` are braced, not separated by commas).

```
\xintFor* #1 in {\xintSeq [+2]{-7}{+2}}\do {stuff with #1}
```

will have `#1=-7,-5,-3,-1, and 1`. The `#1` as issued from the list produced by `\xintSeq` is the litteral representation as would be produced by `\arabic` on a LaTeX counter, it is not a count register. When used in `\ifnum` tests or other contexts where TeX looks for a number it should thus be postfixed with `\relax` or `\space`.

When nesting `\xintFor*` loops, using `\xintSeq` in the inner loops is inefficient, as the arithmetic sequence will be re-created each time. A more efficient style is:

```
\edef\innersequence {\xintSeq[+2]{-50}{50}}%
\xintFor* #1 in {\xintSeq {13}{27}} \do
    {\xintFor* #2 in \innersequence \do {stuff with #1 and #2}%
     .. some other macros .. }
```

This is a general remark applying for any nesting of loops, one should avoid recreating the inner lists of arguments at each iteration of the outer loop. However, in the example above, if the `.. some other macros ..` part closes a group which was opened before the `\edef\innersequence`, then this definition will be lost. An alternative to `\edef`, also

[50] braces around single token items are optional so this is the same as `{123456}`.

efficient, exists when dealing with arithmetic sequences: it is to use the \xintintegers keyword (described later) which simulates infinite arithmetic sequences; the loops will then be terminated via a test #1 (or #2 etc...) and subsequent use of \xintBreakFor.

The \xintFor loops are not completely expandable; but they may be nested and used inside alignments or other contexts where the replacement text closes groups. Here is an example (still using LaTeX's tabular):

A:	$(a \to A)$	$(b \to A)$	$(c \to A)$	$(d \to A)$	$(e \to A)$
B:	$(a \to B)$	$(b \to B)$	$(c \to B)$	$(d \to B)$	$(e \to B)$
C:	$(a \to C)$	$(b \to C)$	$(c \to C)$	$(d \to C)$	$(e \to C)$

```
\begin{tabular}{rccccc}
    \xintFor #7 in {A,B,C} \do {%
      #7:\xintFor* #3 in {abcde} \do {&($ #3 \to #7 $)}\\ }%
\end{tabular}
```

When inserted inside a macro for later execution the # characters must be doubled.[51] For example:

```
\def\T{\def\z {}%
  \xintFor* ##1 in {{u}{v}{w}} \do {%
    \xintFor ##2 in {x,y,z} \do {%
      \expandafter\def\expandafter\z\expandafter {\z\sep (##1,##2)} }%
  }%
}%
\T\def\sep {\def\sep{, }}\z
```

(u,x), (u,y), (u,z), (v,x), (v,y), (v,z), (w,x), (w,y), (w,z)

Similarly when the replacement text of \xintFor defines a macro with parameters, the macro character # must be doubled.

It is licit to use inside an \xintFor a \macro which itself has been defined to use internally some other \xintFor. The same macro parameter #1 can be used with no conflict (as mentioned above, in the definition of \macro the # used in the \xintFor declaration must be doubled, as is the general rule in TeX with things defined inside other things).

The iterated commands as well as the list items are allowed to contain explicit \par tokens. Neither \xintFor nor \xintFor* create groups. The effect is like piling up the iterated commands with each time #1 (or #2 ...) replaced by an item of the list. However, contrarily to the completely expandable \xintApplyUnbraced, but similarly to the non completely expandable \xintApplyInline each iteration is executed first before looking at the next #1[52] (and the starred variant \xintFor* keeps on expanding each unbraced item it finds, gobbling spaces).

23.18 \xintifForFirst, \xintifForLast

n n ★ \xintifForFirst {YES branch}{NO branch} and \xintifForLast {YES branch}{NO
n n ★ branch} execute the YES or NO branch if the \xintFor or \xintFor* loop is currently in its first, respectively last, iteration.

---

[51] sometimes what seems to be a macro argument isn't really; in \raisebox{1cm}{\xintFor #1 in {a,b,c}\do {#1}} no doubling should be done. [52] to be completely honest, both \xintFor and \xintFor* intially scoop up both the list and the iterated commands; \xintFor scoops up a second time the entire comma separated list in order to feed it to \xintCSVtoList. The starred variant \xintFor* which does not need this step will thus be a bit faster on equivalent inputs.

Designed to work as expected under nesting. Don't forget an empty brace pair `{}` if a branch is to do nothing. May be used multiple times in the replacement text of the loop.

There is no such thing as an iteration counter provided by the `\xintFor` loops; the user is invited to define if needed his own count register or LaTeX counter, for example with a suitable `\stepcounter` inside the replacement text of the loop to update it.

23.19 `\xintBreakFor`, `\xintBreakForAndDo`

One may immediately terminate an `\xintFor` or `\xintFor*` loop with `\xintBreakFor`. As the criterion for breaking will be decided on a basis of some test, it is recommended to use for this test the syntax of ifthen[53] or etoolbox[54] or the **xint** own conditionals, rather than one of the various `\if...\fi` of TeX. Else (and this is without even mentioning all the various pecularities of the `\if...\fi` constructs), one has to carefully move the break after the closing of the conditional, typically with `\expandafter\xintBreakFor\fi`.[55]

There is also `\xintBreakForAndDo`. Both are illustrated by various examples in the next section which is devoted to "forever" loops.

23.20 `\xintintegers`, `\xintdimensions`, `\xintrationals`

If the list argument to `\xintFor` (or `\xintFor*`, both are equivalent in this context) is `\xintintegers` (equivalently `\xintegers`) or more generally `\xintintegers[start+delta]` (*the whole within braces*!)[56], then `\xintFor` does an infinite iteration where `#1` (or `#2`, ..., `#9`) will run through the arithmetic sequence of (short) integers with initial value `start` and increment `delta` (default values: `start=1`, `delta=1`; if the optional argument is present it must contains both of them, and they may be explicit integers, or macros or count registers). The `#1` (or `#2`, ..., `#9`) will stand for `\numexpr <opt sign><digits>\relax`, and the litteral representation as a string of digits can thus be obtained as `\the#1` or `\number#1`. Such a `#1` can be used in an `\ifnum` test with no need to be postfixed with a space or a `\relax` and one should *not* add them.

If the list argument is `\xintdimensions` or more generally `\xintdimensions[start+delta]` (*within braces*!), then `\xintFor` does an infinite iteration where `#1` (or `#2`, ..., `#9`) will run through the arithmetic sequence of dimensions with initial value `start` and increment `delta`. Default values: `start=0pt`, `delta=1pt`; if the optional argument is present it must contain both of them, and they may be explicit specifications, or macros, or dimen registers, or length commands in LaTeX (the stretch and shrink components will be discarded). The `#1` will be `\dimexpr <opt sign><digits>sp\relax`, from which one can get the litteral (approximate) representation in points via `\the#1`. So `#1` can be used anywhere TeX expects a dimension (and there is no need in conditionals to insert a `\relax`, and one should *not* do it), and to print its value one uses `\the#1`. The chosen representation guarantees exact incrementation with no rounding errors accumulating from converting into points at each step.

---

[53] http://ctan.org/pkg/ifthen [54] http://ctan.org/pkg/etoolbox [55] the difficulties here are similar to those mentioned in section 10, although less severe, as complete expandability is not to be maintained; hence the allowed use of ifthen. [56] the `start+delta` optional specification may have extra spaces around the plus sign of near the square brackets, such spaces are removed. The same applies with `\xintdimensions` and `\xintrationals`.

```
\def\DimToNum #1{\number\dimexpr #1\relax }
\xintNewNumExpr \FA [2] {{_DimToNum {$2}}^3/{_DimToNum {$1}}^2}    % cube
\xintNewNumExpr \FB [2] {sqrt ({_DimToNum {$2}}*{_DimToNum {$1}})} % sqrt
\xintNewExpr \Ratio [2] {trunc({_DimToNum {$2}}/{_DimToNum{$1}},3)}
\xintFor #1 in {\xintdimensions [0pt+.1pt]} \do
 {\ifdim #1>2cm \expandafter\xintBreakFor\fi
  {\color [rgb]{\Ratio {2cm}{#1},0,0}%
  \vrule width .1pt height \FB {2cm}{#1}sp depth -\FA {2cm}{#1}sp }%
 }% end of For iterated text
```

The graphic, with the code on its right[57], is for illustration only, not only because of pdf rendering artefacts when displaying adjacent rules (which do *not* show in `dvi` output as rendered by `xdvi`, and depend from your viewer), but because not using anything but rules it is quite inefficient and must do lots of computations to not confer a too ragged look to the borders. With a width of `.5pt` rather than `.1pt` for the rules, one speeds up the drawing by a factor of five, but the boundary is then visibly ragged. [58]

If the list argument to `\xintFor` (or `\xintFor*`) is `\xintrationals` or more generally `\xintrationals[start+delta]` (*within braces*!), then `\xintFor` does an infinite iteration where `#1` (or `#2`, ..., `#9`) will run through the arithmetic sequence of **xintfrac** fractions with initial value `start` and increment `delta` (default values: `start=1/1`, `delta=1/1`). This loop works *only with* ***xintfrac*** *loaded*. if the optional argument is present it must contain both of them, and they may be given in any of the formats recognized by **xintfrac** (fractions, decimal numbers, numbers in scientific notations, numerators and denominators in scientific notation, etc...) , or as macros or count registers (if they are short integers). The `#1` (or `#2`, ..., `#9`) will be an `a/b` fraction (without a `[n]` part), where the denominator `b` is the product of the denominators of `start` and `delta` (for reasons of speed `#1` is not reduced to irreducible form, and for another reason explained later `start` and `delta` are not put either into irreducible form; the input may use explicitely `\xintIrr` to achieve that).

```
\xintFor #1 in {\xintrationals [10/21+1/21]} \do
{#1=\xintifInt {#1}
  {\textcolor{blue}{\xintTrunc{10}{#1}}}
  {\xintTrunc{10}{#1}}% in blue if an integer
  \xintifGt {#1}{1.123}{\xintBreakFor}{, }%
}
```

10/21=0.4761904761, 11/21=0.5238095238, 12/21=0.5714285714, 13/21=0.6190476190, 14/21=0.6666666666, 15/21=0.7142857142, 16/21=0.7619047619, 17/21=0.8095238095, 18/21=0.8571428571, 19/21=0.9047619047, 20/21=0.9523809523, 21/21=1.0000000000, 22/21=1.0476190476, 23/21=1.0952380952, 24/21=1.1428571428

The example above confirms that computations are done exactly, and illustrates that the two initial (reduced) denominators are not multiplied when they are found to be equal. It

---

[57] the somewhat peculiar use of `_` and `$` is explained in subsection 26.6; they are made necessary from the fact that the parameters are passed to a *macro* (`\DimToNum`) and not only to *functions*, as are known to `\xintexpr`. But one can also define directly the desired function, for example the constructed `\FA` turns out to have meaning `macro:#1#2->\romannumeral -`0\xintiRound 0{\xintDiv {\xintPow {\DimToNum {#2}}{3}}{\xintPow {\DimToNum {#1}}{2}}}`, where the `\romannumeral` part is only to ensure it expands in only two steps, and could be removed. A handwritten macro would use here `\xintiPow` and not `\xintPow`, as we know it has to deal with integers only. See the next footnote. [58] to tell the whole truth we cheated and divided by `10` the computation time through using the following definitions, together with a horizontal step of `.25pt` rather than `.1pt`. The displayed original code would make the slowest computation of all those done in this document using the **xint** bundle

```
\def\DimToNum #1{\the\numexpr \dimexpr#1\relax/10000\relax } % no need to be more precise!
\def\FA #1#2{\xintDSH {-4}{\xintQuo {\xintiPow {\DimToNum {#2}}{3}}{\xintiSqr {\DimToNum{#1}}}}}
\def\FB #1#2{\xintDSH {-4}{\xintiSqrt {\xintiMul {\DimToNum {#2}}{\DimToNum{#1}}}}}
\def\Ratio #1#2{\xintTrunc {2}{\DimToNum {#2}/\DimToNum{#1}}}
\xintFor #1 in {\xintdimensions [0pt+.25pt]} \do
 {\ifdim #1>2cm \expandafter\xintBreakFor\fi
  {\color [rgb]{\Ratio {2cm}{#1},0,0}%
  \vrule width .25pt height \FB {2cm}{#1}sp depth -\FA {2cm}{#1}sp }%
 }% end of For iterated text
```

macros!

is thus recommended to input `start` and `delta` with a common smallest possible denominator, or as fixed point numbers with the same numbers of digits after the decimal mark; and this is also the reason why `start` and `delta` are not by default made irreducible. As internally the computations are done with numerators and denominators completely expanded, one should be careful not to input numbers in scientific notation with exponents in the hundreds, as they will get converted into as many zeroes.

```
\xintFor #1 in {\xintrationals [0.000+0.125]} \do
{\edef\tmp{\xintTrunc{3}{#1}}%
 \xintifInt {#1}
    {\textcolor{blue}{\tmp}}
    {\tmp}%
    \xintifGt {#1}{2}{\xintBreakFor}{, }%
  }
```

0, 0.125, 0.250, 0.375, 0.500, 0.625, 0.750, 0.875, 1.000, 1.125, 1.250, 1.375, 1.500, 1.625, 1.750, 1.875, 2.000, 2.125

We see here that `\xintTrunc` outputs (deliberately) zero as `0`, not (here) `0.000`, the idea being not to lose the information that the truncated thing was truly zero. Perhaps this behavior should be changed? or made optional? Anyhow printing of fixed points numbers should be dealt with via dedicated packages such as `numprint` or `siunitx`.

23.21 Another table of primes

As a further example, let us dynamically generate a tabular with the first 50 prime numbers after 12345. First we need a macro to test if a (short) number is prime. Such a completely expandable macro was given in subsection 23.10, here we consider a variant which will be slightly more efficient. This new `\IsPrime` has two parameters. The first one is a macro which it redefines to expand to the result of the primality test applied to the second argument. For convenience we use the etoolbox wrappers to various `\ifnum` tests, although here there isn't anymore the constraint of complete expandability (but using explicit `\if..` `\fi` in tabulars has its quirks); equivalent tests are provided by **xint**, but they have some overhead as they are able to deal with arbitrarily big integers.

```
\def\IsPrime #1#2% #1=\Result, #2=tested number (assumed >0).
{\edef\TheNumber {\the\numexpr #2}% hence #2 may be a count or \numexpr.
 \ifnumodd {\TheNumber}
 {\ifnumgreater {\TheNumber}{1}
  {\edef\ItsSquareRoot{\xintiSqrt \TheNumber}%
    \xintFor ##1 in {\xintintegers [3+2]}\do
    {\ifnumgreater {##1}{\ItsSquareRoot} % ##1 is a \numexpr.
               {\def#1{1}\xintBreakFor}
               {}%
     \ifnumequal {\TheNumber}{(\TheNumber/##1)*##1}
                 {\def#1{0}\xintBreakFor }
                 {}%
    }}
  {\def#1{0}}}% 1 is not prime
 {\ifnumequal {\TheNumber}{2}{\def#1{1}}{\def#1{0}}}%
}
```

As we used `\xintFor` inside a macro we had to double the `#` in its `#1` parameter. Here is now the code which creates the prime table (the table has been put in a float, which appears on the next page):

12347	12373	12377	12379	12391	12401	12409
12413	12421	12433	12437	12451	12457	12473
12479	12487	12491	12497	12503	12511	12517
12527	12539	12541	12547	12553	12569	12577
12583	12589	12601	12611	12613	12619	12637
12641	12647	12653	12659	12671	12689	12697
12703	12713	12721	12739	12743	12757	12763
12781	These are the first 50 primes after 12345.					

```
\newcounter{primecount}
\newcounter{cellcount}
\begin{figure*}[ht!]
  \centering
  \begin{tabular}{|*{7}c|}
  \hline
  \setcounter{primecount}{0}\setcounter{cellcount}{0}%
  \xintFor #1 in {\xintintegers [12345+2]} \do
% #1 is a \numexpr.
  {\IsPrime\Result{#1}%
   \ifnumgreater{\Result}{0}
   {\stepcounter{primecount}%
    \stepcounter{cellcount}%
    \ifnumequal {\value{cellcount}}{7}
       {\the#1 \\\setcounter{cellcount}{0}}
       {\the#1 &}}
   {}%
    \ifnumequal {\value{primecount}}{50}
     {\xintBreakForAndDo
      {\multicolumn {6}{l|}{These are the first 50 primes after 12345.}\\}}
     {}%
  }\hline
\end{tabular}
\end{figure*}
```

23.22 Some arithmetic with Fibonacci numbers

Here is again the code employed on the title page to compute Fibonacci numbers:

```
\def\Fibonacci #1{%  \Fibonacci{N} computes F(N) with F(0)=0, F(1)=1.
    \expandafter\Fibonacci_a\expandafter
        {\the\numexpr #1\expandafter}\expandafter
        {\romannumeral0\xintiieval 1\expandafter\relax\expandafter}\expandafter
        {\romannumeral0\xintiieval 1\expandafter\relax\expandafter}\expandafter
        {\romannumeral0\xintiieval 1\expandafter\relax\expandafter}\expandafter
        {\romannumeral0\xintiieval 0\relax}}
%
\def\Fibonacci_a #1{%
    \ifcase #1
          \expandafter\Fibonacci_end_i
    \or
          \expandafter\Fibonacci_end_ii
    \else
          \ifodd #1
              \expandafter\expandafter\expandafter\Fibonacci_b_ii
```

```
        \else
            \expandafter\expandafter\expandafter\Fibonacci_b_i
        \fi
    \fi {#1}%
}% * signs are omitted from the next macros, tacit multiplications
\def\Fibonacci_b_i #1#2#3{\expandafter\Fibonacci_a\expandafter
  {\the\numexpr #1/2\expandafter}\expandafter
  {\romannumeral0\xintiieval sqr(#2)+sqr(#3)\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiieval (2#2-#3)#3\relax}%
}% end of Fibonacci_b_i
\def\Fibonacci_b_ii #1#2#3#4#5{\expandafter\Fibonacci_a\expandafter
  {\the\numexpr (#1-1)/2\expandafter}\expandafter
  {\romannumeral0\xintiieval sqr(#2)+sqr(#3)\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiieval (2#2-#3)#3\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiieval #2#4+#3#5\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiieval #2#5+#3(#4-#5)\relax}%
}% end of Fibonacci_b_ii
\def\Fibonacci_end_i  #1#2#3#4#5{\xintthe#5}
\def\Fibonacci_end_ii #1#2#3#4#5{\xinttheiiexpr #2#5+#3(#4-#5)\relax}
\def\Fibonacci_end_i  #1#2#3#4#5{{#4}{#5}}% {F(N+1)}{F(N)} in \xintexpr format
\def\Fibonacci_end_ii #1#2#3#4#5%
    {\expandafter
     {\romannumeral0\xintiieval #2#4+#3#5\expandafter\relax
      \expandafter}\expandafter
     {\romannumeral0\xintiieval #2#5+#3(#4-#5)\relax}}% idem.
% \FibonacciN returns F(N) (in encapsulated format: needs \xintthe for printing)
\def\FibonacciN {\expandafter\xint_secondoftwo\romannumeral-‘0\Fibonacci }%
```

I have modified the ending, as I now want not only one specific value `F(N)` but a pair of successive values which can serve as starting point of another routine devoted to compute a whole sequence `F(N), F(N+1), F(N+2),....` This pair is, for efficiency, kept in the encapsulated internal **xintexpr** format. `\FibonacciN` outputs the single `F(N)`, also as an `\xintexpr`-ession, and printing it will thus need the `\xintthe` prefix.

Here a code snippet which checks the routine via a \message of the first 51 Fibonacci numbers (this is not an efficient way to generate a sequence of such numbers, it is only for validating `\FibonacciN`).

```
\def\Fibo #1.{\xintthe\FibonacciN {#1}}%
\message{\xintiloop [0+1] \expandafter\Fibo\xintiloopindex.,
  \ifnum\xintiloopindex<49 \repeat \xintthe\FibonacciN{50}.}
```

The various `\romannumeral0\xintiieval` could very well all have been `\xintiiexpr`'s but then we would have needed more `\expandafter`'s. Indeed the order of expansion must be controlled for the whole thing to work, and `\romannumeral0\xintiieval` is the first expanded form of `\xintiiexpr`.

The way we use `\expandafter`'s to chain successive `\xintexpr` evaluations is exactly analogous to well-known expandable techniques made possible by `\numexpr`.

> There is a difference though: `\numexpr` is *NOT* expandable, and to force its expansion we must prefix it with `\the` or `\number`. On the other hand `\xintexpr`, `\xintiexpr`, ..., (or `\xinteval`, `\xintieval`, ...) expand fully when prefixed by `\romannumeral-‘0`: the computation is fully executed and its result encapsulated in a private format.
>
> Using `\xintthe` as prefix is necessary to print the result (this is like `\the` for `\numexpr`), but it is not necessary to get the computation done (contrarily to the situation with `\numexpr`).

And, starting with release 1.09j, it is also allowed to expand a non \xintthe prefixed \xintexpr-ession inside an \edef: the private format is now protected, hence the error message complaining about a missing \xintthe will not be executed, and the integrity of the format will be preserved.

This new possibility brings some efficiency gain, when one writes non-expandable algorithms using xintexpr. If \xintthe is employed inside \edef the number or fraction will be un-locked into its possibly hundreds of digits and all these tokens will possibly weigh on the upcoming shuffling of (braced) tokens. The private encapsulated format has only a few tokens, hence expansion will proceed a bit faster.

see footnote[59]

Our \Fibonacci expands completely under *f*-expansion, so we can use \fdef rather than \edef in a situation such as

\fdef \X {\FibonacciN {100}}

but for the reasons explained above, it is as efficient to employ \edef. And if we want

\edef \Y {(\FibonacciN{100},\FibonacciN{200})},

then \edef is necessary.

Allright, so let's now give the code to generate a sequence of braced Fibonacci numbers {F(N)}{F(N+1)}{F(N+2)}..., using \Fibonacci for the first two and then using the standard recursion F(N+2)=F(N+1)+F(N):

```
\catcode`_ 11
\def\FibonacciSeq #1#2{%#1=starting index, #2>#1=ending index
    \expandafter\Fibonacci_Seq\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2-1}%
}%
\def\Fibonacci_Seq #1#2{%
     \expandafter\Fibonacci_Seq_loop\expandafter
                {\the\numexpr #1\expandafter}\romannumeral0\Fibonacci {#1}{#2}%
}%
\def\Fibonacci_Seq_loop #1#2#3#4{% standard Fibonacci recursion
    {#3}\unless\ifnum #1<#4  \Fibonacci_Seq_end\fi
        \expandafter\Fibonacci_Seq_loop\expandafter
        {\the\numexpr #1+1\expandafter}\expandafter
        {\romannumeral0\xintiieval #2+#3\relax}{#2}{#4}%
}%
\def\Fibonacci_Seq_end\fi\expandafter\Fibonacci_Seq_loop\expandafter
    #1\expandafter #2#3#4{\fi {#3}}%
\catcode`_ 8
```

Deliberately and for optimization, this \FibonacciSeq macro is completely expandable but not *f*-expandable. It would be easy to modify it to be so. But I wanted to check that the \xintFor* does apply full expansion to what comes next each time it fetches an item

[59] To be completely honest the examination by TeX of all successive digits was not avoided, as it occurs already in the locking-up of the result, what is avoided is to spend time un-locking, and then have the macros shuffle around possibly hundreds of digit tokens rather than a few control words.Technical note: I decided (somewhat hesitantly) for reasons of optimization purposes to skip in the private \xintexpr format a \protect-ion for the \.=digits/digits[digits] control sequences used internally. Thus in the improbable case that some macro package (such control sequence names are unavailable to the casual user) has given a meaning to one such control sequence, there is a possibility of a crash when embedding an \xintexpr without \xintthe prefix in an \edef (the computations by themselves do proceed perfectly correctly even if these control sequences have acquired some non \relax meaning).

from its list argument. Thus, there is no need to generate lists of braced Fibonacci numbers beforehand, as `\xintFor*`, without using any `\edef`, still manages to generate the list via iterated full expansion.

I initially used only one `\halign` in a three-column `multicols` environment, but `multicols` only knows to divide the page horizontally evenly, thus I employed in the end one `\halign` for each column (I could have then used a `tabular` as no column break was then needed).

30.	832040	0	**60.**	1548008755920	0	**90.**	2880067194370816120	0
31.	1346269	514229	**61.**	2504730781961	1	**91.**	4660046610375530309	514229
32.	2178309	514229	**62.**	4052739537881	1	**92.**	7540113804746346429	514229
33.	3524578	196418	**63.**	6557470319842	2	**93.**	12200160415121876738	196418
34.	5702887	710647	**64.**	10610209857723	3	**94.**	19740274219868223167	710647
35.	9227465	75025	**65.**	17167680177565	5	**95.**	31940434634990099905	75025
36.	14930352	785672	**66.**	27777890035288	8	**96.**	51680708854858323072	785672
37.	24157817	28657	**67.**	44945570212853	13	**97.**	83621143489848422977	28657
38.	39088169	814329	**68.**	72723460248141	21	**98.**	135301852344706746049	814329
39.	63245986	10946	**69.**	117669030460994	34	**99.**	218922995834555169026	10946
40.	102334155	825275	**70.**	190392490709135	55	**100.**	354224848179261915075	825275
41.	165580141	4181	**71.**	308061521170129	89	**101.**	573147844013817084101	4181
42.	267914296	829456	**72.**	498454011879264	144	**102.**	927372692193078999176	829456
43.	433494437	1597	**73.**	806515533049393	233	**103.**	1500520536206896083277	1597
44.	701408733	831053	**74.**	1304969544928657	377	**104.**	2427893228399975082453	831053
45.	1134903170	610	**75.**	2111485077978050	610	**105.**	3928413764606871165730	610
46.	1836311903	831663	**76.**	3416454622906707	987	**106.**	6356306993006846248183	831663
47.	2971215073	233	**77.**	5527939700884757	1597	**107.**	10284720757613717413913	233
48.	4807526976	831896	**78.**	8944394323791464	2584	**108.**	16641027750620563662096	831896
49.	7778742049	89	**79.**	14472334024676221	4181	**109.**	26925748508234281076009	89
50.	12586269025	831985	**80.**	23416728348467685	6765	**110.**	43566776258854844738105	831985
51.	20365011074	34	**81.**	37889062373143906	10946	**111.**	70492524767089125814114	34
52.	32951280099	832019	**82.**	61305790721611591	17711	**112.**	114059301025943970552219	832019
53.	53316291173	13	**83.**	99194853094755497	28657	**113.**	184551825793033096366333	13
54.	86267571272	832032	**84.**	160500643816367088	46368	**114.**	298611126818977066918552	832032
55.	139583862445	5	**85.**	259695496911122585	75025	**115.**	483162952612010163284885	5
56.	225851433717	832037	**86.**	420196140727489673	121393	**116.**	781774079430987230203437	832037
57.	365435296162	2	**87.**	679891637638612258	196418	**117.**	1264937032042997393488322	2
58.	591286729879	832039	**88.**	1100087778366101931	317811	**118.**	2046711111473984623691759	832039
59.	956722026041	1	**89.**	1779979416004714189	514229	**119.**	3311648143516982017180081	1

Some Fibonacci numbers together with their residues modulo `F(30)`=832040

```
\newcounter{index}
\tabskip 1ex
  \fdef\Fibxxx{\FibonacciN {30}}%
  \setcounter{index}{30}%
\vbox{\halign{\bfseries#.\hfil&#\hfil &\hfil #\cr
  \xintFor* #1 in {\FibonacciSeq {30}{59}}\do
  {\theindex &\xintthe#1 &
    \xintRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{index}\cr }}%
}\vrule
```

```
\vbox{\halign{\bfseries#.\hfil&#\hfil &\hfil #\cr
  \xintFor* #1 in {\FibonacciSeq {60}{89}}\do
  {\theindex &\xintthe#1 &
    \xintRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{index}\cr }}%
}\vrule
\vbox{\halign{\bfseries#.\hfil&#\hfil &\hfil #\cr
  \xintFor* #1 in {\FibonacciSeq {90}{119}}\do
  {\theindex &\xintthe#1 &
   \xintRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{index}\cr }}%
}%
```

This produces the Fibonacci numbers from `F(30)` to `F(119)`, and computes also all the congruence classes modulo `F(30)`. The output has been put in a float, which appears on the preceding page. I leave to the mathematically inclined readers the task to explain the visible patterns. . . `;-)`.

23.23 `\xintForpair`, `\xintForthree`, `\xintForfour`

on The syntax is illustrated in this example. The notation is the usual one for `n`-uples, with parentheses and commas. Spaces around commas and parentheses are ignored.

```
\begin{tabular}{cccc}
    \xintForpair #1#2 in { ( A , a ) , ( B , b ) , ( C , c ) } \do {%
      \xintForpair #3#4 in { ( X , x ) , ( Y , y ) , ( Z , z ) } \do {%
        $\Biggl($\begin{tabular}{cc}
          -#1- & -#3-\\
          -#4- & -#2-\\
        \end{tabular}$\Biggr)$&}\\\noalign{\vskip1\jot}}%
\end{tabular}
```

$$\begin{array}{ccc}
\begin{pmatrix}\text{-A-} & \text{-X-}\\ \text{-x-} & \text{-a-}\end{pmatrix} & \begin{pmatrix}\text{-A-} & \text{-Y-}\\ \text{-y-} & \text{-a-}\end{pmatrix} & \begin{pmatrix}\text{-A-} & \text{-Z-}\\ \text{-z-} & \text{-a-}\end{pmatrix}\\
\begin{pmatrix}\text{-B-} & \text{-X-}\\ \text{-x-} & \text{-b-}\end{pmatrix} & \begin{pmatrix}\text{-B-} & \text{-Y-}\\ \text{-y-} & \text{-b-}\end{pmatrix} & \begin{pmatrix}\text{-B-} & \text{-Z-}\\ \text{-z-} & \text{-b-}\end{pmatrix}\\
\begin{pmatrix}\text{-C-} & \text{-X-}\\ \text{-x-} & \text{-c-}\end{pmatrix} & \begin{pmatrix}\text{-C-} & \text{-Y-}\\ \text{-y-} & \text{-c-}\end{pmatrix} & \begin{pmatrix}\text{-C-} & \text{-Z-}\\ \text{-z-} & \text{-c-}\end{pmatrix}
\end{array}$$

Only `#1#2`, `#2#3`, `#3#4`, . . . , `#8#9` are valid (no error check is done on the input syntax, `#1#3` or similar all end up in errors). One can nest with `\xintFor`, for disjoint sets of macro parameters. There is also `\xintForthree` (from `#1#2#3` to `#7#8#9`) and `\xintForfour` (from `#1#2#3#4` to `#6#7#8#9`). `\par` tokens are accepted in both the comma separated list and the replacement text.

23.24 `\xintAssign`

`\xintAssign`⟨*braced things*⟩`\to`⟨*as many cs as they are things*⟩ defines (without checking if something gets overwritten) the control sequences on the right of `\to` to expand to the successive tokens or braced items found one after the otehr on the on the left of `\to`. It is not expandable.

A ‘full’ expansion is first applied to the material in front of `\xintAssign`, which may thus be a macro expanding to a list of braced items.

Special case: if after this initial expansion no brace is found immediately after \xintAssign, it is assumed that there is only one control sequence following \to, and this control sequence is then defined via \def to expand to the material between \xintAssign and \to. Other types of expansions are specified through an optional parameter to \xintAssign, see *infra*.

\xintAssign \xintDivision{1000000000000}{133333333}\to\Q\R

\meaning\Q: macro:->7500, \meaning\R: macro:->2500

\xintAssign \xintiPow {7}{13}\to\SevenToThePowerThirteen

\SevenToThePowerThirteen=96889010407

(same as \edef\SevenToThePowerThirteen{\xintiPow {7}{13}})

Changed! → \xintAssign admits since 1.09i an optional parameter, for example \xintAssign [e]... or \xintAssign [oo] .... The latter means that the definitions of the macros initially on the right of \to will be made with \oodef which expands twice the replacement text. The default is simply to make the definitions with \def, corresponding to an empty optional paramter []. Possibilities: [], [g], [e], [x], [o], [go], [oo], [goo], [f], [gf].

In all cases, recall that \xintAssign starts with an *f*-expansion of what comes next; this produces some list of tokens or braced items, and the optional parameter only intervenes to decide the expansion type to be applied then to each one of these items.

Note: prior to release 1.09j, \xintAssign did an \edef by default, but it now does \def. Use the optional parameter [e] to force use of \edef.

23.25 \xintAssignArray

\xintAssignArray⟨*braced things*⟩\to\myArray first expands fully what comes immediately after \xintAssignArray and expects to find a list of braced things {A}{B}... (or tokens). It then defines \myArray as a macro with one parameter, such that \myArray{x} expands to give the xth braced thing of this original list (the argument {x} itself is fed to a \numexpr by \myArray, and \myArray expands in two steps to its output). With 0 as parameter, \myArray{0} returns the number M of elements of the array so that the successive elements are \myArray{1}, ..., \myArray{M}.

\xintAssignArray \xintBezout {1000}{113}\to\Bez

will set \Bez{0} to 5, \Bez{1} to 1000, \Bez{2} to 113, \Bez{3} to -20, \Bez{4} to -177, and \Bez{5} to 1: $(-20)\times 1000-(-177)\times 113=1$. This macro is incompatible with expansion-only contexts.

Changed! → \xintAssignArray admits now an optional parameter, for example \xintAssignArray [e].... This means that the definitions of the macros will be made with \edef. The default is [], which makes the definitions with \def. Other possibilities: [], [o], [oo], [f]. Contrarily to \xintAssign one can not use the g here to make the definitions global. For this, one should rather do \xintAssignArray within a group starting with \globaldefs 1.

Note that prior to release 1.09j each item (token or braced material) was submitted to an \edef, but the default is now to use \def.

23.26 \xintRelaxArray

\xintRelaxArray\myArray (globally) sets to \relax all macros which were defined by the previous \xintAssignArray with \myArray as array macro.

23.27 \odef, \oodef, \fdef

\oodef\controlsequence {<stuff>} does

```
\expandafter\expandafter\expandafter\def
\expandafter\expandafter\expandafter\controlsequence
\expandafter\expandafter\expandafter{<stuff>}
```

This works only for a single \controlsequence, with no parameter text, even without parameters. An alternative would be:

```
\def\oodef #1#{\def\oodefparametertext{#1}%
               \expandafter\expandafter\expandafter\expandafter
               \expandafter\expandafter\expandafter\def
               \expandafter\expandafter\expandafter\oodefparametertext
               \expandafter\expandafter\expandafter }
```

but it does not allow \global as prefix, and, besides, would have anyhow its use (almost) limited to parameter texts without macro parameter tokens (except if the expanded thing does not see them, or is designed to deal with them).

There is a similar macro \odef with only one expansion of the replacement text <stuff>, and \fdef which expands fully <stuff> using \romannumeral-`0.

These tools are provided as it is sometimes wasteful (from the point of view of running time) to do an \edef when one knows that the contents expand in only two steps for example, as is the case with all (except \xintloop and \xintiloop) the expandable macros of the xint packages. Each will be defined only if xinttools finds them currently undefined. They can be prefixed with \global.

23.28 The Quick Sort algorithm illustrated

First a completely expandable macro which sorts a list of numbers. The \QSfull macro expands its list argument, which may thus be a macro; its items must expand to possibly big integers (or also decimal numbers or fractions if using xintfrac), but if an item is expressed as a computation, this computation will be redone each time the item is considered! If the numbers have many digits (i.e. hundreds of digits...), the expansion of \QSfull is fastest if each number, rather than being explicitely given, is represented as a single token which expands to it in one step.

If the interest is only in TeX integers, then one should replace the macros \QSMore, QSEqual, QSLess with versions using the etoolbox (LaTeX only) \ifnumgreater, \ifnumequal and \ifnumless conditionals rather than \xintifGt, \xintifEq, \xintifLt.

```
% THE QUICK SORT ALGORITHM EXPANDABLY
\input xintfrac.sty
% HELPER COMPARISON MACROS
\def\QSMore  #1#2{\xintifGt {#2}{#1}{{#2}}{ }}
% the spaces are there to stop the \romannumeral-`0 originating
% in \xintapplyunbraced when it applies a macro to an item
\def\QSEqual #1#2{\xintifEq {#2}{#1}{{#2}}{ }}
\def\QSLess  #1#2{\xintifLt {#2}{#1}{{#2}}{ }}
%
\makeatletter
```

```
\def\QSfull   {\romannumeral0\qsfull }
\def\qsfull   #1{\expandafter\qsfull@a\expandafter{\romannumeral-`0#1}}
\def\qsfull@a #1{\expandafter\qsfull@b\expandafter {\xintLength {#1}}{#1}}
\def\qsfull@b #1{\ifcase #1
                    \expandafter\qsfull@empty
                 \or\expandafter\qsfull@single
                 \else\expandafter\qsfull@c
                 \fi
}%
\def\qsfull@empty  #1{ }  % the space stops the \QSfull \romannumeral0
\def\qsfull@single #1{ #1}
% for simplicity of implementation, we pick up the first item as pivot
\def\qsfull@c #1{\qsfull@ci #1\undef {#1}}
\def\qsfull@ci #1#2\undef {\qsfull@d {#1}}% #3 is the list, #1 its first item
\def\qsfull@d #1#2{\expandafter\qsfull@e\expandafter
                   {\romannumeral0\qsfull
                       {\xintApplyUnbraced {\QSMore {#1}}{#2}}}%
                   {\romannumeral0\xintapplyunbraced {\QSEqual {#1}}{#2}}%
                   {\romannumeral0\qsfull
                       {\xintApplyUnbraced {\QSLess {#1}}{#2}}}%
}%
\def\qsfull@e #1#2#3{\expandafter\qsfull@f\expandafter {#2}{#3}{#1}}%
\def\qsfull@f #1#2#3{\expandafter\space #2#1#3}
\makeatother
% EXAMPLE
\edef\z {\QSfull {{1.0}{0.5}{0.3}{1.5}{1.8}{2.0}{1.7}{0.4}{1.2}{1.4}%
               {1.3}{1.1}{0.7}{1.6}{0.6}{0.9}{0.8}{0.2}{0.1}{1.9}}}
\tt\meaning\z
\def\a {3.123456789123456789}\def\b {3.123456789123456788}
\def\c {3.123456789123456790}\def\d {3.123456789123456787}
\expandafter\def\expandafter\z\expandafter
  {\romannumeral0\qsfull {{\a}\b\c\d}}% \a is braced to not be expanded
\meaning\z
```

Output:

`macro:->{0.1}{0.2}{0.3}{0.4}{0.5}{0.6}{0.7}{0.8}{0.9}{1.0}{1.1}{1.2}{1.3}{1.4}{1.5}{1.6}{1.7}{1.8}{1.9}{2.0}`

`macro:->{\d}{\b}{\a}{\c}`

We then turn to a graphical illustration of the algorithm. For simplicity the pivot is always chosen to be the first list item. We also show later how to illustrate the variant which picks up the last item of each unsorted chunk as pivot.

```
\input xintfrac.sty % if Plain TeX
%
\definecolor{LEFT}{RGB}{216,195,88}
\definecolor{RIGHT}{RGB}{208,231,153}
\definecolor{INERT}{RGB}{199,200,194}
\definecolor{PIVOT}{RGB}{109,8,57}
%
\def\QSMore  #1#2{\xintifGt {#2}{#1}{{#2}}{ }}% space will be gobbled
\def\QSEqual #1#2{\xintifEq {#2}{#1}{{#2}}{ }}
\def\QSLess  #1#2{\xintifLt {#2}{#1}{{#2}}{ }}
```

```
%
\makeatletter
\def\QS@a  #1{\expandafter \QS@b \expandafter {\xintLength {#1}}{#1}}
\def\QS@b #1{\ifcase #1
                      \expandafter\QS@empty
                   \or\expandafter\QS@single
                 \else\expandafter\QS@c
                 \fi
}%
\def\QS@empty  #1{}
\def\QS@single #1{\QSIr {#1}}
\def\QS@c #1{\QS@d #1!{#1}}    % we pick up the first as pivot.
\def\QS@d #1#2!{\QS@e {#1}}% #1 = first element, #3 = list
\def\QS@e #1#2{\expandafter\QS@f\expandafter
                   {\romannumeral0\xintapplyunbraced {\QSMore  {#1}}{#2}}%
                   {\romannumeral0\xintapplyunbraced {\QSEqual {#1}}{#2}}%
                   {\romannumeral0\xintapplyunbraced {\QSLess  {#1}}{#2}}%
}%
\def\QS@f #1#2#3{\expandafter\QS@g\expandafter {#2}{#3}{#1}}%
% Here \QSLr, \QSIr, \QSr have been let to \relax, so expansion stops.
% #2= elements < pivot, #1 = elements = pivot, #3 = elements > pivot
\def\QS@g #1#2#3{\QSLr {#2}\QSIr {#1}\QSRr {#3}}%
%
\def\DecoLEFT   #1{\xintFor* ##1 in {#1} \do {\colorbox{LEFT}{##1}}}
\def\DecoINERT  #1{\xintFor* ##1 in {#1} \do {\colorbox{INERT}{##1}}}
\def\DecoRIGHT  #1{\xintFor* ##1 in {#1} \do {\colorbox{RIGHT}{##1}}}
\def\DecoPivot #1{\begingroup\color{PIVOT}\advance\fboxsep-\fboxrule
                            \fbox{#1}\endgroup}
\def\DecoLEFTwithPivot #1{%
     \xintFor* ##1 in {#1} \do
     {\xintifForFirst {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}%
}
\def\DecoRIGHTwithPivot #1{%
     \xintFor* ##1 in {#1} \do
     {\xintifForFirst {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}%
}
%
\def\QSinitialize #1{\def\QS@list{\QSRr {#1}}%
                     \let\QSRr\DecoRIGHT
%                     \QS@list \par
\par\centerline{\QS@list}
}
\def\QSoneStep {\let\QSLr\DecoLEFTwithPivot
                \let\QSIr\DecoINERT
                \let\QSRr\DecoRIGHTwithPivot
%                    \QS@list
\centerline{\QS@list}
%                \par
                \def\QSLr {\noexpand\QS@a}%
                \let\QSIr\relax
                \def\QSRr {\noexpand\QS@a}%
                    \edef\QS@list{\QS@list}%
```

```
                \let\QSLr\relax
                \let\QSRr\relax
                    \edef\QS@list{\QS@list}%
                \let\QSLr\DecoLEFT
                \let\QSIr\DecoINERT
                \let\QSRr\DecoRIGHT
%                    \QS@list
\centerline{\QS@list}
%                \par
}
\begingroup\offinterlineskip
\small
\QSinitialize {{1.0}{0.5}{0.3}{1.5}{1.8}{2.0}{1.7}{0.4}{1.2}{1.4}%
               {1.3}{1.1}{0.7}{1.6}{0.6}{0.9}{0.8}{0.2}{0.1}{1.9}}
\QSoneStep
\QSoneStep
\QSoneStep
\QSoneStep
\QSoneStep
\endgroup
```

1.0	0.5	0.3	1.5	1.8	2.0	1.7	0.4	1.2	1.4	1.3	1.1	0.7	1.6	0.6	0.9	0.8	0.2	0.1	1.9
$\boxed{1.0}$	0.5	0.3	1.5	1.8	2.0	1.7	0.4	1.2	1.4	1.3	1.1	0.7	1.6	0.6	0.9	0.8	0.2	0.1	1.9
0.5	0.3	0.4	0.7	0.6	0.9	0.8	0.2	0.1	1.0	1.5	1.8	2.0	1.7	1.2	1.4	1.3	1.1	1.6	1.9
$\boxed{0.5}$	0.3	0.4	0.7	0.6	0.9	0.8	0.2	0.1	1.0	$\boxed{1.5}$	1.8	2.0	1.7	1.2	1.4	1.3	1.1	1.6	1.9
0.3	0.4	0.2	0.1	0.5	0.7	0.6	0.9	0.8	1.0	1.2	1.4	1.3	1.1	1.5	1.8	2.0	1.7	1.6	1.9
$\boxed{0.3}$	0.4	0.2	0.1	0.5	$\boxed{0.7}$	0.6	0.9	0.8	1.0	$\boxed{1.2}$	1.4	1.3	1.1	1.5	$\boxed{1.8}$	2.0	1.7	1.6	1.9
0.2	0.1	0.3	0.4	0.5	0.6	0.7	0.9	0.8	1.0	1.1	1.2	1.4	1.3	1.5	1.7	1.6	1.8	2.0	1.9
$\boxed{0.2}$	0.1	0.3	$\boxed{0.4}$	0.5	$\boxed{0.6}$	0.7	$\boxed{0.9}$	0.8	1.0	$\boxed{1.1}$	1.2	$\boxed{1.4}$	1.3	1.5	$\boxed{1.7}$	1.6	1.8	$\boxed{2.0}$	1.9
0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0
$\boxed{0.1}$	0.2	0.3	0.4	0.5	0.6	0.7	$\boxed{0.8}$	0.9	1.0	1.1	1.2	$\boxed{1.3}$	1.4	1.5	$\boxed{1.6}$	1.7	1.8	$\boxed{1.9}$	2.0
0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0

If one wants rather to have the pivot from the end of the yet to sort chunks, then one should use the following variants:

```
\def\QS@c #1{\expandafter\QS@e\expandafter
                   {\romannumeral0\xintnthelt {-1}{#1}}{#1}%
}%
\def\DecoLEFTwithPivot #1{%
     \xintFor* ##1 in {#1} \do
     {\xintifForLast {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}%
}
\def\DecoRIGHTwithPivot #1{%
     \xintFor* ##1 in {#1} \do
     {\xintifForLast {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}%
}
\def\QSinitialize #1{\def\QS@list{\QSLr {#1}}%
                     \let\QSLr\DecoLEFT
%                     \QS@list \par
\par\centerline{\QS@list}
}
```

1.0	0.5	0.3	1.5	1.8	2.0	1.7	0.4	1.2	1.4	1.3	1.1	0.7	1.6	0.6	0.9	0.8	0.2	0.1	1.9

1.0	0.5	0.3	1.5	1.8	2.0	1.7	0.4	1.2	1.4	1.3	1.1	0.7	1.6	0.6	0.9	0.8	0.2	0.1	1.9
1.0	0.5	0.3	1.5	1.8	1.7	0.4	1.2	1.4	1.3	1.1	0.7	1.6	0.6	0.9	0.8	0.2	0.1	1.9	2.0
1.0	0.5	0.3	1.5	1.8	1.7	0.4	1.2	1.4	1.3	1.1	0.7	1.6	0.6	0.9	0.8	0.2	0.1	1.9	2.0
0.1	1.0	0.5	0.3	1.5	1.8	1.7	0.4	1.2	1.4	1.3	1.1	0.7	1.6	0.6	0.9	0.8	0.2	1.9	2.0
0.1	1.0	0.5	0.3	1.5	1.8	1.7	0.4	1.2	1.4	1.3	1.1	0.7	1.6	0.6	0.9	0.8	0.2	1.9	2.0
0.1	0.2	1.0	0.5	0.3	1.5	1.8	1.7	0.4	1.2	1.4	1.3	1.1	0.7	1.6	0.6	0.9	0.8	1.9	2.0
0.1	0.2	1.0	0.5	0.3	1.5	1.8	1.7	0.4	1.2	1.4	1.3	1.1	0.7	1.6	0.6	0.9	0.8	1.9	2.0
0.1	0.2	0.5	0.3	0.4	0.7	0.6	0.8	1.0	1.5	1.8	1.7	1.2	1.4	1.3	1.1	1.6	0.9	1.9	2.0
0.1	0.2	0.5	0.3	0.4	0.7	0.6	0.8	1.0	1.5	1.8	1.7	1.2	1.4	1.3	1.1	1.6	0.9	1.9	2.0
0.1	0.2	0.5	0.3	0.4	0.6	0.7	0.8	0.9	1.0	1.5	1.8	1.7	1.2	1.4	1.3	1.1	1.6	1.9	2.0
0.1	0.2	0.5	0.3	0.4	0.6	0.7	0.8	0.9	1.0	1.5	1.8	1.7	1.2	1.4	1.3	1.1	1.6	1.9	2.0
0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.5	1.2	1.4	1.3	1.1	1.6	1.8	1.7	1.9	2.0
0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.5	1.2	1.4	1.3	1.1	1.6	1.8	1.7	1.9	2.0
0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.5	1.2	1.4	1.3	1.6	1.7	1.8	1.9	2.0
0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.5	1.2	1.4	1.3	1.6	1.7	1.8	1.9	2.0
0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.5	1.4	1.6	1.7	1.8	1.9	2.0
0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.5	1.4	1.6	1.7	1.8	1.9	2.0
0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0
0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0
0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0

It is possible to modify this code to let it do `\QSonestep` repeatedly and stop automatically when the sort is finished.[60]

24 Commands of the xint package

In the description of the macros `{N}` and `{M}` stand for (long) numbers within braces or for a control sequence possibly within braces and *f*-expanding to such a number (without the braces!), or for material within braces which *f*-expands to such a number, as is acceptable on input by the `\xintNum` macro: a sequence of plus and minus signs, followed by some string of zeroes, followed by digits. The margin annotation for such an argument which is parsed by `\xintNum` is $\overset{\text{Num}}{f}$. Sometimes however only a *f* symbol appears in the margin, signaling that the input will not be parsed via `\xintNum`.

The letter `x` (with margin annotation $\overset{\text{num}}{x}$) stands for something which will be inserted in-between a `\numexpr` and a `\relax`. It will thus be completely expanded and must give an integer obeying the TeX bounds. Thus, it may be for example a count register, or itself a `\numexpr` expression, or just a number written explicitely with digits or something like `4*\count 255 + 17`, etc...

For the rules regarding direct use of count registers or `\numexpr` expression, in the argument to the package macros, see the Use of count section.

Some of these macros are extended by **xintfrac** to accept fractions on input, and, generally, to output a fraction. But this means that additions, subtractions, multiplications output in fraction format; to guarantee the integer format on output when the inputs are integers, the original integer-only macros `\xintAdd`, `\xintSub`, `\xintMul`, etc... are available under the names `\xintiAdd`, `\xintiSub`, `\xintiMul`, ..., also when **xintfrac** is not loaded. Even these originally integer-only macros will accept fractions on input if **xint-**

60 http://tex.stackexchange.com/a/142634/4686

frac is loaded as long as they are integers in disguise; they produce on output integers without any forward slash mark nor trailing [n].

But \xintAdd will output fractions A/B[n], with B present even if its value is one. See the xintfrac documentation for additional information.

Contents

24.1 \xintRev

f ⋆ \xintRev{N} will revert the order of the digits of the number, keeping the optional sign. Leading zeroes resulting from the operation are not removed (see the \xintNum macro for this). This macro and all other macros dealing with numbers first expand 'fully' their arguments.

\xintRev{-123000}=-000321
\xintNum{\xintRev{-123000}}=-321

24.2 \xintLen

Num
f ★ \xintLen{N} returns the length of the number, not counting the sign.

\xintLen{-12345678901234567890123456789}=29

Extended by **xintfrac** to fractions: the length of A/B[n] is the length of A plus the length of B plus the absolute value of n and minus one (an integer input as N is internally represented in a form equivalent to N/1[0] so the minus one means that the extended \xintLen behaves the same as the original for integers).

\xintLen{-1e3/5.425}=10

The length is computed on the A/B[n] which would have been returned by \xintRaw: \xintRaw {-1e3/5.425}=-1/5425[6].

Let's point out that the whole thing should sum up to less than circa 2^{31}, but this is a bit theoretical.

\xintLen is only for numbers or fractions. See \xintLength for counting tokens (or rather braced groups), more generally.

24.3 \xintDigitsOf

fN This is a synonym for \xintAssignArray, to be used to define an array giving all the digits of a given (positive, else the minus sign will be treated as first item) number.

\xintDigitsOf\xintiPow {7}{500}\to\digits

7^500 has \digits{0}=423 digits, and the 123rd among them (starting from the most significant) is \digits{123}=3.

24.4 \xintNum

f ★ \xintNum{N} removes chains of plus or minus signs, followed by zeroes.

\xintNum{+---++----+--000000000367941789479}=-367941789479

Extended by **xintfrac** to accept also a fraction on input, as long as it reduces to an integer after division of the numerator by the denominator.

\xintNum{123.48/-0.03}=-4116

24.5 \xintSgn

Num
f ★ \xintSgn{N} returns 1 if the number is positive, 0 if it is zero and -1 if it is negative.
f ★ Extended by **xintfrac** to fractions. \xintiiSgn skips the \xintNum overhead.

24.6 \xintOpp

Num
f ★ \xintOpp{N} return the opposite -N of the number N. Extended by **xintfrac** to fractions. \xintiOpp is a synonym not modified by **xintfrac**[61], and \xintiiOpp skips the \xint-
f ★ Num overhead.

[61] here, and in all similar instances, this means that the macro remains integer-only both on input and output, but it does accept on input a fraction which in disguise is a (big) integer.

24.7 \xintAbs

Num f ★ `\xintAbs{N}` returns the absolute value of the number. Extended by **xintfrac** to fractions. `\xintiAbs` is a synonym not modified by **xintfrac**, and `\xintiiAbs` skips the `\xintNum` overhead. f ★

24.8 \xintAdd

Num Num f f ★ `\xintAdd{N}{M}` returns the sum of the two numbers. Extended by **xintfrac** to fractions. `\xintiAdd` is a synonym not modified by **xintfrac**, and `\xintiiAdd` skips the `\xintNum` overhead. ff ★

24.9 \xintSub

Num Num f f ★ `\xintSub{N}{M}` returns the difference `N-M`. Extended by **xintfrac** to fractions. `\xintiSub` is a synonym not modified by **xintfrac**, and `\xintiiSub` skips the `\xintNum` overhead. ff ★

24.10 \xintCmp

Num Num f f ★ `\xintCmp{N}{M}` returns 1 if `N>M`, 0 if `N=M`, and -1 if `N<M`. Extended by **xintfrac** to fractions.

24.11 \xintEq

Num Num f f ★ `\xintEq{N}{M}` returns 1 if `N=M`, 0 otherwise. Extended by **xintfrac** to fractions.

24.12 \xintGt

Num Num f f ★ `\xintGt{N}{M}` returns 1 if `N>M`, 0 otherwise. Extended by **xintfrac** to fractions.

24.13 \xintLt

Num Num f f ★ `\xintLt{N}{M}` returns 1 if `N<M`, 0 otherwise. Extended by **xintfrac** to fractions.

24.14 \xintIsZero

Num f ★ `\xintIsZero{N}` returns 1 if `N=0`, 0 otherwise. Extended by **xintfrac** to fractions.

24.15 \xintNot

Num f ★ `\xintNot` is a synonym for `\xintIsZero`.

24.16 \xintIsNotZero

Num f ★ `\xintIsNotZero{N}` returns 1 if `N<>0`, 0 otherwise. Extended by **xintfrac** to fractions.

24.17 \xintIsOne

Num f ★ `\xintIsOne{N}` returns 1 if `N=1`, 0 otherwise. Extended by **xintfrac** to fractions.

24.18 \xintAND

Num Num
f f ★ \xintAND{N}{M} returns 1 if N<>0 and M<>0 and zero otherwise. Extended by **xintfrac** to fractions.

24.19 \xintOR

Num Num
f f ★ \xintOR{N}{M} returns 1 if N<>0 or M<>0 and zero otherwise. Extended by **xintfrac** to fractions.

24.20 \xintXOR

Num Num
f f ★ \xintXOR{N}{M} returns 1 if exactly one of N or M is true (i.e. non-zero). Extended by **xintfrac** to fractions.

24.21 \xintANDof

Num
$f \rightarrow *$ f ★ \xintANDof{{a}{b}{c}...} returns 1 if all are true (i.e. non zero) and zero otherwise. The list argument may be a macro, it (or rather its first token) is f-expanded first (each item also is f-expanded). Extended by **xintfrac** to fractions.

24.22 \xintORof

Num
$f \rightarrow *$ f ★ \xintORof{{a}{b}{c}...} returns 1 if at least one is true (i.e. does not vanish). The list argument may be a macro, it is f-expanded first. Extended by **xintfrac** to fractions.

24.23 \xintXORof

Num
$f \rightarrow *$ f ★ \xintXORof{{a}{b}{c}...} returns 1 if an odd number of them are true (i.e. does not vanish). The list argument may be a macro, it is f-expanded first. Extended by **xintfrac** to fractions.

24.24 \xintGeq

Num Num
f f ★ \xintGeq{N}{M} returns 1 if the *absolute value* of the first number is at least equal to the absolute value of the second number. If |N|<|M| it returns 0. Extended by **xintfrac** to fractions. Please note that the macro compares *absolute values*.

24.25 \xintMax

Num Num
f f ★ \xintMax{N}{M} returns the largest of the two in the sense of the order structure on the relative integers (*i.e.* the right-most number if they are put on a line with positive numbers on the right): \xintiMax {-5}{-6}=-5. Extended by **xintfrac** to fractions. \xintiMax is a synonym not modified by **xintfrac**.

24.26 \xintMaxof

$f \to * \overset{\text{Num}}{f}$ ★ \xintMaxof{{a}{b}{c}...} returns the maximum. The list argument may be a macro, it is f-expanded first. Extended by **xintfrac** to fractions. \xintiMaxof is a synonym not modified by **xintfrac**.

24.27 \xintMin

$\overset{\text{Num}}{f}\overset{\text{Num}}{f}$ ★ \xintMin{N}{M} returns the smallest of the two in the sense of the order structure on the relative integers (*i.e.* the left-most number if they are put on a line with positive numbers on the right): \xintiMin {-5}{-6}=-6. Extended by **xintfrac** to fractions. \xintiMin is a synonym not modified by **xintfrac**.

24.28 \xintMinof

$f \to * \overset{\text{Num}}{f}$ ★ \xintMinof{{a}{b}{c}...} returns the minimum. The list argument may be a macro, it is f-expanded first. Extended by **xintfrac** to fractions. \xintiMinof is a synonym not modified by **xintfrac**.

24.29 \xintSum

$*f$ ★ \xintSum{⟨*braced things*⟩} after expanding its argument expects to find a sequence of tokens (or braced material). Each is expanded (with the usual meaning), and the sum of all these numbers is returned. Note: the summands are *not* parsed by \xintNum.

\xintSum is extended by **xintfrac** to fractions. The original, which accepts (after f-expansion) only (big) integers in the strict format and produces a (big) integer is available as \xintiiSum, also with **xintfrac** loaded.

\xintiiSum{{123}{-98763450}{\xintFac{7}}{\xintiMul{3347}{591}}}=-96780210

\xintiiSum{1234567890}=45

An empty sum is no error and returns zero: \xintiiSum {}=0. A sum with only one term returns that number: \xintiiSum {{-1234}}=-1234. Attention that \xintiiSum {-1234} is not legal input and will make the TeX run fail. On the other hand \xintiiSum {1234}=10. Extended by **xintfrac** to fractions.

24.30 \xintMul

$\overset{\text{Num}}{f}\overset{\text{Num}}{f}$ ★ \xintMul{N}{M} returns the product of the two numbers. Extended by **xintfrac** to fractions. \xintiMul is a synonym not modified by **xintfrac**, and \xintiiMul skips the
ff ★ \xintNum overhead.

24.31 \xintSqr

$\overset{\text{Num}}{f}$ ★ \xintSqr{N} returns the square. Extended by **xintfrac** to fractions. \xintiSqr is a synonym not modified by **xintfrac**, and \xintiiSqr skips the \xintNum overhead.
f ★

24.32 \xintPrd

$*f$ ★ \xintPrd{⟨*braced things*⟩} after expanding its argument expects to find a sequence of (of braced items or unbraced single tokens). Each is expanded (with the usual meaning), and the product of all these numbers is returned. Note: the operands are *not* parsed by \xintNum.

\xintiiPrd{{-9876}{\xintFac{7}}{\xintiMul{3347}{591}}}=-98458861798080

\xintiiPrd{123456789123456789}=131681894400

An empty product is no error and returns 1: \xintiiPrd {}=1. A product reduced to a single term returns this number: \xintiiPrd {{-1234}}=-1234. Attention that \xintiiPrd {-1234} is not legal input and will make the TeX compilation fail. On the other hand \xintiiPrd {1234}=24.

$$2^{200}3^{100}7^{100}$$

=\xintiiPrd {{\xintiPow {2}{200}}{\xintiPow {3}{100}}{\xintiPow {7}{100}}}
=2678727931661577575766279517007548402324740266374015348974459614815
42641296549949000044400724076572713000016531207640654562118014357199
4015903343539244028212438966822248927862988084382716133376

With **xintexpr**, the above could be coded simply as

\xinttheiiexpr 2^200*3^100*7^100\relax

Extended by **xintfrac** to fractions. The original, which accepts (after *f*-expansion) only (big) integers in the strict format and produces a (big) integer is available as \xintiiPrd, also with **xintfrac** loaded.

24.33 \xintPow

Num num
f x ★
Changed! → \xintPow{N}{x} returns N^x. When x is zero, this is 1. If N is zero and x<0, if |N|>1 and x<0 negative, or if |N|>1 and x>100000, then an error is raised. Indeed 2^50000 already has 15052 digits; each exact multiplication of two one thousand digits numbers already takes a few seconds, and it would take hours for the expandable computation to conclude with two numbers with each circa 15000 digits. Perhaps some completely expandable but not *f*-expandable variants could fare better?

Extended by **xintfrac** to fractions (\xintPow) and to floats (\xintFloatPow for which the exponent must still obey the TeX bound and \xintFloatPower which has no restriction at all on the size of the exponent). Negative exponents do not then cause errors anymore. The float version is able to deal with things such as 2^999999999 without any problem. For example \xintFloatPow[4]{2}{50000}=3.161e15051 and \xintFloatPow[4]{2}{999999999} =2.306e301029995.[62]

num
f x ★
\xintiPow is a synonym not modified by **xintfrac**, and \xintiiPow is an integer only variant skipping the \xintNum overhead, it produces the same result as \xintiPow with stricter assumptions on the inputs, and is thus a tiny bit faster.

corr. of the previous doc. → Within an \xintiiexpr..\relax the infix operator ^ is mapped to \xintiiPow; within an \xintexpr-ession it is mapped to \xintPow (as extended by **xintfrac**); in \xintfloatexpr, it is mapped to \xintFloatPower.

[62] On my laptop \xintiiPow{2}{9999} obtains all 3010 digits in about ten or eleven seconds. In contrast, the float versions for 8, 16, 24, or even more significant figures, do their jobs in less than one hundredth of a second (1.09j; we used in the text only four significant digits only for reasons of space, not time.) This is done without log/exp which are not (yet?) implemented in **xintfrac**. The LaTeX3 l3fp package does this with log/exp and is ten times faster, but allows only 16 significant figures and the (exactly represented) floating point numbers must have their exponents limited to ±9999.

24.34 \xintSgnFork

x n n n ★ \xintSgnFork{-1|0|1}{⟨A⟩}{⟨B⟩}{⟨C⟩} expandably chooses to execute either the ⟨A⟩, ⟨B⟩ or ⟨C⟩ code, depending on its first argument. This first argument should be anything expanding to either -1, 0 or 1 (a count register must be prefixed by \the and a \numexpr...\relax also must be prefixed by \the). This utility is provided to help construct expandable macros choosing depending on a condition which one of the package macros to use, or which values to confer to their arguments.

24.35 \xintifSgn

Num f n n n ★ Similar to \xintSgnFork except that the first argument may expand to a (big) integer (or a fraction if **xintfrac** is loaded), and it is its sign which decides which of the three branches is taken. Furthermore this first argument may be a count register, with no \the or \number prefix.

24.36 \xintifZero

Num f n n ★ \xintifZero{⟨N⟩}{⟨IsZero⟩}{⟨IsNotZero⟩} expandably checks if the first mandatory argument N (a number, possibly a fraction if **xintfrac** is loaded, or a macro expanding to one such) is zero or not. It then either executes the first or the second branch. Beware that both branches must be present.

24.37 \xintifNotZero

Num f n n ★ \xintifNotZero{⟨N⟩}{⟨IsNotZero⟩}{⟨IsZero⟩} expandably checks if the first mandatory argument N (a number, possibly a fraction if **xintfrac** is loaded, or a macro expanding to one such) is not zero or is zero. It then either executes the first or the second branch. Beware that both branches must be present.

24.38 \xintifOne

Num f n n ★ \xintifOne{⟨N⟩}{⟨IsOne⟩}{⟨IsNotOne⟩} expandably checks if the first mandatory argument N (a number, possibly a fraction if **xintfrac** is loaded, or a macro expanding to one such) is one or not. It then either executes the first or the second branch. Beware that both branches must be present.

24.39 \xintifTrueAelseB, \xintifFalseAelseB

Num f n n ★ \xintifTrueAelseB{⟨N⟩}{⟨true branch⟩}{⟨false branch⟩} is a synonym for \xintifNotZero.

1. with 1.09i, the synonyms \xintifTrueFalse and \xintifTrue are deprecated and will be removed in next release.
2. These macros have no lowercase versions, use \xintifzero, \xintifnotzero.

Num f n n ★ \xintifFalseAelseB{⟨N⟩}{⟨false branch⟩}{⟨true branch⟩} is a synonym for \xintifZero.

24.40 \xintifCmp

Num Num
f f nnn ★ \xintifCmp{⟨A⟩}{⟨B⟩}{⟨if A<B⟩}{⟨if A=B⟩}{⟨if A>B⟩} compares its arguments and chooses accordingly the correct branch.

24.41 \xintifEq

Num Num
f f nn ★ \xintifEq{⟨A⟩}{⟨B⟩}{⟨YES⟩}{⟨NO⟩} checks equality of its two first arguments (numbers, or fractions if **xintfrac** is loaded) and does the YES or the NO branch.

24.42 \xintifGt

Num Num
f f nn ★ \xintifGt{⟨A⟩}{⟨B⟩}{⟨YES⟩}{⟨NO⟩} checks if $A > B$ and in that case executes the YES branch. Extended to fractions (in particular decimal numbers) by **xintfrac**.

24.43 \xintifLt

Num Num
f f nn ★ \xintifLt{⟨A⟩}{⟨B⟩}{⟨YES⟩}{⟨NO⟩} checks if $A < B$ and in that case executes the YES branch. Extended to fractions (in particular decimal numbers) by **xintfrac**.

> The macros described next are all integer-only on input. With **xintfrac** loaded their argument is first given to \xintNum and may thus be a fraction, as long as it is in fact an integer in disguise.

24.44 \xintifOdd

Num
f nn ★ \xintifOdd{⟨A⟩}{⟨YES⟩}{⟨NO⟩} checks if A is and odd integer and in that case executes the YES branch.

24.45 \xintFac

num
x ★ \xintFac{x} returns the factorial. It is an error if the argument is negative or at least 10^5.

With **xintfrac** loaded, the macro is modified to accept a fraction as argument, as long as this fraction turns out to be an integer: \xintFac {66/3}=1124000727777607680000. \xintiFac is a synonym not modified by the loading of **xintfrac**.

24.46 \xintDivision

Num Num
f f ★ \xintDivision{N}{M} returns {quotient Q}{remainder R}. This is euclidean division: N = QM + R, 0 ≤ R < |M|. So the remainder is always non-negative and the formula N = QM + R always holds independently of the signs of N or M. Division by zero is an error
ff ★ (even if N vanishes) and returns {0}{0}. The variant \xintiiDivision skips the overhead of parsing via \xintNum.

This macro is integer only (with **xintfrac** loaded it accepts fractions on input, but they must be integers in disguise) and not to be confused with the **xintfrac** macro \xintDiv which divides one fraction by another.

24.47 \xintQuo

Num Num f f ★ `\xintQuo{N}{M}` returns the quotient from the euclidean division. When both `N` and `M` are positive one has `\xintQuo{N}{M}=\xintiTrunc {0}{N/M}` (using package **xintfrac**). With **xintfrac** loaded it accepts fractions on input, but they must be integers in disguise.
ff ★ The variant `\xintiiQuo` skips the overhead of parsing via `\xintNum`.

24.48 \xintRem

Num Num f f ★ `\xintRem{N}{M}` returns the remainder from the euclidean division. With **xintfrac** loaded it accepts fractions on input, but they must be integers in disguise. The variant
ff ★ `\xintiiRem` skips the overhead of parsing via `\xintNum`.

24.49 \xintFDg

Num f ★ `\xintFDg{N}` returns the first digit (most significant) of the decimal expansion. The variant
f ★ `\xintiiFDg` skips the overhead of parsing via `\xintNum`.

24.50 \xintLDg

Num f ★ `\xintLDg{N}` returns the least significant digit. When the number is positive, this is the
f ★ same as the remainder in the euclidean division by ten. The variant `\xintiiLDg` skips the overhead of parsing via `\xintNum`.

24.51 \xintMON, \xintMMON

Num f ★ `\xintMON{N}` returns `(-1)^N` and `\xintMMON{N}` returns `(-1)^{N-1}`.

`\xintMON {-280914019374101929}=-1`, `\xintMMON {-280914019374101929}=1`

f ★ The variants `\xintiiMON` and `\xintiiMMON` skip the overhead of parsing via `\xintNum`.

24.52 \xintOdd

Num f ★ `\xintOdd{N}` is 1 if the number is odd and 0 otherwise. The variant `\xintiiOdd` skip the
f ★ overhead of parsing via `\xintNum`.

24.53 \xintiSqrt, \xintiSquareRoot

Num f ★ `\xintiSqrt{N}` returns the largest integer whose square is at most equal to `N`.

```
\xintiSqrt {2000000000000000000000000000000000000}=1414213562373095048
\xintiSqrt {3000000000000000000000000000000000000}=1732050807568877293
\xintiSqrt {\xintDSH {-120}{2}}=
1414213562373095048801688724209698078569671875376948073176679
```

Num f ★ `\xintiSquareRoot{N}` returns `{M}{d}` with `d>0`, `M^2-d=N` and `M` smallest (hence `=1+\xintiSqrt{N}`).

```
\xintAssign\xintiSquareRoot {17000000000000000000000000}\to\A\B
\xintiSub{\xintiSqr\A}\B=\A^2-\B
17000000000000000000000000=4123105625618^2-2799177881924
```

A rational approximation to $\sqrt{N}$ is $M-\frac{d}{2M}$ (this is a majorant and the error is at most `1/2M`; if `N` is a perfect square `k^2` then `M=k+1` and this gives `k+1/(2k+2)`, not `k`).

Package **xintfrac** has `\xintFloatSqrt` for square roots of floating point numbers.

The macros described next are strictly for integer-only arguments. These arguments are *not* filtered via \xintNum.

24.54 \xintInc, \xintDec

f ⋆ \xintInc{N} is N+1 and \xintDec{N} is N-1. These macros remain integer-only, even with xintfrac loaded.

24.55 \xintDouble, \xintHalf

f ⋆ \xintDouble{N} returns 2N and \xintHalf{N} is N/2 rounded towards zero. These macros remain integer-only, even with xintfrac loaded.

24.56 \xintDSL

f ⋆ \xintDSL{N} is decimal shift left, *i.e.* multiplication by ten.

24.57 \xintDSR

f ⋆ \xintDSR{N} is decimal shift right, *i.e.* it removes the last digit (keeping the sign), equivalently it is the closest integer to N/10 when starting at zero.

24.58 \xintDSH

$\overset{\text{num}}{x}$ *f* ⋆ \xintDSH{x}{N} is parametrized decimal shift. When x is negative, it is like iterating \xintDSL |x| times (*i.e.* multiplication by 10^{-x}). When x positive, it is like iterating \DSR x times (and is more efficient), and for a non-negative N this is thus the same as the quotient from the euclidean division by 10^x.

24.59 \xintDSHr, \xintDSx

$\overset{\text{num}}{x}$ *f* ⋆ \xintDSHr{x}{N} expects x to be zero or positive and it returns then a value R which is correlated to the value Q returned by \xintDSH{x}{N} in the following manner:

- if N is positive or zero, Q and R are the quotient and remainder in the euclidean division by 10^x (obtained in a more efficient manner than using \xintDivision),
- if N is negative let Q1 and R1 be the quotient and remainder in the euclidean division by 10^x of the absolute value of N. If Q1 does not vanish, then Q=-Q1 and R=R1. If Q1 vanishes, then Q=0 and R=-R1.
- for x=0, Q=N and R=0.

So one has N = 10^x Q + R if Q turns out to be zero or positive, and N = 10^x Q - R if Q turns out to be negative, which is exactly the case when N is at most -10^x.

$\overset{\text{num}}{x}$ *f* ⋆ \xintDSx{x}{N} for x negative is exactly as \xintDSH{x}{N}, *i.e.* multiplication by 10^{-x}. For x zero or positive it returns the two numbers {Q}{R} described above, each one within braces. So Q is \xintDSH{x}{N}, and R is \xintDSHr{x}{N}, but computed simultaneously.

\xintAssign\xintDSx {-1}{-123456789}\to\M
\meaning\M: macro:->-1234567890.
\xintAssign\xintDSx {-20}{123456789}\to\M
\meaning\M: macro:->12345678900000000000000000000.
\xintAssign\xintDSx {0}{-123004321}\to\Q\R
\meaning\Q: macro:->-123004321, \meaning\R: macro:->0.
\xintDSH {0}{-123004321}=-123004321, \xintDSHr {0}{-123004321}=0
\xintAssign\xintDSx {6}{-123004321}\to\Q\R
\meaning\Q: macro:->-123, \meaning\R: macro:->4321.
\xintDSH {6}{-123004321}=-123, \xintDSHr {6}{-123004321}=4321
\xintAssign\xintDSx {8}{-123004321}\to\Q\R
\meaning\Q: macro:->-1, \meaning\R: macro:->23004321.
\xintDSH {8}{-123004321}=-1, \xintDSHr {8}{-123004321}=23004321
\xintAssign\xintDSx {9}{-123004321}\to\Q\R
\meaning\Q: macro:->0, \meaning\R: macro:->-123004321.
\xintDSH {9}{-123004321}=0, \xintDSHr {9}{-123004321}=-123004321

24.60 \xintDecSplit

num x f ★ \xintDecSplit{x}{N} cuts the number into two pieces (each one within a pair of enclosing braces). First the sign if present is *removed*. Then, for x positive or null, the second piece contains the x least significant digits (*empty* if x=0) and the first piece the remaining digits (*empty* when x equals or exceeds the length of N). Leading zeroes in the second piece are not removed. When x is negative the first piece contains the |x| most significant digits and the second piece the remaining digits (*empty* if |x| equals or exceeds the length of N). Leading zeroes in this second piece are not removed. So the absolute value of the original number is always the concatenation of the first and second piece.

This macro's behavior for N non-negative is final and will not change. I am still hesitant about what to do with the sign of a negative N.

\xintAssign\xintDecSplit {0}{-123004321}\to\L\R
\meaning\L: macro:->123004321, \meaning\R: macro:->.
\xintAssign\xintDecSplit {5}{-123004321}\to\L\R
\meaning\L: macro:->1230, \meaning\R: macro:->04321.
\xintAssign\xintDecSplit {9}{-123004321}\to\L\R
\meaning\L: macro:->, \meaning\R: macro:->123004321.
\xintAssign\xintDecSplit {10}{-123004321}\to\L\R
\meaning\L: macro:->, \meaning\R: macro:->123004321.
\xintAssign\xintDecSplit {-5}{-12300004321}\to\L\R
\meaning\L: macro:->12300, \meaning\R: macro:->004321.
\xintAssign\xintDecSplit {-11}{-12300004321}\to\L\R
\meaning\L: macro:->12300004321, \meaning\R: macro:->.
\xintAssign\xintDecSplit {-15}{-12300004321}\to\L\R
\meaning\L: macro:->12300004321, \meaning\R: macro:->.

24.61 \xintDecSplitL

num x f ★ \xintDecSplitL{x}{N} returns the first piece after the action of \xintDecSplit.

24.62 \xintDecSplitR

num x f ★ \xintDecSplitR{x}{N} returns the second piece after the action of \xintDecSplit.

25 Commands of the xintfrac package

This package was first included in release 1.03 of the **xint** bundle. The general rule of the bundle that each macro first expands (what comes first, fully) each one of its arguments applies.

Frac f f stands for an integer or a fraction (see subsection 8.1 for the accepted input formats) or something which expands to an integer or fraction. It is possible to use in the numerator or the denominator of f count registers and even expressions with infix arithmetic operators, under some rules which are explained in the previous Use of count registers section.

num x As in the xint.sty documentation, x stands for something which will internally be embedded in a \numexpr. It may thus be a count register or something like 4*\count 255 + 17, etc..., but must expand to an integer obeying the TeX bound.

The fraction format on output is the scientific notation for the 'float' macros, and the A/B[n] format for all other fraction macros, with the exception of \xintTrunc, \xintRound (which produce decimal numbers) and \xintIrr, \xintJrr, \xintRawWithZeros (which returns an A/B with no trailing [n], and prints the B even if it is 1), and \xintPRaw which does not print the [n] if n=0 or the B if B=1.

To be certain to print an integer output without trailing [n] nor fraction slash, one should use either \xintPRaw {\xintIrr {f}} or \xintNum {f} when it is already known that f evaluates to a (big) integer. For example \xintPRaw {\xintAdd {2/5}{3/5}} gives a perhaps disappointing 25/25[63], whereas \xintPRaw {\xintIrr {\xintAdd {2/5}{3/5}}} returns 1. As we knew the result was an integer we could have used \xintNum {\xintAdd {2/5}{3/5}}=1.

Some macros (such as \xintiTrunc, \xintiRound, and \xintFac) always produce directly integers on output.

Contents

[63] yes, \xintAdd blindly multiplies denominators...

25.1 \xintNum

f ★ The macro is extended to accept a fraction on input. But this fraction should reduce to an integer. If not an error will be raised. The original is available as \xintiNum. It is imprudent to apply \xintNum to numbers with a large power of ten given either in scientific notation or with the [n] notation, as the macro will add the necessary zeroes to get an explicit integer.

25.2 \xintifInt

Frac *f* *n n* ★ \xintifInt{f}{YES branch}{NO branch} expandably chooses the YES branch if f reveals itself after expansion and simplification to be an integer. As with the other **xint** conditionals, both branches must be present although one of the two (or both, but why then?) may well be an empty brace pair {}. As will all other **xint** conditionals, spaces in-between the braced things do not matter, but a space after the closing brace of the NO branch is significant.

25.3 \xintLen

Frac *f* ★ The original macro is extended to accept a fraction on input.

\xintLen {201710/298219}=11, \xintLen {1234/1}=4, \xintLen {1234}=4

25.4 \xintRaw

Frac *f* ★ This macro 'prints' the fraction f as it is received by the package after its parsing and expansion, in a form A/B[n] equivalent to the internal representation: the denominator B is always strictly positive and is printed even if it has value 1.

```
\xintRaw{\the\numexpr 571*987\relax.123e-10/\the\numexpr -201+59\relax e-7}=
-563577123/142[-6]
```

25.5 \xintPRaw

Frac *f* ★ PRaw stands for “pretty raw”. It does *not* show the [n] if n=0 and does *not* show the B if B=1.

\xintPRaw {123e10/321e10}=123/321, \xintPRaw {123e9/321e10}=123/321[-1]

\xintPRaw {\xintIrr{861/123}}=7 vz. \xintIrr{861/123}=7/1

See also \xintFrac (or \xintFwOver) for math mode. As is examplified above the \xintIrr macro which puts the fraction into irreducible form does not remove the /1 if the fraction is an integer. One can use \xintNum for that, but there will be an error message if the fraction was not an integer; so the combination \xintPRaw{\xintIrr{f}} is the way to go.

25.6 \xintNumerator

Frac *f* ★ This returns the numerator corresponding to the internal representation of a fraction, with positive powers of ten converted into zeroes of this numerator:

\xintNumerator {178000/25600000[17]}=17800000000000000000000

\xintNumerator {312.289001/20198.27}=312289001

\xintNumerator {178000e-3/256e5}=178000

\xintNumerator {178.000/25600000}=178000

As shown by the examples, no simplification of the input is done. For a result uniquely associated to the value of the fraction first apply \xintIrr.

25.7 \xintDenominator

Frac *f* ★ This returns the denominator corresponding to the internal representation of the fraction:[64]

\xintDenominator {178000/25600000[17]}=25600000

\xintDenominator {312.289001/20198.27}=20198270000

\xintDenominator {178000e-3/256e5}=25600000000

\xintDenominator {178.000/25600000}=25600000000

As shown by the examples, no simplification of the input is done. The denominator looks wrong in the last example, but the numerator was tacitly multiplied by 1000 through the removal of the decimal point. For a result uniquely associated to the value of the fraction first apply \xintIrr.

25.8 \xintRawWithZeros

Frac *f* ★ This macro ‘prints’ the fraction f (after its parsing and expansion) in A/B form, with A as returned by \xintNumerator{f} and B as returned by \xintDenominator{f}.

\xintRawWithZeros{\the\numexpr 571*987\relax.123e-10/\the\numexpr -201+59\relax e-7}=
-563577123/142000000

25.9 \xintREZ

Frac *f* ★ This command normalizes a fraction by removing the powers of ten from its numerator and denominator:

\xintREZ {178000/25600000[17]}=178/256[15]

[64] recall that the [] construct excludes presence of a decimal point.

\xintREZ {178000000000000e30/256000000000000e15}=178/256[15]
As shown by the example, it does not otherwise simplify the fraction.

25.10 \xintFrac

Frac f ★ This is a LaTeX only command, to be used in math mode only. It will print a fraction, internally represented as something equivalent to A/B[n] as \frac {A}{B}10^n. The power of ten is omitted when n=0, the denominator is omitted when it has value one, the number being separated from the power of ten by a \cdot. $\xintFrac {178.000/25600000}$ gives $\frac{178000}{25600000}10^{-3}$, $\xintFrac {178.000/1}$ gives $178000 \cdot 10^{-3}$, $\xintFrac {3.5/5.7}$ gives $\frac{35}{57}$, and $\xintFrac {\xintNum {\xintFac{10} /\xintiSqr{\xintFac {5}}}}$ gives 252. As shown by the examples, simplification of the input (apart from removing the decimal points and moving the minus sign to the numerator) is not done automatically and must be the result of macros such as \xintIrr, \xintREZ, or \xintNum (for fractions being in fact integers.)

25.11 \xintSignedFrac

Frac f ★ This is as \xintFrac except that a negative fraction has the sign put in front, not in the numerator.

\[\xintFrac {-355/113}=\xintSignedFrac {-355/113}\]

$$\frac{-355}{113} = -\frac{355}{113}$$

25.12 \xintFwOver

Frac f ★ This does the same as \xintFrac except that the \over primitive is used for the fraction (in case the denominator is not one; and a pair of braces contains the A\over B part). $\xintFwOver {178.000/25600000}$ gives $\frac{178000}{25600000}10^{-3}$, $\xintFwOver {178.000/1}$ gives $178000 \cdot 10^{-3}$, $\xintFwOver {3.5/5.7}$ gives $\frac{35}{57}$, and $\xintFwOver {\xintNum {\xintFac{10}/\xintiSqr{\xintFac {5}}}}$ gives 252.

25.13 \xintSignedFwOver

Frac f ★ This is as \xintFwOver except that a negative fraction has the sign put in front, not in the numerator.

\[\xintFwOver {-355/113}=\xintSignedFwOver {-355/113}\]

$$\frac{-355}{113} = -\frac{355}{113}$$

25.14 \xintIrr

Frac f ★ This puts the fraction into its unique irreducible form:

\xintIrr {178.256/256.178}=6856/9853 = $\frac{6856}{9853}$

Note that the current implementation does not cleverly first factor powers of 2 and 5, so input such as \xintIrr {2/3[100]} will make xintfrac do the Euclidean division of 2·10^{100} by 3, which is a bit stupid.

Starting with release 1.08, \xintIrr does not remove the trailing /1 when the output is an integer. This was deemed better for various (stupid?) reasons and thus the output format is now *always* A/B with B>0. Use \xintPRaw on top of \xintIrr if it is needed to get rid of a possible trailing /1. For display in math mode, use rather \xintFrac{\xintIrr {f}} or \xintFwOver{\xintIrr {f}}.

25.15 \xintJrr

Frac f ★ This also puts the fraction into its unique irreducible form:

\xintJrr {178.256/256.178}=6856/9853

This is faster than \xintIrr for fractions having some big common factor in the numerator and the denominator.

\xintJrr {\xintiPow{\xintFac {15}}{3}/\xintiiPrdExpr {\xintFac{10}}{\xintFac{30}}{\xintFac{5}}\relax }=1001/51705840

But to notice the difference one would need computations with much bigger numbers than in this example. Starting with release 1.08, \xintJrr does not remove the trailing /1 when the output is an integer.

25.16 \xintTrunc

num x Frac f ★ \xintTrunc{x}{f} returns the integral part, a dot, and then the first x digits of the decimal expansion of the fraction f. The argument x should be non-negative.

In the special case when f evaluates to 0, the output is 0 with no decimal point nor decimal digits, else the post decimal mark digits are always printed. A non-zero negative f which is smaller in absolute value than 10^{-x} will give -0.000....

\xintTrunc {16}{-803.2028/20905.298}=-0.0384210165289200

\xintTrunc {20}{-803.2028/20905.298}=-0.03842101652892008523

\xintTrunc {10}{\xintPow {-11}{-11}}=-0.0000000000

\xintTrunc {12}{\xintPow {-11}{-11}}=-0.000000000003

\xintTrunc {12}{\xintAdd {-1/3}{3/9}}=0

The digits printed are exact up to and including the last one.

25.17 \xintiTrunc

num x Frac f ★ \xintiTrunc{x}{f} returns the integer equal to 10^x times what \xintTrunc{x}{f} would produce.

\xintiTrunc {16}{-803.2028/20905.298}=-384210165289200

\xintiTrunc {10}{\xintPow {-11}{-11}}=0

\xintiTrunc {12}{\xintPow {-11}{-11}}=-3

The difference between \xintTrunc{0}{f} and \xintiTrunc{0}{f} is that the latter never has the decimal mark always present in the former except for f=0. And \xintTrunc{0}{-0.5} returns "-0." whereas \xintiTrunc{0}{-0.5} simply returns "0".

25.18 \xintXTrunc

num x Frac f ☆ \xintXTrunc{x}{f} is completely expandable but not *f*-expandable, as is indicated by the hollow star in the margin. It can not be used as argument to the other package macros, but is designed to be used inside an `\edef`, or rather a `\write`. Here is an example session where the user after some warming up checks that `1/66049=1/257^2` has period `257*256=65792` (it is also checked here that this is indeed the smallest period).

```
xxx:_xint $ etex -jobname worksheet-66049
This is pdfTeX, Version 3.1415926-2.5-1.40.14 (TeX Live 2013)
 restricted \write18 enabled.
**\relax
entering extended mode

*\input xintfrac.sty
(./xintfrac.sty (./xint.sty (./xinttools.sty)))
*\message{\xintTrunc {100}{1/71}}% Warming up!

0.01408450704225352112676056338028169014084507042253521126760563380281690140845
0704225352112676056338O
*\message{\xintTrunc {350}{1/71}}% period is 35

0.01408450704225352112676056338028169014084507042253521126760563380281690140845
07042253521126760563380281690140845070422535211267605633802816901408450704225352
11267605633802816901408450704225352112676056338028169014084507042253521126760563
38028169014084507042253521126760563380281690140845070422535211267605633802816
901408450704225352112676056338028169
*\edef\Z {\xintXTrunc  {65792}{1/66049}}% getting serious...

*\def\trim 0.{}\oodef\Z {\expandafter\trim\Z}% removing 0.

*\edef\W {\xintXTrunc {131584}{1/66049}}% a few seconds

*\oodef\W {\expandafter\trim\W}

*\oodef\ZZ {\expandafter\Z\Z}% doubling the period

*\ifx\W\ZZ \message{YES!}\else\message{BUG!}\fi % xint never has bugs...
YES!
*\message{\xintTrunc {260}{1/66049}}% check visually that 256 is not a period

0.00001514027464458205271843631243470756559523989765174340262532362337052794137
68565761782918742145982528123060152311162924495450347469303093158109888113370376
53863041075565110751109025117715635361625459885842329179851322502990204242304955
41189117170585474420505
*\edef\X {\xintXTrunc {257*128}{1/66049}}% infix here ok, less than 8 tokens

*\oodef\X {\expandafter\trim\X}% we now have the first 257*128 digits

*\oodef\XX {\expandafter\X\X}% was 257*128 a period?

*\ifx\XX\Z \message{257*128 is a period}\else \message{257 * 128 not a period}\fi
257 * 128 not a period
*\immediate\write-1 {1/66049=0.\Z... (repeat)}

*\oodef\ZA {\xintNum {\Z}}% we remove the 0000, or we could use next \xintiMul
```

```
*\immediate\write-1 {10\string^65792-1=\xintiiMul {\ZA}{66049}}

*% This was slow :( I should write a multiplication, still completely

*% expandable, but not f-expandable, which could be much faster on such cases.

*\bye
No pages of output.
Transcript written on worksheet-66049.log.
xxx:_xint $
```

Using `\xintTrunc` rather than `\xintXTrunc` would be hopeless on such long outputs (and even `\xintXTrunc` needed of the order of seconds to complete here). But it is not worth it to use `\xintXTrunc` for less than hundreds of digits.

Fraction arguments to `\xintXTrunc` corresponding to a `A/B[N]` with a negative `N` are treated somewhat less efficiently (additional memory impact) than for positive or zero `N`. This is because the algorithm tries to work with the smallest denominator hence does not extend `B` with zeroes, and technical reasons lead to the use of some tricks.[65]

Contrarily to `\xintTrunc`, in the case of the second argument revealing itself to be exactly zero, `\xintXTrunc` will output `0.000...`, not `0`. Also, the first argument must be at least `1`.

25.19 \xintRound

num x Frac f ★

`\xintRound{x}{f}` returns the start of the decimal expansion of the fraction `f`, rounded to `x` digits precision after the decimal point. The argument `x` should be non-negative. Only when `f` evaluates exactly to zero does `\xintRound` return `0` without decimal point. When `f` is not zero, its sign is given in the output, also when the digits printed are all zero.

```
\xintRound {16}{-803.2028/20905.298}=-0.0384210165289201
\xintRound {20}{-803.2028/20905.298}=-0.03842101652892008523
\xintRound {10}{\xintPow {-11}{-11}}=-0.0000000000
\xintRound {12}{\xintPow {-11}{-11}}=-0.000000000004
\xintRound {12}{\xintAdd {-1/3}{3/9}}=0
```

The identity `\xintRound {x}{-f}=-\xintRound {x}{f}` holds. And regarding $(-11)^{-11}$ here is some more of its expansion:

```
-0.00000000000350493899481392497604003313162598556370...
```

25.20 \xintiRound

num x Frac f ★

`\xintiRound{x}{f}` returns the integer equal to `10^x` times what `\xintRound{x}{f}` would return.

```
\xintiRound {16}{-803.2028/20905.298}=-384210165289201
\xintiRound {10}{\xintPow {-11}{-11}}=0
```

[65] Technical note: I do not provide an `\xintXFloat` because this would almost certainly mean having to clone the entire core division routines into a “long division” variant. But this could have given another approach to the implementation of `\xintXTrunc`, especially for the case of a negative `N`. Doing these things with TeX is an effort. Besides an `\xintXFloat` would be interesting only if also for example the square root routine was provided in an `X` version (I have not given thought to that). If feasible `X` routines would be interesting in the `\xintexpr` context where things are expanded inside `\csname..\endcsname`.

Differences between `\xintRound{0}{f}` and `\xintiRound{0}{f}`: the former cannot be used inside integer-only macros, and the latter removes the decimal point, and never returns `-0` (and removes all superfluous leading zeroes.)

25.21 \xintFloor

Frac f ★ `\xintFloor {f}` returns the largest relative integer `N` with `N` ≤ `f`.

`\xintFloor {-2.13}=-3`, `\xintFloor {-2}=-2`, `\xintFloor {2.13}=2`

25.22 \xintCeil

Frac f ★ `\xintCeil {f}` returns the smallest relative integer `N` with `N` > `f`.

`\xintCeil {-2.13}=-2`, `\xintCeil {-2}=-2`, `\xintCeil {2.13}=3`

25.23 \xintTFrac

Frac f ★ `\xintTFrac{f}` returns the fractional part, `f=trunc(f)+frac(f)`. The `T` stands for 'Trunc', and there could similar macros associated to 'Round', 'Floor', and 'Ceil'. Inside `\xintexpr..\relax`, the function `frac` is mapped to `\xintTFrac`. Inside `\xintfloatexpr..\relax`, `frac` first applies `\xintTFrac` to its argument (which may be in float format, or an exact fraction), and only next makes the float conversion.

`\xintTFrac {1235/97}=71/97[0]` `\xintTFrac {-1235/97}=-71/97[0]`
`\xintTFrac {1235.973}=973/1[-3]` `\xintTFrac {-1235.973}=-973/1[-3]`
`\xintTFrac {1.122435727e5}=5727/1[-4]`

25.24 \xintE

Frac f num x ★ `\xintE {f}{x}` multiplies the fraction `f` by `10^x`. The *second* argument `x` must obey the TEX bounds. Example:

`\count 255 123456789 \xintE {10}{\count 255}->10/1[123456789]`

Be careful that for obvious reasons such gigantic numbers should not be given to \xintNum, or added to something with a widely different order of magnitude, as the package always works to get the *exact* result. There is *no problem* using them for *float* operations:

`\xintFloatAdd {1e1234567890}{1}=1.000000000000000e1234567890`

25.25 \xintFloatE

[num x] Frac f num x ★ `\xintFloatE [P]{f}{x}` multiplies the input `f` by `10^x`, and converts it to float format according to the optional first argument or current value of `\xintDigits`.

`\xintFloatE {1.23e37}{53}=1.230000000000000e90`

25.26 \xintDigits, \xinttheDigits

The syntax `\xintDigits := D;` (where spaces do not matter) assigns the value of `D` to the number of digits to be used by floating point operations. The default is `16`. The maximal ★ value is `32767`. The macro `\xinttheDigits` serves to print the current value.

25.27 \xintFloat

[num x] Frac f ★ The macro `\xintFloat [P]{f}` has an optional argument `P` which replaces the current value of `\xintDigits`. The (rounded truncation of the) fraction `f` is then printed in scientific form, with `P` digits, a lowercase `e` and an exponent `N`. The first digit is from `1` to `9`, it is preceded by an optional minus sign and is followed by a dot and `P-1` digits, the trailing zeroes are not trimmed. In the exceptional case where the rounding went to the next power of ten, the output is `10.0...0eN` (with a sign, perhaps). The sole exception is for a zero value, which then gets output as `0.e0` (in an `\xintCmp` test it is the only possible output of `\xintFloat` or one of the 'Float' macros which will test positive for equality with zero).

```
\xintFloat[32]{1234567/7654321}=1.6129020457856418616360615134902e-1
\xintFloat[32]{1/\xintFac{100}}=1.0715102881254669231835467595192e-158
```

The argument to `\xintFloat` may be an `\xinttheexpr`-ession, like the other macros; only its final evaluation is submitted to `\xintFloat`: the inner evaluations of chained arguments are not at all done in 'floating' mode. For this one must use `\xintthefloatexpr`.

25.28 \xintAdd

Frac f Frac f ★ The original macro is extended to accept fractions on input. Its output will now always be in the form `A/B[n]`. The original is available as `\xintiAdd`.

25.29 \xintFloatAdd

[num x] Frac f Frac f ★ `\xintFloatAdd [P]{f}{g}` first replaces `f` and `g` with their float approximations, with 2 safety digits. It then adds exactly and outputs in float format with precision `P` (which is optional) or `\xintDigits` if `P` was absent, the result of this computation.

25.30 \xintSub

Frac f Frac f ★ The original macro is extended to accept fractions on input. Its output will now always be in the form `A/B[n]`. The original is available as `\xintiSub`.

25.31 \xintFloatSub

[num x] Frac f Frac f ★ `\xintFloatSub [P]{f}{g}` first replaces `f` and `g` with their float approximations, with 2 safety digits. It then subtracts exactly and outputs in float format with precision `P` (which is optional), or `\xintDigits` if `P` was absent, the result of this computation.

25.32 \xintMul

Frac f Frac f ★ The original macro is extended to accept fractions on input. Its output will now always be in the form `A/B[n]`. The original, only for big integers, and outputting a big integer, is available as `\xintiMul`.

25.33 \xintFloatMul

[num x] Frac f Frac f ★ `\xintFloatMul [P]{f}{g}` first replaces `f` and `g` with their float approximations, with 2 safety digits. It then multiplies exactly and outputs in float format with precision `P` (which is optional), or `\xintDigits` if `P` was absent, the result of this computation.

25.34 \xintSqr

Frac f ★

The original macro is extended to accept a fraction on input. Its output will now always be in the form `A/B[n]`. The original which outputs only big integers is available as \xintiSqr.

25.35 \xintDiv

Frac f Frac f ★

`\xintDiv{f}{g}` computes the fraction `f/g`. As with all other computation macros, no simplification is done on the output, which is in the form `A/B[n]`.

25.36 \xintFloatDiv

[num x] Frac f Frac f ★

`\xintFloatDiv [P]{f}{g}` first replaces `f` and `g` with their float approximations, with 2 safety digits. It then divides exactly and outputs in float format with precision `P` (which is optional), or `\xintDigits` if `P` was absent, the result of this computation.

25.37 \xintFac

Num f ★

The original is extended to allow a fraction on input but this fraction `f` must simplify to a integer `n` (non negative and at most `999999`, but already `100000!` is prohibitively time-costly). On output `n!` (no trailing `/1[0]`). The original macro (which has less overhead) is still available as \xintiFac.

25.38 \xintPow

Frac f Num f ★

`\xintPow{f}{g}`: the original macro is extended to accept fractions on input. The output will now always be in the form `A/B[n]` (even when the exponent vanishes: `\xintPow {2/3}{0}=1/1[0]`). The original is available as \xintiPow.

The exponent is allowed to be input as a fraction but it must simplify to an integer: `\xintPow {2/3}{10/2}=32/243[0]`. This integer will be checked to not exceed `100000`. Indeed `2^50000` already has `15052` digits, and squaring such a number would take hours (I think) with the expandable routine of **xint**.

25.39 \xintFloatPow

[num x] Frac f num x ★

`\xintFloatPow [P]{f}{x}` uses either the optional argument `P` or the value of `\xintDigits`. It computes a floating approximation to `f^x`. The precision `P` must be at least `1`, naturally.

The exponent `x` will be fed to a `\numexpr`, hence count registers are accepted on input for this `x`. And the absolute value `|x|` must obey the TeX bound. For larger exponents use the slightly slower routine \xintFloatPower which allows the exponent to be a fraction simplifying to an integer and does not limit its size. This slightly slower routine is the one to which `^` is mapped inside `\xintthefloatexpr...\relax`.

The macro `\xintFloatPow` chooses dynamically an appropriate number of digits for the intermediate computations, large enough to achieve the desired accuracy (hopefully).

```
\xintFloatPow [8]{3.1415}{1234567890}=1.6122066e613749456
```

25.40 \xintFloatPower

$[{}^{\text{num}}_{x}]\ {}^{\text{Frac}}_{f}\ {}^{\text{Num}}_{f}\ \star$ \xintFloatPower[P]{f}{g} computes a floating point value f^g where the exponent g is not constrained to be at most the TeX bound 2147483647. It may even be a fraction A/B but must simplify to a (possibly big) integer.

\xintFloatPower [8]{1.000000000001}{1e12}=2.7182818e0

\xintFloatPower [8]{3.1415}{3e9}=1.4317729e1491411192

Note that 3e9>2^31. But the number following e in the output must at any rate obey the TeX 2147483647 bound.

Inside an \xintfloatexpr-ession, \xintFloatPower is the function to which ^ is mapped. The exponent may then be something like (144/3/(1.3-.5)-37) which is, in disguise, an integer.

The intermediate multiplications are done with a higher precision than \xintDigits or the optional P argument, in order for the final result to hopefully have the desired accuracy.

25.41 \xintFloatSqrt

$[{}^{\text{num}}_{x}]\ {}^{\text{Frac}}_{f}\ \star$ \xintFloatSqrt[P]{f} computes a floating point approximation of $\sqrt{\mathtt{f}}$, either using the optional precision P or the value of \xintDigits. The computation is done for a precision of at least 17 figures (and the output is rounded if the asked-for precision was smaller).

\xintFloatSqrt [50]{12.3456789e12}

≈ 3.5136418286444621616658231167580770371591427181243e6

\xintDigits:=50;\xintFloatSqrt {\xintFloatSqrt {2}}

≈ 1.1892071150027210667174999705604759152929720924638e0

25.42 \xintSum

$f \to *\ {}^{\text{Frac}}_{f}\ \star$ The original command is extended to accept fractions on input and produce fractions on output. The output will now always be in the form A/B[n]. The original, for big integers only (in strict format), is available as \xintiiSum.

25.43 \xintPrd

$f \to *\ {}^{\text{Frac}}_{f}\ \star$ The original is extended to accept fractions on input and produce fractions on output. The output will now always be in the form A/B[n]. The original, for big integers only (in strict format), is available as \xintiiPrd.

25.44 \xintCmp

${}^{\text{Frac}}_{f}\ {}^{\text{Frac}}_{f}\ \star$ The macro is extended to fractions. Its output is still either -1, 0, or 1 with no forward slash nor trailing [n].

For choosing branches according to the result of comparing f and g, the following syntax is recommended: \xintSgnFork{\xintCmp{f}{g}}{code for f<g}{code for f=g}{code for f>g}.

25.45 \xintIsOne

${}^{\text{Frac}}_{f}\ \star$ See \xintIsOne (subsection 24.17).

25.46 \xintGeq

Frac Frac
f f ★ The macro is extended to fractions. Beware that the comparison is on the *absolute values* of the fractions. Can be used as: `\xintSgnFork{\xintGeq{f}{g}}{}{code for |f|<|g|}{code for |f|≥|g|}`

25.47 \xintMax

Frac Frac
f f ★ The macro is extended to fractions. But now `\xintMax {2}{3}` returns `3/1[0]`. The original, for use with (possibly big) integers only, is available as \xintiMax: `\xintiMax {2}{3}=3`.

25.48 \xintMaxof

Frac
$f \rightarrow * f$ ★ See \xintMaxof (subsection 24.26).

25.49 \xintMin

Frac Frac
f f ★ The macro is extended to fractions. The original, for (big) integers only, is available as \xintiMin.

25.50 \xintMinof

Frac
$f \rightarrow * f$ ★ See \xintMinof (subsection 24.28).

25.51 \xintAbs

Frac
f ★ The macro is extended to fractions. The original, for (big) integers only, is available as \xintiAbs. Note that `\xintAbs {-2}=2/1[0]` whereas `\xintiAbs {-2}=2`.

25.52 \xintSgn

Frac
f ★ The macro is extended to fractions. Naturally, its output is still either `-1`, `0`, or `1` with no forward slash nor trailing `[n]`.

25.53 \xintOpp

Frac
f ★ The macro is extended to fractions. The original is available as \xintiOpp. Note that `\xintOpp {3}` now outputs `-3/1[0]` whereas `\xintiOpp {3}` returns `-3`.

25.54 \xintDivision, \xintQuo, \xintRem, \xintFDg, \xintLDg, \xintMON, \xintMMON, \xintOdd

Frac Frac
f f ★
Frac
or f ★ These macros accept a fraction on input if this fraction in fact reduces to an integer (if not an `\xintError:NotAnInteger` will be raised). There is no difference in the format of the outputs, which are still (possibly big) integers without fraction slash nor trailing `[n]`, the sole difference is in the extended range of accepted inputs.

All have variants whose names start with `xintii` rather than `xint`; these variants accept on input only integers in the strict format (they do not use \xintNum). They thus have less overhead, and may be used when one is dealing exclusively with (big) integers. These variants are already available in **xint**, there is no need for **xintfrac** to be loaded.

`\xintNum {1e80}`

100000000000000000000000000000000000000000000000000000000000000000000000000000000

26 Expandable expressions with the xintexpr package

The **xintexpr** package was first released with version `1.07` of the **xint** bundle. It loads automatically **xintfrac**, hence also **xint** and **xinttools**.

The syntax is described in section 20 and section 21.

Contents

26.1 The \xintexpr expressions

x ⋆ An **xintexpr**ession is a construct `\xintexpr`⟨*expandable_expression*⟩`\relax` where the expandable expression is read and completely expanded from left to right.

During this parsing, braced sub-content `{`⟨*expandable*⟩`}` may be serving as a macro parameter, or a branch of the `?` two-way and `:` three-way operators; else it is treated in a special manner:

1. it is allowed to occur only at the spots where numbers are legal,
2. the ⟨*expandable*⟩ contents is then completely expanded as if it were put in a `\csname..\endcsname`,[66] thus it escapes entirely the parser scope and infix notations will not be recognized except if the expanded macros know how to handle them by themselves,
3. and this complete expansion *must* produce a number or a fraction, possibly with decimal mark and trailing `[n]`, the scientific notation is *not* authorized.

Braces are the only way to input some number or fraction with a trailing `[n]`: square brackets are *not* accepted in a `\xintexpr...\relax` if not within such braces.

[66] well, actually it *is* put in such a `\csname..\endcsname`.

An \xintexpr..\relax *must* end in a \relax (which will be absorbed). Like a \numexpr expression, it is not printable as is, nor can it be directly employed as argument to the other package macros. For this one must use one of the two equivalent forms:

x ★ • \xinttheexpr⟨*expandable_expression*⟩\relax, or

x ★ • \xintthe\xintexpr⟨*expandable_expression*⟩\relax.

The computations are done *exactly*, and with no simplification of the result. The output format for the result can be coded inside the expression through the use of one of the functions round, trunc, float, reduce.[67] Here are some examples

```
\xinttheexpr 1/5!-1/7!-1/9!\relax=1784764800/219469824000[0]
\xinttheexpr round(1/5!-1/7!-1/9!,18)\relax=0.008132164902998236
\xinttheexpr float(1/5!-1/7!-1/9!,18)\relax=813216490299823633[-20]
\xinttheexpr reduce(1/5!-1/7!-1/9!)\relax=2951/362880
\xinttheexpr 1.99^-2 - 2.01^-2 \relax=800/1599920001[4]
\xinttheexpr round(1.99^-2 - 2.01^-2, 10)\relax=0.0050002500
```

- the expression may contain arbitrarily many levels of nested parenthesized sub-expressions.
- sub-contents giving numbers of fractions should be either
  1. parenthesized,
  2. a sub-expression \xintexpr...\relax,
  3. or within braces.

When a sub-expression is hit against in the midst of absorbing the digits of a number, a *
1.09j → to force tacit multiplication is inserted.. Similarly, if it is an opening parenthesis which is
1.09k → hit against.

- an expression can not be given as argument to the other package macros, nor printed, for this one must use \xinttheexpr...\relax or \xintthe\xintexpr...\relax.

- one does not use \xinttheexpr...\relax as a sub-constituent of an \xintexpr...\relax but simply \xintexpr...\relax; this is mainly because most of the time \xinttheexpr..\relax will insert explicit square brackets which are not parsable, as already mentioned, in the surrounding expression.

- each **xintexpr**ession is completely expandable and obtains its result in two expansion steps.

In an algorithm implemented non-expandably, one may define macros to expand to infix expressions to be used within others. One then has the choice between parentheses or \xintexpr...\relax: \def\x {(\a+\b)} or \def\x {\xintexpr \a+\b\relax}. The latter is the better choice as it allows also to be prefixed with \xintthe. Furthemore, if \a and \b are already defined \oodef\x {\xintexpr \a+\b\relax} will do the computation on the spot.

[67] In round and trunc the second optional parameter is the number of digits of the fractional part; in float it is the total number of digits of the mantissa.

26.2 `\numexpr` or `\dimexpr` expressions, count and dimension registers and variables

Count registers, count control sequences, dimen registers, dimen control sequences, skips and skip control sequences, `\numexpr`, `\dimexpr`, `\glueexpr` can be inserted directly, they will be unpacked using `\number` (which gives the internal value in terms of scaled points for the dimensional variables: 1 pt = 65536 sp; stretch and shrink components are thus discarded). Tacit multiplication is implied, when a number or decimal number prefixes such a register or control sequence.

LaTeX lengths are skip control sequences and LaTeX counters should be inserted using `\value`.

In the case of numbered registers like `\count255` or `\dimen0`, the resulting digits will be re-parsed, so for example `\count255 0` is like `100` if `\the\count255` would give `10`. Control sequences define complete numbers, thus cannot be extended that way with more digits, on the other hand they are more efficient as they avoid the re-parsing of their unpacked contents.

A token list variable must be prefixed by `\the`, it will not be unpacked automatically (the parser will actually try `\number`, and thus fail). Do not use `\the` but only `\number` with a dimen or skip, as the `\xintexpr` parser doesn't understand `pt` and its presence is a syntax error. To use a dimension expressed in terms of points or other TeX recognized units, incorporate it in `\dimexpr...\relax`.

If one needs to optimize, `1.72\dimexpr 3.2pt\relax` is less efficient than `1.72*{\number\dimexpr 3.2pt\relax}` as the latter avoids re-parsing the digits of the representation of the dimension as scaled points.

```
\xinttheexpr 1.72\dimexpr 3.2pt\relax/2.71828\relax=
\xinttheexpr 1.72*{\number\dimexpr 3.2pt\relax}/2.71828\relax
36070980/271828[3]=36070980/271828[3]
```

Regarding how dimensional expressions are converted by TeX into scaled points see subsection 9.2.

26.3 Catcodes and spaces

26.3.1 `\xintexprSafeCatcodes`

Active characters will interfere with `\xintexpr`-essions. One may prefix them with `\string` within `\xintexpr..\relax`, thus preserving expandability, or there is the non-expandable `\xintexprSafeCatcodes` which can be issued before the use of `\xintexpr`. This command sets (not globally) the catcodes of the relevant characters to safe values. This is used internally by `\xintNewExpr` (restoring the catcodes on exit), hence `\xintNewExpr` does not have to be protected against active characters.

26.3.2 `\xintexprRestoreCatcodes`

Restores the catcodes to the earlier state.

Unbraced spaces inside an `\xinttheexpr...\relax` should mostly be innocuous (except inside macro arguments).

\xintexpr and \xinttheexpr are for the most part agnostic regarding catcodes: (unbraced) digits, binary operators, minus and plus signs as prefixes, dot as decimal mark, parentheses, may be indifferently of catcode letter or other or subscript or superscript, ..., it doesn't matter.[68]

The characters +,-,*,/,^,!,&,|,?,:,<,>,=,(,)," , the dot and the comma should not be active as everything is expanded along the way. If one of them is active, it should be prefixed with \string.

The ! as either logical negation or postfix factorial operator must be a standard (*i.e.* catcode 12) !, more precisely a catcode 11 exclamation point ! must be avoided as it is used internally by \xintexpr for various special purposes.

Digits, slash, square brackets, minus sign, in the output from an \xinttheexpr are all of catcode 12. For \xintthefloatexpr the 'e' in the output is of catcode 11.

A macro with arguments will expand and grab its arguments before the parser may get a chance to see them, so the situation with catcodes and spaces is not the same within such macro arguments (or within braces used to protect square brackets).

26.4 Expandability, \xinteval

As is the case with all other package macros \xintexpr f-expands (in two steps) to its final (non-printable) result; and \xinttheexpr f-expands (in two steps) to the chain of digits (and possibly minus sign -, decimal mark ., fraction slash /, scientific e, square brackets [,]) representing the result.

New with 1.09j! → Starting with 1.09j, an \xintexpr..\relax can be inserted without \xintthe prefix inside an \edef, or a \write. It expands to a private more compact representation (five tokens) than \xinttheexpr or \xintthe\xintexpr.

The material between \xintexpr and relax should contain only expandable material; the exception is with brace pairs which, apart from their usual rôle for macro arguments, are also allowed in places where the scanner expects a numeric operand, the braced material should expand to some number (or fraction), but scientific notation is not allowed. Conversely fractions in A/B[N] format (either explicit or from macro expansion) must be enclosed in such a brace pair.

The once expanded \xintexpr is \romannumeral0\xinteval. And there is similarly \xintieval, \xintiieval, and \xintfloateval. For the other cases one can use \romannumeral-`0 as prefix. For an example of expandable algorithms making use of chains

New with 1.09j! → of \xinteval-uations connected via \expandafter see subsection 23.22.

An expression can only be legally finished by a \relax token, which will be absorbed.

26.5 Memory considerations

The parser creates an undefined control sequence for each intermediate computation (this does not refer to the intermediate steps needed in the evaluations of the \xintAdd, \xintMul, etc... macros corresponding to the infix operators, but only to each conversion of such an infix operator into a computation). So, a moderately sized expression might create 10, or 20 such control sequences. On my TeX installation, the memory available for such things is of circa 200,000 multi-letter control words. So this means that a document containing hundreds, perhaps even thousands of expressions will compile with no problem.

[68] Furthermore, although \xintexpr uses \string, it is (we hope) escape-char agnostic.

Besides the hash table, also TEX main memory is impacted. Thus, if **xintexpr** is used for computing plots[69], this may cause a problem.

There is a solution.[70]

A document can possibly do tens of thousands of evaluations only if some formulas are being used repeatedly, for example inside loops, with counters being incremented, or with data being fetched from a file. So it is the same formula used again and again with varying numbers inside.

With the `\xintNewExpr` command, it is possible to convert once and for all an expression containing parameters into an expandable macro with parameters. Only this initial definition of this macro actually activates the `\xintexpr` parser and will (very moderately) impact the hash-table: once this unique parsing is done, a macro with parameters is produced which is built-up recursively from the `\xintAdd`, `\xintMul`, etc... macros, exactly as it would be necessary to do without the facilities of the **xintexpr** package.

26.6 The \xintNewExpr command

The command is used as:

`\xintNewExpr{\myformula}[n]{`⟨*stuff*⟩`}`, where

- ⟨*stuff*⟩ will be inserted inside `\xinttheexpr . . . \relax`,
- `n` is an integer between zero and nine, inclusive, and tells how many parameters will `\myformula` have (it is *mandatory* even if `n=0`[71])
- the placeholders `#1`, `#2`, ..., `#n` are used inside ⟨*stuff*⟩ in their usual rôle.

The macro `\myformula` is defined without checking if it already exists, LATEX users might prefer to do first `\newcommand*\myformula {}` to get a reasonable error message in case `\myformula` already exists.

The definition of `\myformula` made by `\xintNewExpr` is global. The protection against active characters is done automatically.

It will be a completely expandable macro entirely built-up using `\xintAdd`, `\xintSub`, `\xintMul`, `\xintDiv`, `\xintPow`, etc. . . as corresponds to the expression written with the infix operators.

Macros created by `\xintNewExpr` can thus be nested:

```
\xintNewExpr \MyFunction [1]{reduce(2*#1^3 - #1^-2*3)}
(1) \MyFunction {\MyFunction {2/3}}
\xintNewFloatExpr \MyOtherFunction [1]{(#1+#1^-1)/(#1-#1^-1)}
(2) \MyOtherFunction {1.234}
(3) \MyOtherFunction {\MyOtherFunction {1.234}}
```

(1) `-130071402086777/278538069600`
(2) `4.825876699645722e0`
(3) `1.089730014373938e0`

---

[69] this is not very probable as so far **xint** does not include a mathematical library with floating point calculations, but provides only the basic operations of algebra. [70] which convinced me that I could stick with the parser implementation despite its potential impact on the hash-table and other parts of TEX's memory. [71] there is some use for `\xintNewExpr[0]` compared to an `\edef` as `\xintNewExpr` has some built-in catcode protection.

A "formula" created by `\xintNewExpr` is thus a macro whose parameters are given to a possibly very complicated combination of the various macros of **xint** and **xintfrac**; hence one can not use infix notation inside the arguments, as in for example `\myformula {28^7-35^12}` which would have been allowed by

`\def\myformula #1{\xinttheexpr (#1)^3\relax}`

One will have to do `\myformula {\xinttheexpr 28^7-35^12\relax}`, or redefine `\myformula` to have more parameters.

```
\xintNewExpr\DET[9]{ #1*#5*#9+#2*#6*#7+#3*#4*#8-#1*#6*#8-#2*#4*#9-#3*#5*#7 }
  \meaning\DET:macro:#1#2#3#4#5#6#7#8#9->\romannumeral-`0\xintSub{\xint
Sub{\xintSub{\xintAdd{\xintAdd{\xintMul{\xintMul{#1}{#5}}{#9}}{\xintMul
{\xintMul{#2}{#6}}{#7}}}{\xintMul{\xintMul{#3}{#4}}{#8}}}{\xintMul{\xin
tMul{#1}{#6}}{#8}}}{\xintMul{\xintMul{#2}{#4}}{#9}}}{\xintMul{\xintMul{
#3}{#5}}{#7}}
            \xintNum{\DET {1}{1}{1}{10}{-10}{5}{11}{-9}{6}}=0
           \xintNum{\DET {1}{2}{3}{10}{0}{-10}{21}{2}{-17}}=0
```

Remark: `\meaning` has been used within the argument to a `\printnumber` command, to avoid going into the right margin, but this zaps all spaces originally in the output from `\meaning`. Here is as an illustration the raw output of `\meaning` on the previous example:

```
  macro:#1#2#3#4#5#6#7#8#9->\romannumeral -`0\xintSub {\xintSub {\xintSub
{\xintAdd {\xintAdd {\xintMul {\xintMul {#1}{#5}}{#9}}{\xintMul {\xintMul
{#2}{#6}}{#7}}}{\xintMul {\xintMul {#3}{#4}}{#8}}}{\xintMul {\xintMul {#1}{#6}}{#8}}}{\xin
Mul {\xintMul {#2}{#4}}{#9}}}{\xintMul {\xintMul {#3}{#5}}{#7}}
```

This is why `\printnumber` was used, to have breaks across lines.

26.6.1 Use of conditional operators

The `1.09a` conditional operators `?` and `:` cannot be parsed by `\xintNewExpr` when they contain macro parameters `#1`,..., `#9` within their scope. However replacing them with the functions `if` and, respectively `ifsgn`, the parsing should succeed. And the created macro will *not evaluate the branches to be skipped*, thus behaving exactly like `?` and `:` would have in the `\xintexpr`.

```
                        \xintNewExpr\Formula [3]
    { if((#1>#2) & (#2>#3), sqrt(#1-#2)*sqrt(#2-#3), #1^2+#3/#2) }
\meaning\Formula:macro:#1#2#3->\romannumeral-`0\xintifNotZero{\xintAND{
\xintGt{#1}{#2}}{\xintGt{#2}{#3}}}{\xintMul{\XINTinFloatSqrt[\XINTdigit
s]{\xintSub{#1}{#2}}}{\XINTinFloatSqrt[\XINTdigits]{\xintSub{#2}{#3}}}}
{\xintAdd{\xintPow{#1}{2}}{\xintDiv{#3}{#2}}}
```

This formula (with `\xintifNotZero`) will gobble the false branch.

Remark: this `\XINTinFloatSqrt` macro is a non-user package macro used internally within `\xintexpr`-essions, it produces the result in `A[n]` form rather than in scientific notation, and for reasons of the inner workings of `\xintexpr`-essions, this is necessary; a hand-made macro would have used instead the equivalent `\xintFloatSqrt`.

Another example

```
                       \xintNewExpr\myformula [3]
```

```
                        { ifsgn(#1,#2/#3,#2-#3,#2*#3) }
macro:#1#2#3->\romannumeral-‘0\xintifSgn{#1}{\xintDiv{#2}{#3}}{\xintSub
{#2}{#3}}{\xintMul{#2}{#3}}
```

Again, this macro gobbles the false branches, as would have the operator `:` inside an `\xintexpr`-ession.

26.6.2 Use of macros

For macros to be inserted within such a created **xint**-formula command, there are two cases:

- the macro does not involve the numbered parameters in its arguments: it may then be left as is, and will be evaluated once during the construction of the formula,
- it does involve at least one of the parameters as argument. Then:
  1. the whole thing (macro + argument) should be braced (not necessary if it is already included into a braced group),
  2. the macro should be coded with an underscore `_` in place of the backslash `\`.
  3. the parameters should be coded with a dollar sign `$1`, `$2`, etc...

Here is a silly example illustrating the general principle (the macros here have equivalent functional forms which are more convenient; but some of the more obscure package macros of **xint** dealing with integers do not have functions pre-defined to be in correspondance with them):

```
\xintNewExpr\myformI[2]{ {_xintRound{$1}{$2}} - {_xintTrunc{$1}{$2}} }
\meaning\myformI:
macro:#1#2->\romannumeral -‘0\xintSub {\xintRound {#1}{#2}}{\xintTrunc {#1}{#2}}
\xintNewIIExpr\formula [3]{rem(#1,quo({_the_numexpr $2_relax},#3))}
\meaning\formula:
macro:#1#2#3->\romannumeral -‘0\xintiiRem {#1}{\xintiiQuo {\the \numexpr
#2\relax }{#3}}
```

26.7 `\xintiexpr`, `\xinttheiexpr`

x ★ Equivalent to doing `\xintexpr round(...)\relax`. Thus, only the final result is rounded to an integer. Half integers are rounded towards $+\infty$ for positive numbers and towards $-\infty$ for negative ones. Can be used on comma separated lists of expressions.

1.09i warning: Initially baptized `\xintnumexpr`, `\xintthenumexpr` but I am not too happy about this choice of name; one should keep in mind that `\numexpr`'s integer division rounds, whereas in `\xintiexpr`, the `/` is an exact fractional operation, and only the final result is rounded to an integer.

So `\xintnumexpr`, `\xintthenumexpr` are deprecated, and although still provided for the time being this might change in the future.

26.8 \xintiiexpr, \xinttheiiexpr

x ★ This variant maps / to the euclidean quotient and deals almost only with (long) integers. It uses the 'ii' macros for addition, subtraction, multiplication, power, square, sums, products, euclidean quotient and remainder. The round and trunc, in the presence of the second optional argument, are mapped to \xintiRound, respectively \xintiTrunc, hence they always produce (long) integers.

To input a fraction to round, trunc, floor or ceil one can use braces, else the / will do the euclidean quotient. The minus sign should be put together with the fraction: round(-{30/18}) is illegal (even if the fraction had been an integer), use round({-30/18})=-2.

Decimal numbers are allowed only if postfixed immediately with e or E, the number will then be truncated to an integer after multiplication by the power of ten with exponent the number following e or E.

```
\xinttheiiexpr 13.4567e3+10000123e-3\relax=23456
```

A fraction within braces should be followed immediately by an e (or inside a round, trunc, etc...) to convert it into an integer as expected by the main operations. The truncation is only done after the e action.

The reduce function is not available and will raise un error. The frac function also. The sqrt function is mapped to \xintiSqrt.

Numbers in float notation, obtained via a macro like \xintFloatSqrt, are a bit of a challenge: they can not be within braces (this has been mentioned already, e is not legal within braces) and if not braced they will be truncated when the parser meets the e. The way out of the dilemma is to use a sub-expression:

```
\xinttheiiexpr \xintFloatSqrt{2}\relax=1
\xinttheiiexpr \xintexpr\xintFloatSqrt{2}\relax e10\relax=14142135623
\xinttheiiexpr round(\xintexpr\xintFloatSqrt{2}\relax,10)\relax=14142135624
```

(recall that round is mapped within \xintiiexpr..\relax to \xintiRound which always outputs an integer).

The whole point of \xintiiexpr is to gain some speed in integer only algorithms, and the above explanations related to how to use fractions therein are a bit peripheral. We observed of the order of 30% speed gain when dealing with numbers with circa one hundred digits, but this gain decreases the longer the manipulated numbers become and becomes negligible for numbers with thousand digits: the overhead from parsing fraction format is little compared to other expensive aspects of the expandable shuffling of tokens.

26.9 \xintboolexpr, \xinttheboolexpr

x ★ Equivalent to doing \xintexpr ...\relax and returning 1 if the result does not vanish, and 0 is the result is zero. As \xintexpr, this can be used on comma separated lists of expressions, and will return a comma separated list of 0's and 1's.

26.10 \xintfloatexpr, \xintthefloatexpr

x ★ \xintfloatexpr...\relax is exactly like \xintexpr...\relax but with the four binary operations and the power function mapped to \xintFloatAdd, \xintFloatSub, \xintFloatMul, \xintFloatDiv and \xintFloatPower. The precision is from the current setting of \xintDigits (it can not be given as an optional parameter).

Currently, the factorial function hasn't yet a float version; so inside `\xintthefloatexpr . . . \relax`, `n!` will be computed exactly. Perhaps this will be improved in a future release.

Note that `1.000000001` and `(1+1e-9)` will not be equivalent for `D=\xinttheDigits` set to nine or less. Indeed the addition implicit in `1+1e-9` (and executed when the closing parenthesis is found) will provoke the rounding to 1. Whereas `1.000000001`, when found as operand of one of the four elementary operations is kept with `D+2` digits, and even more for the power function.

```
\xintDigits:= 9; \xintthefloatexpr (1+1e-9)-1\relax=0.e0
\xintDigits:= 9; \xintthefloatexpr 1.000000001-1\relax=1.00000000e-9
```

For the fun of it: `\xintDigits:=20;`

```
\xintthefloatexpr (1+1e-7)^1e7\relax=2.7182816925449662712e0
```

```
\xintDigits:=36;
\xintthefloatexpr ((1/13+1/121)*(1/179-1/173))/(1/19-1/18)\relax
5.64487459334466559166166079096852897e-3
\xintFloat{\xinttheexpr ((1/13+1/121)*(1/179-1/173))/(1/19-1/18)\relax}
5.64487459334466559166166079096852912e-3
```

The latter result is the rounding of the exact result. The previous one has rounding errors coming from the various roundings done for each sub-expression. It was a bit funny to discover that `maple`, configured with `Digits:=36;` and with decimal dots everywhere to let it input the numbers as floats, gives exactly the same result with the same rounding errors as does `\xintthefloatexpr`!

Using `\xintthefloatexpr` only pays off compared to using `\xinttheexpr` followed with `\xintFloat` if the computations turn out to involve hundreds of digits. For elementary calculations with hand written numbers (not using the scientific notation with exponents differing greatly) it will generally be more efficient to use `\xinttheexpr`. The situation is quickly otherwise if one starts using the Power function. Then, `\xintthefloat` is often useful; and sometimes indispensable to achieve the (approximate) computation in reasonable time.

We can try some crazy things:

```
\xintDigits:=12;\xintthefloatexpr 1.000000000000001^1e15\relax
2.71828182846e0
```

Contrarily to some professional computing sofware which are our concurrents on this market, the `1.000000000000001` wasn't rounded to 1 despite the setting of `\xintDigits`; it would have been if we had input it as `(1+1e-15)`.

26.11 \xintifboolexpr

x n n ★ `\xintifboolexpr{<expr>}{YES}{NO}` does `\xinttheexpr <expr>\relax` and then executes the `YES` or the `NO` branch depending on whether the outcome was non-zero or zero. `<expr>` can involove various `&` and `|`, parentheses, `all`, `any`, `xor`, the `bool` or `togl` operators, but is not limited to them: the most general computation can be done, the test is on whether the outcome of the computation vanishes or not.

Will not work on an expression composed of comma separated sub-expressions.

26.12 \xintifboolfloatexpr

x n n ★ \xintifboolfloatexpr{<expr>}{YES}{NO} does \xintthefloatexpr <expr>\relax and then executes the YES or the NO branch depending on whether the outcome was non zero or zero.

26.13 \xintifbooliiexpr

x n n ★ \xintifbooliiexpr{<expr>}{YES}{NO} does \xinttheiiexpr <expr>\relax and then executes the YES or the NO branch depending on whether the outcome was non zero or zero.

26.14 \xintNewFloatExpr

This is exactly like \xintNewExpr except that the created formulas are set-up to use \xintthefloatexpr. The precision used for numbers fetched as parameters will be the one locally given by \xintDigits at the time of use of the created formulas, not \xintNewFloatExpr. However, the numbers hard-wired in the original expression will have been evaluated with the then current setting for \xintDigits.

26.15 \xintNewIExpr

Like \xintNewExpr but using \xinttheiexpr. Former denomination was \xintNewNumExpr which is deprecated and should not be used.

26.16 \xintNewIIExpr

Like \xintNewExpr but using \xinttheiiexpr.

26.17 \xintNewBoolExpr

Like \xintNewExpr but using \xinttheboolexpr.

26.18 Technicalities

As already mentioned \xintNewExpr\myformula[n] does not check the prior existence of a macro \myformula. And the number of parameters n given as mandatory argument withing square brackets should be (at least) equal to the number of parameters in the expression.

Obviously I should mention that \xintNewExpr itself can not be used in an expansion-only context, as it creates a macro.

The \escapechar setting may be arbitrary when using \xintexpr.

The format of the output of \xintexpr⟨*stuff*⟩\relax is a ! (with catcode 11) followed by \XINT_expr_usethe which prints an error message in the document and in the log file if it is executed, then a \xint_protect token, a token doing the actual printing and finally a token \.=A/B[n]. Using \xinttheexpr means zapping the first three things, the fourth one will then unlock A/B[n] from the (presumably undefined, but it does not matter) control sequence \.=A/B[n].

Thanks to the release 1.09j added \xint_protect token and the fact that \XINT_expr_usethe is \protected, one can now use \xintexpr inside an \edef, with no need of the \xintthe prefix.

Note that \xintexpr is thus compatible with complete expansion, contrarily to \numexpr which is non-expandable, if not prefixed by \the or \number, and away from contexts where TeX is building a number. See subsection 23.22 for some illustration.

I decided to put all intermediate results (from each evaluation of an infix operators, or of a parenthesized subpart of the expression, or from application of the minus as prefix, or of the exclamation sign as postfix, or any encountered braced material) inside \csname... \endcsname, as this can be done expandably and encapsulates an arbitrarily long fraction in a single token (left with undefined meaning), thus providing tremendous relief to the programmer in his/her expansion control.

As the \xintexpr computations corresponding to functions and infix or postfix operators are done inside \csname...\endcsname, the *f*-expandability could possibly be dropped and one could imagine implementing the basic operations with expandable but not *f*-expandable macros (as \xintXTrunc.) I have not investigated that possibility.

Syntax errors in the input such as using a one-argument function with two arguments will generate low-level TeX processing unrecoverable errors, with cryptic accompanying message.

Some other problems will give rise to 'error messages' macros giving some indication on the location and nature of the problem. Mainly, an attempt has been made to handle gracefully missing or extraneous parentheses.

When the scanner is looking for a number and finds something else not otherwise treated, it assumes it is the start of the function name and will expand forward in the hope of hitting an opening parenthesis; if none is found at least it should stop when encountering the \relax marking the end of the expressions.

Note that \relax is mandatory (contrarily to a \numexpr).

26.19 Acknowledgements

I was greatly helped in my preparatory thinking, prior to producing such an expandable parser, by the commented source of the l3fp package, specifically the l3fp-parse.dtx file (in the version of April-May 2013). Also the source of the calc package was instructive, despite the fact that here for \xintexpr the principles are necessarily different due to the aim of achieving expandability.

27 Commands of the xintbinhex package

This package was first included in the 1.08 release of xint. It provides expandable conversions of arbitrarily long numbers to and from binary and hexadecimal.

The argument is first f-expanded. It then may start with an optional minus sign (unique, of category code other), followed with optional leading zeroes (arbitrarily many, category code other) and then “digits” (hexadecimal letters may be of category code letter or other, and must be uppercased). The optional (unique) minus sign (plus sign is not allowed) is kept in the output. Leading zeroes are allowed, and stripped. The hexadecimal letters on output are of category code letter, and uppercased.

Contents

27.1 \xintDecToHex

$f\star$ Converts from decimal to hexadecimal.

```
\xintDecToHex{2718281828459045235360287471352662497757247093699959574
966967627724076630353547594571382178525166427427466391932003}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F4
6DCE46C6032936BF37DAC918814C63
```

27.2 \xintDecToBin

$f\star$ Converts from decimal to binary.

```
\xintDecToBin{2718281828459045235360287471352662497757247093699959574
966967627724076630353547594571382178525166427427466391932003}
->10001101010010011100101111100011001101001010010011010100101110000
010100011111011111010000101010000000101111001000101001110001111000001
011000101111110001000001101100010001110001001000101110101110111100101
011010101110110000010111011001110001101001001110010111101000110110
111001110010001101100011000000011001010010011011010111111001101111101
1010110010010001100010000000101001100011000 11
```

27.3 \xintHexToDec

$f\star$ Converts from hexadecimal to decimal.

```
\xintHexToDec{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B576
0BB38D272F46DCE46C6032936BF37DAC918814C63}
->27182818284590452353602874713526624977572470936999595749669676277
2407663035354759457138217852516642742746639193200 3
```

27.4 \xintBinToDec

$f\star$ Converts from binary to decimal.

```
\xintBinToDec{1000110101001001110010111110001100110100101001001101010
0101110000010100011111011111010000101010000000101111001000101001110001 11
```

```
11000001011000101111100010000011011000100011100010010001011101011101111
00101011010101110110000010111011001110001101001001110010111101000110110
11100111001000110110001100000001100101001001101101011111100110111110110
101100100100011000100000010100110001100011}
->2718281828459045235360287471352662497757247093699959574966967627724
076630353547594571382178525166427427466391932003
```

27.5 \xintBinToHex

f ★ Converts from binary to hexadecimal.

```
\xintBinToHex{1000110101001001110010111110001100110100101001001101010
01011100000101000111110111110100001010100000010111100100010100111000111
11000001011000101111100010000011011000100011100010010001011101011101111
00101011010101110110000010111011001110001101001001110010111101000110110
11100111001000110110001100000001100101001001101101011111100110111110110
101100100100011000100000010100110001100011}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F4
6DCE46C6032936BF37DAC918814C63
```

27.6 \xintHexToBin

f ★ Converts from hexadecimal to binary.

```
\xintHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B576
0BB38D272F46DCE46C6032936BF37DAC918814C63}
->100011010100100111001011111000110011010010100100110101001011100000
10100011111011111010000101010000001011110010001010011100011111000001
01100010111110001000001101100010001110001001000101110101110111100101
01101010111011000001011101100111000110100100111001011110100011011011
10011100100011011000110000000011001010010011011010111111001101111011
0101100100100011000100000010100110001100011
```

27.7 \xintCHexToBin

f ★ Also converts from hexadecimal to binary. Faster on inputs with at least one hundred hexadecimal digits.

```
\xintCHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B57
60BB38D272F46DCE46C6032936BF37DAC918814C63}
->100011010100100111001011111000110011010010100100110101001011100000
10100011111011111010000101010000001011110010001010011100011111000001
01100010111110001000001101100010001110001001000101110101110111100101
01101010111011000001011101100111000110100100111001011110100011011011
10011100100011011000110000000011001010010011011010111111001101111011
0101100100100011000100000010100110001100011
```

28 Commands of the xintgcd package

This package was included in the original release `1.0` of the **xint** bundle.

Since release `1.09a` the macros filter their inputs through the `\xintNum` macro, so one can use count registers, or fractions as long as they reduce to integers.

Contents

28.1 `\xintGCD`

$\overset{\text{Num}}{f}\ \overset{\text{Num}}{f}\ \star$

`\xintGCD{N}{M}` computes the greatest common divisor. It is positive, except when both `N` and `M` vanish, in which case the macro returns zero.

`\xintGCD{10000}{1113}=1`

`\xintGCD{123456789012345}{9876543210321}=3`

28.2 `\xintGCDof`

$f \rightarrow * \overset{\text{Num}}{f}\ \star$

`\xintGCDof{{a}{b}{c}...}` computes the greatest common divisor of all integers `a`, `b`, ... The list argument may be a macro, it is f-expanded first and must contain at least one item.

28.3 `\xintLCM`

$\overset{\text{Num}}{f}\ \overset{\text{Num}}{f}\ \star$

`\xintGCD{N}{M}` computes the least common multiple. It is `0` if one of the two integers vanishes.

28.4 `\xintLCMof`

$f \rightarrow * \overset{\text{Num}}{f}\ \star$

`\xintLCMof{{a}{b}{c}...}` computes the least common multiple of all integers `a`, `b`, ... The list argument may be a macro, it is f-expanded first and must contain at least one item.

28.5 `\xintBezout`

$\overset{\text{Num}}{f}\ \overset{\text{Num}}{f}\ \star$

`\xintBezout{N}{M}` returns five numbers `A`, `B`, `U`, `V`, `D` within braces. `A` is the first (expanded, as usual) input number, `B` the second, `D` is the GCD, and `UA - VB = D`.

```
\xintAssign {{\xintBezout {10000}{1113}}}\to\X
\meaning\X: macro:->\xintBezout {10000}{1113}.
\xintAssign {\xintBezout {10000}{1113}}\to\A\B\U\V\D
\A: 10000, \B: 1113, \U: -131, \V: -1177, \D: 1.
\xintAssign {\xintBezout {123456789012345}{9876543210321}}\to\A\B\U\V\D
\A: 123456789012345, \B: 9876543210321, \U: 256654313730, \V: 3208178892607,
\D: 3.
```

28.6 \xintEuclideAlgorithm

Num Num
f f ★

`\xintEuclideAlgorithm{N}{M}` applies the Euclide algorithm and keeps a copy of all quotients and remainders.

```
\xintAssign {{\xintEuclideAlgorithm {10000}{1113}}}\to\X
\meaning\X: macro:->\xintEuclideAlgorithm {10000}{1113}.
```

The first token is the number of steps, the second is `N`, the third is the GCD, the fourth is `M` then the first quotient and remainder, the second quotient and remainder, … until the final quotient and last (zero) remainder.

28.7 \xintBezoutAlgorithm

Num Num
f f ★

`\xintBezoutAlgorithm{N}{M}` applies the Euclide algorithm and keeps a copy of all quotients and remainders. Furthermore it computes the entries of the successive products of the 2 by 2 matrices $\begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}$ formed from the quotients arising in the algorithm.

```
\xintAssign {{\xintEuclideAlgorithm {10000}{1113}}}\to\X
\meaning\X: macro:->\xintBezoutAlgorithm {10000}{1113}.
```

The first token is the number of steps, the second is `N`, then `0`, `1`, the GCD, `M`, `1`, `0`, the first quotient, the first remainder, the top left entry of the first matrix, the bottom left entry, and then these four things at each step until the end.

28.8 \xintTypesetEuclideAlgorithm

Num Num
f f

This macro is just an example of how to organize the data returned by `\xintEuclideAlgorithm`. Copy the source code to a new macro and modify it to what is needed.

```
\xintTypesetEuclideAlgorithm {123456789012345}{9876543210321}
```

$$\begin{aligned}
123456789012345 &= 12 \times 9876543210321 + 4938270488493\\
9876543210321 &= 2 \times 4938270488493 + 2233335\\
4938270488493 &= 2211164 \times 2233335 + 536553\\
2233335 &= 4 \times 536553 + 87123\\
536553 &= 6 \times 87123 + 13815\\
87123 &= 6 \times 13815 + 4233\\
13815 &= 3 \times 4233 + 1116\\
4233 &= 3 \times 1116 + 885\\
1116 &= 1 \times 885 + 231\\
885 &= 3 \times 231 + 192\\
231 &= 1 \times 192 + 39\\
192 &= 4 \times 39 + 36\\
39 &= 1 \times 36 + 3\\
36 &= 12 \times 3 + 0
\end{aligned}$$

28.9 \xintTypesetBezoutAlgorithm

Num Num
f f

This macro is just an example of how to organize the data returned by `\xintBezoutAlgorithm`. Copy the source code to a new macro and modify it to what is needed.

```
\xintTypesetBezoutAlgorithm {10000}{1113}
```

$$\begin{aligned}
10000 &= 8\times 1113+1096\\
8 &= 8\times 1+0\\
1 &= 8\times 0+1\\
1113 &= 1\times 1096+17\\
9 &= 1\times 8+1\\
1 &= 1\times 1+0\\
1096 &= 64\times 17+8\\
584 &= 64\times 9+8\\
65 &= 64\times 1+1\\
17 &= 2\times 8+1\\
1177 &= 2\times 584+9\\
131 &= 2\times 65+1\\
8 &= 8\times 1+0\\
10000 &= 8\times 1177+584\\
1113 &= 8\times 131+65\\
131\times 10000 &- 1177\times 1113 = -1
\end{aligned}$$

29 Commands of the xintseries package

Some arguments to the package commands are macros which are expanded only later, when given their parameters. The arguments serving as indices are systematically given to a `\numexpr` expressions (new with `1.06`!) , hence f-expanded, they may be count registers, etc...

This package was first released with version `1.03` of the **xint** bundle.

We use $\overset{\text{Frac}}{f}$ for the expansion type of various macro arguments, but if only **xint** and not **xintfrac** is loaded this should be more appropriately $\overset{\text{Num}}{f}$. The macro `\xintiSeries` is special and expects summing big integers obeying the strict format, even if **xintfrac** is loaded.

Contents

29.1 `\xintSeries`

$\overset{\text{num}}{x}\ \overset{\text{num}}{x}\ \overset{\text{Frac}}{f}\ \star$ `\xintSeries{A}{B}{\coeff}` computes $\sum_{n=A}^{n=B}$ `\coeff{n}`. The initial and final indices must obey the `\numexpr` constraint of expanding to numbers at most `2^31-1`. The `\coeff` macro must be a one-parameter f-expandable command, taking on input an explicit number `n` and producing some number or fraction `\coeff{n}`; it is expanded at the time it is needed.[72]

```
\def\coeff #1{\xintiiMON{#1}/#1.5} % (-1)^n/(n+1/2)
\edef\w {\xintSeries {0}{50}{\coeff}} % we want to re-use it
\edef\z {\xintJrr {\w}[0]} % the [0] for a microsecond gain.
% \xintJrr preferred to \xintIrr: a big common factor is suspected.
% But numbers much bigger would be needed to show the greater efficiency.
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} = \xintFrac\z \]
```

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} = \frac{173909338287370940432112792101626602278714}{110027467159390003025279917226039729050575}$$

For info, before action by `\xintJrr` the inner representation of the result has a denominator of `\xintLen {\xintDenominator\w}=117` digits. This troubled me as `101!!` has only 81 digits: `\xintLen {\xintQuo {\xintFac {101}}{\xintiMul {\xintiPow {2}{50}}{\xintFac{50}}}}=81`. The explanation lies in the too clever to be efficient `#1.5` trick. It leads to a silly extra `5^{51}` (which has 36 digits) in the denominator. See the explanations in the next section.

> Note: as soon as the coefficients look like factorials, it is more efficient to use the `\xintRationalSeries` macro whose evaluation will avoid a denominator build-up; indeed the raw operations of addition and subtraction of fractions blindly multiply out denominators. So the raw evaluation of $\sum_{n=0}^{N} 1/n!$ with `\xintSeries` will have a denominator equal to $\prod_{n=0}^{N} n!$. Needless to say this makes it more difficult to compute the exact value of this sum with `N=50`, for example, whereas with `\xintRationalSeries` the denominator does not get bigger than 50!.
>
> For info: by the way $\prod_{n=0}^{50} n!$ is easily computed by **xint** and is a number with 1394 digits. And $\prod_{n=0}^{100} n!$ is also computable by **xint** (24 seconds on my laptop for the brute force iterated multiplication of all factorials, a specialized routine would do it faster) and has 6941 digits (this means more than two pages if printed...). Whereas 100! only has 158 digits.

```
\def\coeffleibnitz #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}
\cnta 1
\loop  % in this loop we recompute from scratch each partial sum!
% we can afford that, as \xintSeries is fast enough.
\noindent\hbox to 2em{\hfil\texttt{\the\cnta.} }%
         \xintTrunc {12}
                    {\xintSeries {1}{\cnta}{\coeffleibnitz}}\dots
\endgraf
\ifnum\cnta < 30 \advance\cnta 1 \repeat
```

1. 1.000000000000...	6. 0.616666666666...	11. 0.736544011544...
2. 0.500000000000...	7. 0.759523809523...	12. 0.653210678210...
3. 0.833333333333...	8. 0.634523809523...	13. 0.730133755133...
4. 0.583333333333...	9. 0.745634920634...	14. 0.658705183705...
5. 0.783333333333...	10. 0.645634920634...	15. 0.725371850371...

[72] `\xintiiMON` is like `\xintMON` but does not parse its argument through `\xintNum`, for efficiency; other macros of this type are `\xintiiAdd`, `\xintiiMul`, `\xintiiSum`, `\xintiiPrd`, `\xintiiMMON`, `\xintiiLDg`, `\xintiiFDg`, `\xintiiOdd`, ...

16. 0.662871850371...	21. 0.716390450794...	26. 0.674285961081...
17. 0.721695379783...	22. 0.670935905339...	27. 0.711322998118...
18. 0.666139824228...	23. 0.714414166209...	28. 0.675608712404...
19. 0.718771403175...	24. 0.672747499542...	29. 0.710091471024...
20. 0.668771403175...	25. 0.712747499542...	30. 0.676758137691...

29.2 \xintiSeries

num num
x x f ★ \xintiSeries{A}{B}{\coeff} computes $\sum_{n=A}^{n=B}$ \coeff{n} where \coeff{n} must f-expand to a (possibly long) integer in the strict format.

```
\def\coeff #1{\xintiTrunc {40}{\xintMON{#1}/#1.5}}%
% better:
\def\coeff #1{\xintiTrunc {40}
   {\the\numexpr 2*\xintiiMON{#1}\relax/\the\numexpr 2*#1+1\relax [0]}}%
% better still:
\def\coeff #1{\xintiTrunc {40}
 {\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}}%
% (-1)^n/(n+1/2) times 10^40, truncated to an integer.
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} \approx
        \xintTrunc {40}{\xintiSeries {0}{50}{\coeff}[-40]}\dots\]
```

The #1.5 trick to define the \coeff macro was neat, but 1/3.5, for example, turns internally into 10/35 whereas it would be more efficient to have 2/7. The second way of coding the wanted coefficient avoids a superfluous factor of five and leads to a faster evaluation. The third way is faster, after all there is no need to use \xintMON (or rather \xintiiMON which has less parsing overhead) on integers obeying the TEX bound. The denominator having no sign, we have added the [0] as this speeds up (infinitesimally) the parsing.

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} \approx 1.5805993064935250412367895069567264144810$$

We should have cut out at least the last two digits: truncating errors originating with the first coefficients of the sum will never go away, and each truncation introduces an uncertainty in the last digit, so as we have 40 terms, we should trash the last two digits, or at least round at 38 digits. It is interesting to compare with the computation where rounding rather than truncation is used, and with the decimal expansion of the exactly computed partial sum of the series:

```
\def\coeff #1{\xintiRound {40} % rounding at 40
  {\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}}%
% (-1)^n/(n+1/2) times 10^40, rounded to an integer.
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} \approx
        \xintTrunc {40}{\xintiSeries {0}{50}{\coeff}[-40]}\]
\def\exactcoeff #1%
  {\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}%
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12}
   = \xintTrunc {50}{\xintSeries {0}{50}{\exactcoeff}}\dots\]
```

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} \approx 1.5805993064935250412367895069567264144804$$

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}} = 1.580599306493525041236789506956726414480686802883 67\dots$$

This shows indeed that our sum of truncated terms estimated wrongly the 39th and 40th digits of the exact result[73] and that the sum of rounded terms fared a bit better.

29.3 `\xintRationalSeries`

$\overset{\text{num}}{x}$ $\overset{\text{num}}{x}$ $\overset{\text{Frac}}{f}$ $\overset{\text{Frac}}{f}$ ★

`\xintRationalSeries{A}{B}{f}{\ratio}` evaluates $\sum_{\mathtt{n=A}}^{\mathtt{n=B}}$ `F(n)`, where `F(n)` is specified indirectly via the data of `f=F(A)` and the one-parameter macro `\ratio` which must be such that `\macro{n}` expands to `F(n)/F(n-1)`. The name indicates that `\xintRationalSeries` was designed to be useful in the cases where `F(n)/F(n-1)` is a rational function of `n` but it may be anything expanding to a fraction. The macro `\ratio` must be an expandable-only compatible command and expand to its value after iterated full expansion of its first token. `A` and `B` are fed to a `\numexpr` hence may be count registers or arithmetic expressions built with such; they must obey the TeX bound. The initial term `f` may be a macro `\f`, it will be expanded to its value representing `F(A)`.

```
\def\ratio #1{2/#1[0]}% 2/n, to compute exp(2)
\cnta 0 % previously declared count
\loop \edef\z {\xintRationalSeries {0}{\cnta}{1}{\ratio }}%
\noindent$\sum_{n=0}^{\the\cnta} \frac{2^n}{n!}=
           \xintTrunc{12}\z\dots=
           \xintFrac\z=\xintFrac{\xintIrr\z}$\vtop to 5pt{}\endgraf
\ifnum\cnta<20 \advance\cnta 1 \repeat
```

$\sum_{n=0}^{0} \frac{2^n}{n!} = 1.000000000000\dots = 1 = 1$

$\sum_{n=0}^{1} \frac{2^n}{n!} = 3.000000000000\dots = 3 = 3$

$\sum_{n=0}^{2} \frac{2^n}{n!} = 5.000000000000\dots = \frac{10}{2} = 5$

$\sum_{n=0}^{3} \frac{2^n}{n!} = 6.333333333333\dots = \frac{38}{6} = \frac{19}{3}$

$\sum_{n=0}^{4} \frac{2^n}{n!} = 7.000000000000\dots = \frac{168}{24} = 7$

$\sum_{n=0}^{5} \frac{2^n}{n!} = 7.266666666666\dots = \frac{872}{120} = \frac{109}{15}$

$\sum_{n=0}^{6} \frac{2^n}{n!} = 7.355555555555\dots = \frac{5296}{720} = \frac{331}{45}$

$\sum_{n=0}^{7} \frac{2^n}{n!} = 7.380952380952\dots = \frac{37200}{5040} = \frac{155}{21}$

$\sum_{n=0}^{8} \frac{2^n}{n!} = 7.387301587301\dots = \frac{297856}{40320} = \frac{2327}{315}$

$\sum_{n=0}^{9} \frac{2^n}{n!} = 7.388712522045\dots = \frac{2681216}{362880} = \frac{20947}{2835}$

$\sum_{n=0}^{10} \frac{2^n}{n!} = 7.388994708994\dots = \frac{26813184}{3628800} = \frac{34913}{4725}$

$\sum_{n=0}^{11} \frac{2^n}{n!} = 7.389046015712\dots = \frac{294947072}{39916800} = \frac{164591}{22275}$

$\sum_{n=0}^{12} \frac{2^n}{n!} = 7.389054566832\dots = \frac{3539368960}{479001600} = \frac{691283}{93555}$

$\sum_{n=0}^{13} \frac{2^n}{n!} = 7.389055882389\dots = \frac{46011804672}{6227020800} = \frac{14977801}{2027025}$

$\sum_{n=0}^{14} \frac{2^n}{n!} = 7.389056070325\dots = \frac{644165281792}{87178291200} = \frac{314533829}{42567525}$

$\sum_{n=0}^{15} \frac{2^n}{n!} = 7.389056095384\dots = \frac{9662479259648}{1307674368000} = \frac{4718007451}{638512875}$

$\sum_{n=0}^{16} \frac{2^n}{n!} = 7.389056098516\dots = \frac{154599668219904}{20922789888000} = \frac{1572669151}{212837625}$

$\sum_{n=0}^{17} \frac{2^n}{n!} = 7.389056098884\dots = \frac{2628194359869440}{355687428096000} = \frac{16041225341}{2170943775}$

[73] as the series is alternating, we can roughly expect an error of $\sqrt{40}$ and the last two digits are off by 4 units, which is not contradictory to our expectations.

$\sum_{n=0}^{18} \frac{2^n}{n!} = 7.389056098925\cdots = \frac{4730749847791206 4}{6402373705728000} = \frac{103122162907}{13956067125}$
$\sum_{n=0}^{19} \frac{2^n}{n!} = 7.389056098930\cdots = \frac{89884247108085350 4}{121645100408832000} = \frac{4571749222213}{618718975875}$
$\sum_{n=0}^{20} \frac{2^n}{n!} = 7.389056098930\cdots = \frac{17976849421618118656}{2432902008176640000} = \frac{68576238333199}{9280784638125}$

Such computations would become quickly completely inaccessible via the \xintSeries macros, as the factorials in the denominators would get all multiplied together: the raw addition and subtraction on fractions just blindly multiplies denominators! Whereas \xintRationalSeries evaluate the partial sums via a less silly iterative scheme.

```
\def\ratio #1{-1/#1[0]}% -1/n, comes from the series of exp(-1)
\cnta 0 % previously declared count
\loop
\edef\z {\xintRationalSeries {0}{\cnta}{1}{\ratio }}%
\noindent$\sum_{n=0}^{\the\cnta} \frac{(-1)^n}{n!}=
           \xintTrunc{20}\z\dots=\xintFrac{\z}=\xintFrac{\xintIrr\z}$
         \vtop to 5pt{}\endgraf
\ifnum\cnta<20 \advance\cnta 1 \repeat
```

$\sum_{n=0}^{0} \frac{(-1)^n}{n!} = 1.00000000000000000000\cdots = 1 = 1$
$\sum_{n=0}^{1} \frac{(-1)^n}{n!} = 0\cdots = 0 = 0$
$\sum_{n=0}^{2} \frac{(-1)^n}{n!} = 0.50000000000000000000\cdots = \frac{1}{2} = \frac{1}{2}$
$\sum_{n=0}^{3} \frac{(-1)^n}{n!} = 0.33333333333333333333\cdots = \frac{2}{6} = \frac{1}{3}$
$\sum_{n=0}^{4} \frac{(-1)^n}{n!} = 0.37500000000000000000\cdots = \frac{9}{24} = \frac{3}{8}$
$\sum_{n=0}^{5} \frac{(-1)^n}{n!} = 0.36666666666666666666\cdots = \frac{44}{120} = \frac{11}{30}$
$\sum_{n=0}^{6} \frac{(-1)^n}{n!} = 0.36805555555555555555\cdots = \frac{265}{720} = \frac{53}{144}$
$\sum_{n=0}^{7} \frac{(-1)^n}{n!} = 0.36785714285714285714\cdots = \frac{1854}{5040} = \frac{103}{280}$
$\sum_{n=0}^{8} \frac{(-1)^n}{n!} = 0.36788194444444444444\cdots = \frac{14833}{40320} = \frac{2119}{5760}$
$\sum_{n=0}^{9} \frac{(-1)^n}{n!} = 0.36787918871252204585\cdots = \frac{133496}{362880} = \frac{16687}{45360}$
$\sum_{n=0}^{10} \frac{(-1)^n}{n!} = 0.36787946428571428571\cdots = \frac{1334961}{3628800} = \frac{16481}{44800}$
$\sum_{n=0}^{11} \frac{(-1)^n}{n!} = 0.36787943923360590027\cdots = \frac{14684570}{39916800} = \frac{1468457}{3991680}$
$\sum_{n=0}^{12} \frac{(-1)^n}{n!} = 0.36787944132128159905\cdots = \frac{176214841}{479001600} = \frac{16019531}{43545600}$
$\sum_{n=0}^{13} \frac{(-1)^n}{n!} = 0.36787944116069116069\cdots = \frac{2290792932}{6227020800} = \frac{63633137}{172972800}$
$\sum_{n=0}^{14} \frac{(-1)^n}{n!} = 0.36787944117216190628\cdots = \frac{32071101049}{87178291200} = \frac{2467007773}{6706022400}$
$\sum_{n=0}^{15} \frac{(-1)^n}{n!} = 0.36787944117139718991\cdots = \frac{481066515734}{1307674368000} = \frac{34361893981}{93405312000}$
$\sum_{n=0}^{16} \frac{(-1)^n}{n!} = 0.36787944117144498468\cdots = \frac{7697064251745}{20922789888000} = \frac{15549624751}{42268262400}$
$\sum_{n=0}^{17} \frac{(-1)^n}{n!} = 0.36787944117144217323\cdots = \frac{130850092279664}{355687428096000} = \frac{8178130767479}{22230464256000}$
$\sum_{n=0}^{18} \frac{(-1)^n}{n!} = 0.36787944117144232942\cdots = \frac{2355301661033953}{6402373705728000} = \frac{138547156531409}{376610217984000}$
$\sum_{n=0}^{19} \frac{(-1)^n}{n!} = 0.36787944117144232120\cdots = \frac{44750731559645106}{121645100408832000} = \frac{92079694567171}{250298560512000}$
$\sum_{n=0}^{20} \frac{(-1)^n}{n!} = 0.36787944117144232161\cdots = \frac{895014631192902121}{2432902008176640000} = \frac{4282366656425369}{11640679464960000}$

We can incorporate an indeterminate if we define \ratio to be a macro with two parameters: \def\ratioexp #1#2{\xintDiv{#1}{#2}}% x/n: x=#1, n=#2. Then, if \x expands to some fraction x, the command

```
\xintRationalSeries {0}{b}{1}{\ratioexp{\x}}
```

will compute $\sum_{n=0}^{n=b} x^n/n!$:

```
\cnta 0
\def\ratioexp #1#2{\xintDiv{#1}{#2}}% #1/#2
\loop
\noindent
$\sum_{n=0}^{\the\cnta} (.57)^n/n! = \xintTrunc {50}
     {\xintRationalSeries {0}{\cnta}{1}{\ratioexp{.57}}}\dots$
     \vtop to 5pt {}\endgraf
\ifnum\cnta<50 \advance\cnta 10 \repeat
```

$\sum_{n=0}^{0}(.57)^n/n! = 1.00000000000000000000000000000000000000000000000000\dots$
$\sum_{n=0}^{10}(.57)^n/n! = 1.76826705137947002480668058035714285714285714285714\dots$
$\sum_{n=0}^{20}(.57)^n/n! = 1.76826705143373515162089324271187082272833005529082\dots$
$\sum_{n=0}^{30}(.57)^n/n! = 1.76826705143373515162089339282382144915484884979430\dots$
$\sum_{n=0}^{40}(.57)^n/n! = 1.76826705143373515162089339282382144915485219867776\dots$
$\sum_{n=0}^{50}(.57)^n/n! = 1.76826705143373515162089339282382144915485219867776\dots$

Observe that in this last example the `x` was directly inserted; if it had been a more complicated explicit fraction it would have been worthwile to use `\ratioexp\x` with `\x` defined to expand to its value. In the further situation where this fraction `x` is not explicit but itself defined via a complicated, and time-costly, formula, it should be noted that `\xintRationalSeries` will do again the evaluation of `\x` for each term of the partial sum. The easiest is thus when `x` can be defined as an `\edef`. If however, you are in an expandable-only context and cannot store in a macro like `\x` the value to be used, a variant of `\xintRationalSeries` is needed which will first evaluate this `\x` and then use this result without recomputing it. This is `\xintRationalSeriesX`, documented next.

Here is a slightly more complicated evaluation:

```
\cnta 1
\loop \edef\z {\xintRationalSeries
                    {\cnta}
                    {2*\cnta-1}
                    {\xintiPow {\the\cnta}{\cnta}/\xintFac{\cnta}}
                    {\ratioexp{\the\cnta}}}%
\edef\w {\xintRationalSeries {0}{2*\cnta-1}{1}{\ratioexp{\the\cnta}}}%
\noindent
$\sum_{n=\the\cnta}^{\the\numexpr 2*\cnta-1\relax} \frac{\the\cnta^n}{n!}/%
          \sum_{n=0}^{\the\numexpr 2*\cnta-1\relax} \frac{\the\cnta^n}{n!} =
          \xintTrunc{8}{\xintDiv\z\w}\dots$ \vtop to 5pt{}\endgraf
\ifnum\cnta<20 \advance\cnta 1 \repeat
```

$\sum_{n=1}^{1} \frac{1^n}{n!}/\sum_{n=0}^{1} \frac{1^n}{n!} = 0.50000000\dots$
$\sum_{n=2}^{3} \frac{2^n}{n!}/\sum_{n=0}^{3} \frac{2^n}{n!} = 0.52631578\dots$
$\sum_{n=3}^{5} \frac{3^n}{n!}/\sum_{n=0}^{5} \frac{3^n}{n!} = 0.53804347\dots$
$\sum_{n=4}^{7} \frac{4^n}{n!}/\sum_{n=0}^{7} \frac{4^n}{n!} = 0.54317053\dots$
$\sum_{n=5}^{9} \frac{5^n}{n!}/\sum_{n=0}^{9} \frac{5^n}{n!} = 0.54502576\dots$
$\sum_{n=6}^{11} \frac{6^n}{n!}/\sum_{n=0}^{11} \frac{6^n}{n!} = 0.54518217\dots$
$\sum_{n=7}^{13} \frac{7^n}{n!}/\sum_{n=0}^{13} \frac{7^n}{n!} = 0.54445274\dots$
$\sum_{n=8}^{15} \frac{8^n}{n!}/\sum_{n=0}^{15} \frac{8^n}{n!} = 0.54327992\dots$
$\sum_{n=9}^{17} \frac{9^n}{n!}/\sum_{n=0}^{17} \frac{9^n}{n!} = 0.54191055\dots$
$\sum_{n=10}^{19} \frac{10^n}{n!}/\sum_{n=0}^{19} \frac{10^n}{n!} = 0.54048295\dots$
$\sum_{n=11}^{21} \frac{11^n}{n!}/\sum_{n=0}^{21} \frac{11^n}{n!} = 0.53907332\dots$
$\sum_{n=12}^{23} \frac{12^n}{n!}/\sum_{n=0}^{23} \frac{12^n}{n!} = 0.53772178\dots$
$\sum_{n=13}^{25} \frac{13^n}{n!}/\sum_{n=0}^{25} \frac{13^n}{n!} = 0.53644744\dots$
$\sum_{n=14}^{27} \frac{14^n}{n!}/\sum_{n=0}^{27} \frac{14^n}{n!} = 0.53525726\dots$
$\sum_{n=15}^{29} \frac{15^n}{n!}/\sum_{n=0}^{29} \frac{15^n}{n!} = 0.53415135\dots$
$\sum_{n=16}^{31} \frac{16^n}{n!}/\sum_{n=0}^{31} \frac{16^n}{n!} = 0.53312615\dots$
$\sum_{n=17}^{33} \frac{17^n}{n!}/\sum_{n=0}^{33} \frac{17^n}{n!} = 0.53217628\dots$
$\sum_{n=18}^{35} \frac{18^n}{n!}/\sum_{n=0}^{35} \frac{18^n}{n!} = 0.53129566\dots$
$\sum_{n=19}^{37} \frac{19^n}{n!}/\sum_{n=0}^{37} \frac{19^n}{n!} = 0.53047810\dots$
$\sum_{n=20}^{39} \frac{20^n}{n!}/\sum_{n=0}^{39} \frac{20^n}{n!} = 0.52971771\dots$

29.4 \xintRationalSeriesX

num num Frac Frac
x x f f f ★

\xintRationalSeriesX{A}{B}{\first}{\ratio}{\g} is a parametrized version of \xintRationalSeries where \first is now a one-parameter macro such that \first {\g} gives the initial term and \ratio is a two-parameter macro such that \ratio{n}{ \g} represents the ratio of one term to the previous one. The parameter \g is evaluated only once at the beginning of the computation, and can thus itself be the yet unevaluated result of a previous computation.

Let \ratio be such a two-parameter macro; note the subtle differences between

\xintRationalSeries {A}{B}{\first}{\ratio{\g}}

and \xintRationalSeriesX {A}{B}{\first}{\ratio}{\g}.

First the location of braces differ... then, in the former case \first is a *no-parameter* macro expanding to a fractional number, and in the latter, it is a *one-parameter* macro which will use \g. Furthermore the X variant will expand \g at the very beginning whereas the former non-X former variant will evaluate it each time it needs it (which is bad if this evaluation is time-costly, but good if \g is a big explicit fraction encapsulated in a macro).

The example will use the macro \xintPowerSeries which computes efficiently exact partial sums of power series, and is discussed in the next section.

```
\def\firstterm #1{1[0]}% first term of the exponential series
% although it is the constant 1, here it must be defined as a
% one-parameter macro. Next comes the ratio function for exp:
\def\ratioexp  #1#2{\xintDiv {#1}{#2}}% x/n
% These are the (-1)^{n-1}/n of the log(1+h) series:
\def\coefflog #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}%
% Let L(h) be the first 10 terms of the log(1+h) series and
% let E(t) be the first 10 terms of the exp(t) series.
% The following computes E(L(a/10)) for a=1,...,12.
\cnta 0
\loop
\noindent\xintTrunc {18}{%
     \xintRationalSeriesX {0}{9}{\firstterm}{\ratioexp}
         {\xintPowerSeries{1}{10}{\coefflog}{\the\cnta[-1]}}}\dots
\endgraf
\ifnum\cnta < 12 \advance \cnta 1 \repeat
```

1.099999999999083906...	1.499954310225476533...	1.870485649686617459...
1.199999998111624029...	1.599659266069210466...	1.907197560339468199...
1.299999835744121464...	1.698137473697423757...	1.845117565491393752...
1.399996091955359088...	1.791898112718884531...	1.593831932293536053...

These completely exact operations rapidly create numbers with many digits. Let us print in full the raw fractions created by the operation illustrated above:

E(L(1[-1]))=43553495273430499375312847830569575544652599841891642065630853442715414147101380720658820298104601315534223370128916508905683005693656447898877952000000000/39594086612242519324387557078266845776303882240000000000000000000[-90] (length of numerator: 155)

E(L(12[-2]))=4434537700544174654421092523472648247118935991604117296038825841980841532261080707075058900962803059710371332802034641237155887714188380658982959014134632946402759999397422009303463626532643

541704863984316744555312271367954598414044364800000000/395940866122425193243875570782668457763038822400000000000000000000[-180] (length of numerator: 245)

E(L(123[-3]))=444641592651941777154254148848854866198954971552616390074295913531792113850864779762350800814416981762774148663052493217566759754097977420731516373336789722730765496139079185229545102248282391199621029237793811740122110919735433161132757168955864017710881850585395079859843831617966207195391567803471832147436302936555630100480000000000/395940866122425193243875570782668457763038822400000000000000000000000[-270] (length of numerator: 335)

We see that the denominators here remain the same, as our input only had various powers of ten as denominators, and **xintfrac** efficiently assemble (some only, as we can see) powers of ten. Notice that 1 more digit in an input denominator seems to mean 90 more in the raw output. We can check that with some other test cases:

E(L(1/7))=518138516117322604916074833164833344883840590133006168125125346674309133532553948047136691585715900449768925914489452341864351924224000000000000/453371201621089791788096627821377652892232653817581525466548360950870896010226899427964653421154077863588092639042087157760000000000000000000[0] (length of numerator: 141; length of denominator: 141)

E(L(1/71))=164799489177219556498025955806107098256158101756209369864657152282149780083067798039175325186850716609293467854603842163754716919123274624394132188208895310089982001627351524910000588238596565380887916286153347403881434316800000000/162510607383091507102283159265830434485606350979982865517923046004017115844425486049111273926394712850261667426510159483544917475146636033045963798199826115486814955381536472641379276308916890414267771321449447424000000000000000000[0] (length of numerator: 232; length of denominator: 232)

E(L(1/712))=209623173880163120675481637897216200283968902248203238943136902264182865559717266406341976325767001357109452980607391271438079195073959301528254006087908156888129567520269011715459969154688799089625738271433856535377918700884980798641197021855117078629780316835353043067415753497212012899985019017494798220551782400000000000/209329172233767379973271986231161997566292788454774484652603429574146596817758309378641205048095830135707522121389654690301198396108060572490342602456343055829220334691330984419090140201839416227006587667057555033000272129209621768247300082961810343260003611903508489426616664834303221920647163859173376000000000000000000000[0] (length of numerator: 322; length of denominator: 322)

For info the last fraction put into irreducible form still has 288 digits in its denominator.[74] Thus decimal numbers such as `0.123` (equivalently `123[-3]`) give less computing intensive tasks than fractions such as `1/712`: in the case of decimal numbers the (raw) denominators originate in the coefficients of the series themselves, powers of ten of the input within brackets being treated separately. And even then the numerators will grow with the size of the input in a sort of linear way, the coefficient being given by the order of series: here

[74] putting this fraction in irreducible form takes more time than is typical of the other computations in this document; so exceptionally I have hard-coded the 288 in the document source.

10 from the log and 9 from the exp, so 90. One more digit in the input means 90 more digits in the numerator of the output: obviously we can not go on composing such partial sums of series and hope that **xint** will joyfully do all at the speed of light! Briefly said, imagine that the rules of the game make the programmer like a security guard at an airport scanning machine: a never-ending flux of passengers keep on arriving and all you can do is re-shuffle the first nine of them, organize marriages among some, execute some, move children farther back among the first nine only. If a passenger comes along with many hand luggages, this will slow down the process even if you move him to ninth position, because sooner or later you will have to digest him, and the children will be big too. There is no way to move some guy out of the file and to a discrete interrogatory room for separate treatment or to give him/her some badge saying “I left my stuff in storage box 357”.

Hence, truncating the output (or better, rounding) is the only way to go if one needs a general calculus of special functions. This is why the package **xintseries** provides, besides `\xintSeries`, `\xintRationalSeries`, or `\xintPowerSeries` which compute *exact* sums, also has `\xintFxPtPowerSeries` for fixed-point computations.

Update: release `1.08a` of **xintseries** now includes a tentative naive `\xintFloatPowerSeries`.

29.5 `\xintPowerSeries`

$\overset{\text{num}}{x}\ \overset{\text{num}}{x}\ \overset{\text{Frac}}{f}\ \overset{\text{Frac}}{f}\ \star$ `\xintPowerSeries{A}{B}{\coeff}{f}` evaluates the sum $\sum_{n=A}^{n=B}$ `\coeff{n}` $\cdot$ $\mathtt{f}^{\mathtt{n}}$. The initial and final indices are given to a `\numexpr` expression. The `\coeff` macro (which, as argument to `\xintPowerSeries` is expanded only at the time `\coeff{n}` is needed) should be defined as a one-parameter expandable command, its input will be an explicit number.

The `f` can be either a fraction directly input or a macro `\f` expanding to such a fraction. It is actually more efficient to encapsulate an explicit fraction `f` in such a macro, if it has big numerators and denominators (‘big’ means hundreds of digits) as it will then take less space in the processing until being (repeatedly) used.

This macro computes the *exact* result (one can use it also for polynomial evaluation). Starting with release `1.04` a Horner scheme for polynomial evaluation is used, which has the advantage to avoid a denominator build-up which was plaguing the `1.03` version. [75]

> Note: as soon as the coefficients look like factorials, it is more efficient to use the `\xintRationalSeries` macro whose evaluation, also based on a similar Horner scheme, will avoid a denominator build-up originating in the coefficients themselves.

```
\def\geom #1{1[0]} % the geometric series
\def\f {5/17[0]}
\[ \sum_{n=0}^{n=20} \Bigl(\frac 5{17}\Bigr)^n
 =\xintFrac{\xintIrr{\xintPowerSeries {0}{20}{\geom}{\f}}}
 =\xintFrac{\xinttheexpr (17^21-5^21)/12/17^20\relax}\]
```

$$\sum_{n=0}^{n=20}\Bigl(\frac{5}{17}\Bigr)^n=\frac{5757661159377657976885341}{4064231406647572522401601}=\frac{69091933912531895722624092}{48770776879770870268819212}$$

[75] with powers `f^k`, from `k=0` to `N`, a denominator `d` of `f` became `d^{1+2+...+N}`, which is bad. With the `1.04` method, the part of the denominator originating from `f` does not accumulate to more than `d^N`.

```
\def\coefflog #1{1/#1[0]}% 1/n
\def\f {1/2[0]}%
\[ \log 2 \approx \sum_{n=1}^{20} \frac1{n\cdot 2^n}
    = \xintFrac {\xintIrr {\xintPowerSeries {1}{20}{\coefflog}{\f}}}\]
\[ \log 2 \approx \sum_{n=1}^{50} \frac1{n\cdot 2^n}
    = \xintFrac {\xintIrr {\xintPowerSeries {1}{50}{\coefflog}{\f}}}\]
```

$$\log 2 \approx \sum_{n=1}^{20} \frac{1}{n\cdot 2^n} = \frac{42299423848079}{61025172848640}$$

$$\log 2 \approx \sum_{n=1}^{50} \frac{1}{n\cdot 2^n} = \frac{60463469751752265663579884559739219}{87230347965792839223946208178339840}$$

```
\cnta 1 % previously declared count
\loop   % in this loop we recompute from scratch each partial sum!
% we can afford that, as \xintPowerSeries is fast enough.
\noindent\hbox to 2em{\hfil\texttt{\the\cnta.} }%
         \xintTrunc {12}
             {\xintPowerSeries {1}{\cnta}{\coefflog}{\f}}\dots
\endgraf
\ifnum \cnta < 30 \advance\cnta 1 \repeat
```

1. 0.500000000000...	11. 0.693109245355...	21. 0.693147159757...
2. 0.625000000000...	12. 0.693129590407...	22. 0.693147170594...
3. 0.666666666666...	13. 0.693138980431...	23. 0.693147175777...
4. 0.682291666666...	14. 0.693143340085...	24. 0.693147178261...
5. 0.688541666666...	15. 0.693145374590...	25. 0.693147179453...
6. 0.691145833333...	16. 0.693146328265...	26. 0.693147180026...
7. 0.692261904761...	17. 0.693146777052...	27. 0.693147180302...
8. 0.692750186011...	18. 0.693146988980...	28. 0.693147180435...
9. 0.692967199900...	19. 0.693147089367...	29. 0.693147180499...
10. 0.693064856150...	20. 0.693147137051...	30. 0.693147180530...

```
%\def\coeffarctg #1{1/\the\numexpr\xintMON{#1}*(2*#1+1)\relax }%
\def\coeffarctg  #1{1/\the\numexpr\ifodd #1 -2*#1-1\else2*#1+1\fi\relax }%
% the above gives (-1)^n/(2n+1). The sign being in the denominator,
%              **** no [0] should be added ****,
% else nothing is guaranteed to work (even if it could by sheer luck)
% NOTE in passing this aspect of \numexpr:
%         ****  \numexpr -(1)\relax does not work!!! ****
\def\f {1/25[0]}% 1/5^2
\[\mathrm{Arctg}(\frac15)\approx
            \frac15\sum_{n=0}^{15} \frac{(-1)^n}{(2n+1)25^n}
= \xintFrac{\xintIrr {\xintDiv
            {\xintPowerSeries {0}{15}{\coeffarctg}{\f}}{5}}}\]
```

$$\mathrm{Arctg}(\frac{1}{5}) \approx \frac{1}{5}\sum_{n=0}^{15} \frac{(-1)^n}{(2n+1)25^n} = \frac{165918726519122955895391793269168}{840539304153062403202056884765625}$$

29.6 \xintPowerSeriesX

num num Frac Frac
x x f f

This is the same as \xintPowerSeries apart from the fact that the last parameter f is expanded once and for all before being then used repeatedly. If the f parameter is to be an explicit big fraction with many (dozens) digits, rather than using it directly it is slightly better to have some macro \g defined to expand to the explicit fraction and then use \xintPowerSeries with \g; but if f has not yet been evaluated and will be the output of a complicated expansion of some \f, and if, due to an expanding only context, doing \edef \g{\f} is no option, then \xintPowerSeriesX should be used with \f as last parameter.

```
\def\ratioexp #1#2{\xintDiv {#1}{#2}}% x/n
% These are the (-1)^{n-1}/n of the log(1+h) series:
\def\coefflog #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}%
% Let L(h) be the first 10 terms of the log(1+h) series and
% let E(t) be the first 10 terms of the exp(t) series.
% The following computes L(E(a/10)-1) for a=1,..., 12.
\cnta 1
\loop
\noindent\xintTrunc {18}{%
   \xintPowerSeriesX {1}{10}{\coefflog}
  {\xintSub
      {\xintRationalSeries {0}{9}{1[0]}{\ratioexp{\the\cnta[-1]}}}
      {1}}}\dots
\endgraf
\ifnum\cnta < 12 \advance \cnta 1 \repeat
```

0.099999999998556159...	0.499511320760604148...	-1.597091692317639401...
0.199999995263443554...	0.593980619762352217...	-12.648937932093322763...
0.299999338075041781...	0.645144282733914916...	-66.259639046914679687...
0.399974460740121112...	0.398118280111436442...	-304.768437445462801227...

29.7 \xintFxPtPowerSeries

num num ★
x x
Frac Frac num ★
f f x

\xintFxPtPowerSeries{A}{B}{\coeff}{f}{D} computes $\sum_{n=A}^{n=B}$ \coeff{n} $\cdot$ f^{n} with each term of the series truncated to D digits after the decimal point. As usual, A and B are completely expanded through their inclusion in a \numexpr expression. Regarding D it will be similarly be expanded each time it is used inside an \xintTrunc. The one-parameter macro \coeff is similarly expanded at the time it is used inside the computations. Idem for f. If f itself is some complicated macro it is thus better to use the variant \xintFxPtPowerSeriesX which expands it first and then uses the result of that expansion.

The current (1.04) implementation is: the first power f^A is computed exactly, then *truncated*. Then each successive power is obtained from the previous one by multiplication by the exact value of f, and truncated. And \coeff{n}·f^n is obtained from that by multiplying by \coeff{n} (untruncated) and then truncating. Finally the sum is computed exactly. Apart from that \xintFxPtPowerSeries (where FxPt means 'fixed-point') is like \xintPowerSeries.

There should be a variant for things of the type $\sum c_n \frac{f^n}{n!}$ to avoid having to compute the factorial from scratch at each coefficient, the same way \xintFxPtPowerSeries does not compute f^n from scratch at each n. Perhaps in the next package release.

$$e^{-\frac{1}{2}} \approx$$

1.00000000000000000000	0.60653056795634920635	0.60653065971263344622
0.50000000000000000000	0.60653066483754960317	0.60653065971263342289
0.62500000000000000000	0.60653065945526069224	0.60653065971263342361
0.60416666666666666667	0.60653065972437513778	0.60653065971263342359
0.60677083333333333333	0.60653065971214266299	0.60653065971263342359
0.60651041666666666667	0.60653065971265234943	0.60653065971263342359
0.60653211805555555555	0.60653065971263274611	

```
\def\coeffexp #1{1/\xintFac {#1}[0]}% 1/n!
\def\f {-1/2[0]}% [0] for faster input parsing
\cnta 0 % previously declared \count register
\noindent\loop
$\xintFxPtPowerSeries {0}{\cnta}{\coeffexp}{\f}{20}$\\
\ifnum\cnta<19 \advance\cnta 1 \repeat\par
% One should **not** trust the final digits, as the potential truncation
% errors of up to 10^{-20} per term accumulate and never disappear! (the
% effect is attenuated by the alternating signs in the series). We  can
% confirm that the last two digits (of our evaluation of the nineteenth
% partial sum) are wrong via the evaluation with more digits:
\xintFxPtPowerSeries {0}{19}{\coeffexp}{\f}{25}= 0.6065306597126334236037992
```

It is no difficulty for **xintfrac** to compute exactly, with the help of `\xintPowerSeries`, the nineteenth partial sum, and to then give (the start of) its exact decimal expansion:

$$\texttt{\xintPowerSeries \{0\}\{19\}\{\coeffexp\}\{\f\}} = \frac{38682746160036397317757}{63777066403145711616000} = 0.606530659712633423603799152126\ldots$$

Thus, one should always estimate a priori how many ending digits are not reliable: if there are `N` terms and `N` has `k` digits, then digits up to but excluding the last `k` may usually be trusted. If we are optimistic and the series is alternating we may even replace `N` with $\sqrt{\texttt{N}}$ to get the number `k` of digits possibly of dubious significance.

29.8 `\xintFxPtPowerSeriesX`

num x num x Frac f Frac f num x ★

`\xintFxPtPowerSeriesX{A}{B}{\coeff}{\f}{D}` computes, exactly as `\xintFxPtPowerSeries`, the sum of `\coeff{n}·\f^n` from `n=A` to `n=B` with each term of the series being *truncated* to `D` digits after the decimal point. The sole difference is that `\f` is first expanded and it is the result of this which is used in the computations.

Let us illustrate this on the numerical exploration of the identity

`log(1+x) = -log(1/(1+x))`

Let `L(h)=log(1+h)`, and `D(h)=L(h)+L(-h/(1+h))`. Theoretically thus, `D(h)=0` but we shall evaluate `L(h)` and `-h/(1+h)` keeping only 10 terms of their respective series. We will assume `|h|<0.5`. With only ten terms kept in the power series we do not have quite 3 digits precision as `2^10=1024`. So it wouldn't make sense to evaluate things more precisely than, say circa 5 digits after the decimal points.

```
\cnta 0
\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}% (-1)^{n-
1}/n
\def\coeffalt #1{\the\numexpr\ifodd#1 -1\else1\fi\relax [0]}%   (-1)^n
\loop
```

```
\noindent \hbox to 2.5cm {\hss\texttt{D(\the\cnta/100): }}%
\xintAdd {\xintFxPtPowerSeriesX {1}{10}{\coefflog}{\the\cnta [-2]}{5}}
         {\xintFxPtPowerSeriesX {1}{10}{\coefflog}
             {\xintFxPtPowerSeriesX {1}{10}{\coeffalt}{\the\cnta [-2]}{5}}
          {5}}\endgraf
\ifnum\cnta < 49 \advance\cnta 7 \repeat
```

```
   D(0/100): 0/1[0]                  D(28/100): 4/1[-5]
   D(7/100): 2/1[-5]                 D(35/100): 4/1[-5]
  D(14/100): 2/1[-5]                 D(42/100): 9/1[-5]
  D(21/100): 3/1[-5]                 D(49/100): 42/1[-5]
```

Let's say we evaluate functions on `[-1/2,+1/2]` with values more or less also in `[-1/2,+1/2]` and we want to keep 4 digits of precision. So, roughly we need at least 14 terms in series like the geometric or log series. Let's make this 15. Then it doesn't make sense to compute intermediate summands with more than 6 digits precision. So we compute with 6 digits precision but return only 4 digits (rounded) after the decimal point. This result with 4 post-decimal points precision is then used as input to the next evaluation.

```
\loop
\noindent \hbox to 2.5cm {\hss\texttt{D(\the\cnta/100): }}%
\xintRound{4}
 {\xintAdd {\xintFxPtPowerSeriesX {1}{15}{\coefflog}{\the\cnta [-2]}{6}}
           {\xintFxPtPowerSeriesX {1}{15}{\coefflog}
                  {\xintRound {4}{\xintFxPtPowerSeriesX {1}{15}{\coeffalt}
                                 {\the\cnta [-2]}{6}}}
            {6}}%
 }\endgraf
\ifnum\cnta < 49 \advance\cnta 7 \repeat
```

```
   D(0/100): 0                       D(28/100): -0.0001
   D(7/100): 0.0000                  D(35/100): -0.0001
  D(14/100): 0.0000                  D(42/100): -0.0000
  D(21/100): -0.0001                 D(49/100): -0.0001
```

Not bad... I have cheated a bit: the 'four-digits precise' numeric evaluations were left unrounded in the final addition. However the inner rounding to four digits worked fine and made the next step faster than it would have been with longer inputs. The morale is that one should not use the raw results of `\xintFxPtPowerSeriesX` with the `D` digits with which it was computed, as the last are to be considered garbage. Rather, one should keep from the output only some smaller number of digits. This will make further computations faster and not less precise. I guess there should be some command to do this final truncating, or better, rounding, at a given number `D'<D` of digits. Maybe for the next release.

29.9 \xintFloatPowerSeries

[num x] num x num x Frac Frac f f ★

`\xintFloatPowerSeries[P]{A}{B}{\coeff}{f}` computes $\sum_{n=A}^{n=B}$ `\coeff{n}` $\cdot$ f^n with a floating point precision given by the optional parameter `P` or by the current setting of `\xintDigits`.

In the current, preliminary, version, no attempt has been made to try to guarantee to the final result the precision `P`. Rather, `P` is used for all intermediate floating point evaluations. So rounding errors will make some of the last printed digits invalid. The operations done are first the evaluation of `f^A` using `\xintFloatPow`, then each successive power is obtained from this first one by multiplication by `f` using `\xintFloatMul`, then again with `\xintFloatMul` this is multiplied with `\coeff{n}`, and the sum is done adding one term at a time with `\xintFloatAdd`. To sum up, this is just the naive transformation of `\xintFxPtPowerSeries` from fixed point to floating point.

```
\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}%
\xintFloatPowerSeries [8]{1}{30}{\coefflog}{-1/2[0]}
```

-6.9314718e-1

29.10 `\xintFloatPowerSeriesX`

$[^{\text{num}}_{x}]$ $^{\text{num}}_{x}$ $^{\text{num}}_{x}$ Frac Frac f f ★

`\xintFloatPowerSeriesX[P]{A}{B}{\coeff}{f}` is like `\xintFloatPowerSeries` with the difference that `f` is expanded once and for all at the start of the computation, thus allowing efficient chaining of such series evaluations.

```
\def\coeffexp #1{1/\xintFac {#1}[0]}% 1/n! (exact, not float)
\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}%
\xintFloatPowerSeriesX [8]{0}{30}{\coeffexp}
    {\xintFloatPowerSeries [8]{1}{30}{\coefflog}{-1/2[0]}}
```

5.0000001e-1

29.11 Computing $\log 2$ and π

In this final section, the use of `\xintFxPtPowerSeries` (and `\xintPowerSeries`) will be illustrated on the (expandable... why make things simple when it is so easy to make them difficult!) computations of the first digits of the decimal expansion of the familiar constants $\log 2$ and π.

Let us start with $\log 2$. We will get it from this formula (which is left as an exercise):

`log(2)=-2 log(1-13/256)-5 log(1-1/9)`

The number of terms to be kept in the log series, for a desired precision of `10^{-D}` was roughly estimated without much theoretical analysis. Computing exactly the partial sums with `\xintPowerSeries` and then printing the truncated values, from `D=0` up to `D=100` showed that it worked in terms of quality of the approximation. Because of possible strings of zeroes or nines in the exact decimal expansion (in the present case of $\log 2$, strings of zeroes around the fourtieth and the sixtieth decimals), this does not mean though that all digits printed were always exact. In the end one always end up having to compute at some higher level of desired precision to validate the earlier result.

Then we tried with `\xintFxPtPowerSeries`: this is worthwile only for `D`'s at least 50, as the exact evaluations are faster (with these short-length `f`'s) for a lower number of digits. And as expected the degradation in the quality of approximation was in this range of the order of two or three digits. This meant roughly that the 3+1=4 ending digits were wrong. Again, we ended up having to compute with five more digits and compare with the earlier value to validate it. We use truncation rather than rounding because our goal is not to obtain the correct rounded decimal expansion but the correct exact truncated one.

```
\def\coefflog #1{1/#1[0]}% 1/n
\def\xa {13/256[0]}%  we will compute log(1-13/256)
\def\xb {1/9[0]}%     we will compute log(1-1/9)
\def\LogTwo #1%
%  get log(2)=-2log(1-13/256)- 5log(1-1/9)
```

```
{% we want to use \printnumber, hence need something expanding in two steps
 % only, so we use here the \romannumeral0 method
  \romannumeral0\expandafter\LogTwoDoIt \expandafter
    % Nb Terms for 1/9:
  {\the\numexpr #1*150/143\expandafter}\expandafter
    % Nb Terms for 13/256:
  {\the\numexpr #1*100/129\expandafter}\expandafter
    % We print #1 digits, but we know the ending ones are garbage
  {\the\numexpr #1\relax}% allows #1 to be a count register
}%
\def\LogTwoDoIt #1#2#3%
%  #1=nb of terms for 1/9, #2=nb of terms for 13/256,
{% #3=nb of digits for computations, also used for printing
 \xinttrunc {#3} % lowercase form to stop the \romannumeral0 expansion!
 {\xintAdd
  {\xintMul {2}{\xintFxPtPowerSeries {1}{#2}{\coefflog}{\xa}{#3}}}
  {\xintMul {5}{\xintFxPtPowerSeries {1}{#1}{\coefflog}{\xb}{#3}}}%
 }%
}%
\noindent $\log 2 \approx \LogTwo {60}\dots$\endgraf
\noindent\phantom{$\log 2$}${}\approx{}$\printnumber{\LogTwo {65}}\dots\endgraf
\noindent\phantom{$\log 2$}${}\approx{}$\printnumber{\LogTwo {70}}\dots\endgraf
```

$\log 2 \approx$ 0.693147180559945309417232121458176568075500134360255254120484...
$\approx$ 0.693147180559945309417232121458176568075500134360255254120680
00711...
$\approx$ 0.693147180559945309417232121458176568075500134360255254120680
0094933723...

Here is the code doing an exact evaluation of the partial sums. We have added a +1 to the number of digits for estimating the number of terms to keep from the log series: we experimented that this gets exactly the first `D` digits, for all values from `D=0` to `D=100`, except in one case (`D=40`) where the last digit is wrong. For values of `D` higher than `100` it is more efficient to use the code using `\xintFxPtPowerSeries`.

```
\def\LogTwo #1% get log(2)=-2log(1-13/256)- 5log(1-1/9)
{%
    \romannumeral0\expandafter\LogTwoDoIt \expandafter
    {\the\numexpr (#1+1)*150/143\expandafter}\expandafter
    {\the\numexpr (#1+1)*100/129\expandafter}\expandafter
    {\the\numexpr #1\relax}%
}%
\def\LogTwoDoIt #1#2#3%
{%   #3=nb of digits for truncating an EXACT partial sum
  \xinttrunc {#3}
    {\xintAdd
      {\xintMul {2}{\xintPowerSeries {1}{#2}{\coefflog}{\xa}}}
      {\xintMul {5}{\xintPowerSeries {1}{#1}{\coefflog}{\xb}}}%
    }%
}%
```

Let us turn now to Pi, computed with the Machin formula. Again the numbers of terms to keep in the two `arctg` series were roughly estimated, and some experimentations showed that removing the last three digits was enough (at least for D=0-100 range). And the algorithm does print the correct digits when used with D=1000 (to be convinced of that one needs to run it for D=1000 and again, say for D=1010.) A theoretical analysis could help confirm that this algorithm always gets better than 10^{-D} precision, but again, strings of zeroes or nines encountered in the decimal expansion may falsify the ending digits, nines may be zeroes (and the last non-nine one should be increased) and zeroes may be nine (and the last non-zero one should be decreased).

```
% pi = 16 Arctg(1/5) - 4 Arctg(1/239) (John Machin's formula)
\def\coeffarctg #1{\the\numexpr\ifodd#1 -1\else1\fi\relax/%
                                       \the\numexpr 2*#1+1\relax [0]}%
% the above computes (-1)^n/(2n+1).
\def\xa {1/25[0]}%      1/5^2, the [0] for (infinitesimally) faster pars-
ing
\def\xb {1/57121[0]}% 1/239^2, the [0] for faster parsing
\def\Machin #1{% \Machin {\mycount} is allowed
    \romannumeral0\expandafter\MachinA \expandafter
     % number of terms for arctg(1/5):
    {\the\numexpr (#1+3)*5/7\expandafter}\expandafter
     % number of terms for arctg(1/239):
    {\the\numexpr (#1+3)*10/45\expandafter}\expandafter
     % do the computations with 3 additional digits:
    {\the\numexpr #1+3\expandafter}\expandafter
     % allow #1 to be a count register:
    {\the\numexpr #1\relax }}%
\def\MachinA #1#2#3#4%
% #4: digits to keep after decimal point for final printing
% #3=#4+3: digits for evaluation of the necessary number of terms
% to be kept in the arctangent series, also used to truncate each
% individual summand.
{\xinttrunc {#4} % lowercase macro to match the initial \romannumeral0.
 {\xintSub
  {\xintMul {16/5}{\xintFxPtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}}
  {\xintMul {4/239}{\xintFxPtPowerSeries {0}{#2}{\coeffarctg}{\xb}{#3}}}%
 }}%
\[ \pi = \Machin {60}\dots \]
```

$$\pi = 3.141592653589793238462643383279502884197169399375105820974944\dots$$

Here is a variant\MachinBis, which evaluates the partial sums *exactly* using \xintPowerSeries, before their final truncation. No need for a "+3" then.

```
\def\MachinBis #1{% #1 may be a count register,
% the final result will be truncated to #1 digits post decimal point
    \romannumeral0\expandafter\MachinBisA \expandafter
     % number of terms for arctg(1/5):
    {\the\numexpr #1*5/7\expandafter}\expandafter
     % number of terms for arctg(1/239):
    {\the\numexpr #1*10/45\expandafter}\expandafter
```

```
      % allow #1 to be a count register:
    {\the\numexpr #1\relax }}%
\def\MachinBisA #1#2#3%
{\xinttrunc {#3} %
 {\xintSub
   {\xintMul {16/5}{\xintPowerSeries {0}{#1}{\coeffarctg}{\xa}}}
   {\xintMul{4/239}{\xintPowerSeries {0}{#2}{\coeffarctg}{\xb}}}%
}}%
```

Let us use this variant for a loop showing the build-up of digits:

```
    \cnta 0 % previously declared \count register
    \loop
    \MachinBis{\cnta} \endgraf  % Plain's \loop does not accept \par
    \ifnum\cnta < 30 \advance\cnta 1 \repeat
```

```
3.
3.1
3.14
3.141
3.1415
3.14159
3.141592
3.1415926
3.14159265
3.141592653
3.1415926535
3.14159265358
3.141592653589
3.1415926535897
3.14159265358979
3.141592653589793
3.1415926535897932
3.14159265358979323
3.141592653589793238
3.1415926535897932384
3.14159265358979323846
3.141592653589793238462
3.1415926535897932384626
3.14159265358979323846264
3.141592653589793238462643
3.1415926535897932384626433
3.14159265358979323846264338
3.141592653589793238462643383
3.1415926535897932384626433832
3.14159265358979323846264338327
3.141592653589793238462643383279
```

You want more digits and have some time? compile this copy of the `\Machin` with `etex` (or `pdftex`):

```
% Compile with e-TeX extensions enabled (etex, pdftex, ...)
\input xintfrac.sty
\input xintseries.sty
% pi = 16 Arctg(1/5) - 4 Arctg(1/239) (John Machin's formula)
\def\coeffarctg #1{\the\numexpr\ifodd#1 -1\else1\fi\relax/%
                                  \the\numexpr 2*#1+1\relax [0]}%
\def\xa {1/25[0]}%
\def\xb {1/57121[0]}%
\def\Machin #1{%
    \romannumeral0\expandafter\MachinA \expandafter
    {\the\numexpr (#1+3)*5/7\expandafter}\expandafter
    {\the\numexpr (#1+3)*10/45\expandafter}\expandafter
    {\the\numexpr #1+3\expandafter}\expandafter
    {\the\numexpr #1\relax }}%
\def\MachinA #1#2#3#4%
{\xinttrunc {#4}
 {\xintSub
  {\xintMul {16/5}{\xintFxPtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}}
```

```
  {\xintMul {4/239}{\xintFxPtPowerSeries {0}{#2}{\coeffarctg}{\xb}{#3}}}%
}}%
\pdfresettimer
\oodef\Z {\Machin {1000}}
\odef\W {\the\pdfelapsedtime}
\message{\Z}
\message{computed in \xintRound {2}{\W/65536} seconds.}
\bye
```

This will log the first 1000 digits of π after the decimal point. On my laptop (a 2012 model) this took about 16 seconds last time I tried. [76] As mentioned in the introduction, the file pi.tex by D. Roegel shows that orders of magnitude faster computations are possible within TeX, but recall our constraints of complete expandability and be merciful, please.

Why truncating rather than rounding? One of our main competitors on the market of scientific computing, a canadian product (not encumbered with expandability constraints, and having barely ever heard of TeX ;-), prints numbers rounded in the last digit. Why didn't we follow suit in the macros `\xintFxPtPowerSeries` and `\xintFxPtPowerSeriesX`? To round at `D` digits, and excluding a rewrite or cloning of the division algorithm which anyhow would add to it some overhead in its final steps, **xintfrac** needs to truncate at `D+1`, then round. And rounding loses information! So, with more time spent, we obtain a worst result than the one truncated at `D+1` (one could imagine that additions and so on, done with only `D` digits, cost less; true, but this is a negligeable effect per summand compared to the additional cost for this term of having been truncated at `D+1` then rounded). Rounding is the way to go when setting up algorithms to evaluate functions destined to be composed one after the other: exact algebraic operations with many summands and an `f` variable which is a fraction are costly and create an even bigger fraction; replacing `f` with a reasonable rounding, and rounding the result, is necessary to allow arbitrary chaining.

But, for the computation of a single constant, we are really interested in the exact decimal expansion, so we truncate and compute more terms until the earlier result gets validated. Finally if we do want the rounding we can always do it on a value computed with `D+1` truncation.

30 Commands of the xintcfrac package

This package was first included in release `1.04` of the **xint** bundle.

Contents

[76] With `1.09i` and earlier **xint** releases, this used to be `42` seconds; the `1.09j` division is much faster with small denominators as occurs here with `\xa=1/25`, and I believe this to be the main explanation for the speed gain.

30.1 Package overview

A *simple* continued fraction has coefficients `[c0,c1,...,cN]` (usually called partial quotients, but I really dislike this entrenched terminology), where `c0` is a positive or negative integer and the others are positive integers. As we will see it is possible with **xintcfrac** to specify the coefficient function `c:n->cn`. Note that the index then starts at zero as indicated. With the `amsmath` macro `\cfrac` one can display such a continued fraction as

$$c_0 + \cfrac{1}{c_1 + \cfrac{1}{c_2 + \cfrac{1}{c_3 + \cfrac{1}{\ddots}}}}$$

Here is a concrete example:

$$\frac{208341}{66317} = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{292 + \cfrac{1}{2}}}}}$$

But the difference with `amsmath`'s `\cfrac` is that this was input as

```
\[ \xintFrac {208341/66317}=\xintCFrac {208341/66317} \]
```

The command `\xintCFrac` produces in two expansion steps the whole thing with the many chained `\cfrac`'s and all necessary braces, ready to be printed, in math mode. This is LaTeX only and with the `amsmath` package (we shall mention another method for Plain TeX users of `amstex`).

A *generalized* continued fraction has the same structure but the numerators are not restricted to be ones, and numbers used in the continued fraction may be arbitrary, also fractions, irrationals, indeterminates. The *centered* continued fraction associated to a rational number is an example:

$$\frac{915286}{188421} = 5 - \cfrac{1}{7 + \cfrac{1}{39 - \cfrac{1}{53 - \cfrac{1}{13}}}} = 4 + \cfrac{1}{1 + \cfrac{1}{6 + \cfrac{1}{38 + \cfrac{1}{1 + \cfrac{1}{51 + \cfrac{1}{1 + \cfrac{1}{12}}}}}}}$$

`\[ \xintFrac {915286/188421}=\xintGCFrac {\xintFtoCC {915286/188421}} \]`
The command `\xintGCFrac`, contrarily to `\xintCFrac`, does not compute anything, it just typesets. Here, it is the command `\xintFtoCC` which did the computation of the centered continued fraction of `f`. Its output has the 'inline format' described in the next paragraph. In the display, we also used `\xintCFrac` (code not shown), for comparison of the two types of continued fractions.

A generalized continued fraction may be input 'inline' as:

```
a0+b0/a1+b1/a2+b2/...../a(n-1)+b(n-1)/an
```

Fractions among the coefficients are allowed but they must be enclosed within braces. Signed integers may be left without braces (but the + signs are mandatory). Or, they may be macros expanding (in two steps) to some number or fractional number.

```
\xintGCFrac {1+-1/57+\xintPow {-3}{7}/\xintQuo {132}{25}}
```

$$\frac{1907}{1902} = 1 - \cfrac{1}{57 - \cfrac{2187}{5}}$$

The left hand side was obtained with the following code:

```
\xintFrac{\xintGCtoF {1+-1/57+\xintPow {-3}{7}/\xintQuo {132}{25}}}
```

It uses the macro `\xintGCtoF` to convert a generalized fraction from the 'inline format' to the fraction it evaluates to.

A simple continued fraction is a special case of a generalized continued fraction and may be input as such to macros expecting the 'inline format', for example `-7+1/6+1/19+1/1+1/33`. There is a simpler comma separated format:

```
\xintFrac{\xintCstoF{-7,6,19,1,33}}=\xintCFrac{\xintCstoF{-7,6,19,1,33}}
```

$$\frac{-28077}{4108} = -7 + \cfrac{1}{6 + \cfrac{1}{19 + \cfrac{1}{1 + \cfrac{1}{33}}}}$$

This comma separated format may also be used with fractions among the coefficients: in that case, computing with `\xintFtoCs` from the resulting `f` its real coefficients will give a new comma separated list with only integers. This list has no spaces: the spaces in the display below arise from the math mode processing.

```
\xintFrac{1084483/398959}=[\xintFtoCs{1084483/398959}]
```

$$\frac{1084483}{398959} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 2]$$

If one prefers other separators, one can use \xintFtoCx whose first argument will be the separator to be used.

```
\xintFrac{2721/1001}=\xintFtoCx {+1/(}{2721/1001})\cdots)
```

$$\frac{2721}{1001} = 2 + 1/(1 + 1/(2 + 1/(1 + 1/(1 + 1/(4 + 1/(1 + 1/(1 + 1/(6 + 1/(2)\cdots)$$

People using Plain TeX and `amstex` can achieve the same effect as `\xintCFrac` with: `$$ \xintFwOver{2721/1001}=\xintFtoCx {+\cfrac1\\ }{2721/1001}\endcfrac$$`

Using `\xintFtoCx` with first argument an empty pair of braces `{}` will return the list of the coefficients of the continued fraction of `f`, without separator, and each one enclosed in a pair of group braces. This can then be manipulated by the non-expandable macro \xintAssignArray or the expandable ones \xintApply and \xintListWithSep.

As a shortcut to using `\xintFtoCx` with separator `1+/`, there is \xintFtoGC:

```
2721/1001=\xintFtoGC {2721/1001}
2721/1001=2+1/1+1/2+1/1+1/1+1/4+1/1+1/1+1/6+1/2
```

Let us compare in that case with the output of \xintFtoCC:

```
2721/1001=\xintFtoCC {2721/1001}
2721/1001=3+-1/4+-1/2+1/5+-1/2+1/7+-1/2
```

The '`\printnumber`' macro which we use to print long numbers can also be useful on long continued fractions.

```
\printnumber{\xintFtoCC {35037018906350720204351049/%
                         244241737886197404558180}}
```

`143+1/2+1/5+-1/4+-1/4+-1/4+-1/3+1/2+1/2+1/6+-1/22+1/2+1/10+-1/5+-1/11+-1/3+1/4+-1/2+1/2+1/4+-1/2+1/23+1/3+1/8+-1/6+-1/9`. If we apply \xintGCtoF to this generalized continued fraction, we discover that the original fraction was reducible:

```
\xintGCtoF {143+1/2+...+-1/9}=2897319801297630107/20197107104701740
```

When a generalized continued fraction is built with integers, and numerators are only 1's or `-1`'s, the produced fraction is irreducible. And if we compute it again with the last sub-fraction omitted we get another irreducible fraction related to the bigger one by a Bezout identity. Doing this here we get:

```
\xintGCtoF {143+1/2+...+-1/6}=328124887710626729/2287346221788023
```

and indeed:

$$\begin{vmatrix} 2897319801297630107 & 328124887710626729 \\ 20197107104701740 & 2287346221788023 \end{vmatrix} = 1$$

More generally the various fractions obtained from the truncation of a continued fraction to its initial terms are called the convergents. The commands of **xintcfrac** such as \xintFtoCv, \xintFtoCCv, and others which compute such convergents, return them as a list of braced items, with no separator. This list can then be treated either with `\xintAssignArray`, or `\xintListWithSep`, or any other way (but then, some TeX programming knowledge will be necessary). Here is an example:

```
$$\xintFrac{915286/188421}\to \xintListWithSep {,}%
{\xintApply{\xintFrac}{\xintFtoCv{915286/188421}}}$$
```

$$\frac{915286}{188421}\to 4, 5, \frac{34}{7}, \frac{1297}{267}, \frac{1331}{274}, \frac{69178}{14241}, \frac{70509}{14515}, \frac{915286}{188421}$$

```
$$\xintFrac{915286/188421}\to \xintListWithSep {,}%
{\xintApply{\xintFrac}{\xintFtoCCv{915286/188421}}}$$
```

$$\frac{915286}{188421}\to 5, \frac{34}{7}, \frac{1331}{274}, \frac{70509}{14515}, \frac{915286}{188421}$$

We thus see that the 'centered convergents' obtained with `\xintFtoCCv` are among the fuller list of convergents as returned by `\xintFtoCv`.

Here is a more complicated use of `\xintApply` and `\xintListWithSep`. We first define a macro which will be applied to each convergent:

```
\newcommand{\mymacro}[1]{$\xintFrac{#1}=[\xintFtoCs{#1}]$\vtop to 6pt{}}
```

Next, we use the following code:

```
$\xintFrac{49171/18089}\to{}$
\xintListWithSep {, }{\xintApply{\mymacro}{\xintFtoCv{49171/18089}}}
```

It produces:
$\frac{49171}{18089}\to 2 = [2]$, $3 = [3]$, $\frac{8}{3} = [2, 1, 2]$, $\frac{11}{4} = [2, 1, 3]$, $\frac{19}{7} = [2, 1, 2, 2]$, $\frac{87}{32} = [2, 1, 2, 1, 1, 4]$, $\frac{106}{39} = [2, 1, 2, 1, 1, 5]$, $\frac{193}{71} = [2, 1, 2, 1, 1, 4, 2]$, $\frac{1264}{465} = [2, 1, 2, 1, 1, 4, 1, 1, 6]$, $\frac{1457}{536} = [2, 1, 2, 1, 1, 4, 1, 1, 7]$, $\frac{2721}{1001} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 2]$, $\frac{23225}{8544} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8]$, $\frac{49171}{18089} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 2]$.

The macro `\xintCntoF` allows to specify the coefficients as functions of the index. The values to which expand the coefficient function do not have to be integers.

```
\def\cn #1{\xintiPow {2}{#1}}% 2^n
\[\xintFrac{\xintCntoF {6}{\cn}}=\xintCFrac [l]{\xintCntoF {6}{\cn}}\]
```

$$\frac{3541373}{2449193} = 1 + \cfrac{1}{2 + \cfrac{1}{4 + \cfrac{1}{8 + \cfrac{1}{16 + \cfrac{1}{32 + \cfrac{1}{64}}}}}}$$

Notice the use of the optional argument `[l]` to `\xintCFrac`. Other possibilities are `[r]` and (default) `[c]`.

```
\def\cn #1{\xintPow {2}{-#1}}% 1/2^n
\[\xintFrac{\xintCntoF {6}{\cn}} = \xintGCFrac [r]{\xintCntoGC {6}{\cn}}
```

```
= [\xintFtoCs {\xintCntoF {6}{\cn}}]\]
```

$$\frac{3159019}{2465449} = 1 + \cfrac{1}{\frac{1}{2} + \cfrac{1}{\frac{1}{4} + \cfrac{1}{\frac{1}{8} + \cfrac{1}{\frac{1}{16} + \cfrac{1}{\frac{1}{32} + \cfrac{1}{\frac{1}{64}}}}}}} = [1, 3, 1, 1, 4, 14, 1, 1, 1, 1, 79, 2, 1, 1, 2]$$

We used `\xintCntoGC` as we wanted to display also the continued fraction and not only the fraction returned by `\xintCntoF`.

There are also `\xintGCntoF` and `\xintGCntoGC` which allow the same for generalized fractions. The following initial portion of a generalized continued fraction for π:

$$\frac{92736}{29520} = \cfrac{4}{1 + \cfrac{1}{3 + \cfrac{4}{5 + \cfrac{9}{7 + \cfrac{16}{9 + \cfrac{25}{11}}}}}} = 3.1414634146\dots$$

was obtained with this code:

```
\def\an #1{\the\numexpr 2*#1+1\relax }%
\def\bn #1{\the\numexpr (#1+1)*(#1+1)\relax }%
\[ \xintFrac{\xintDiv {4}{\xintGCntoF {5}{\an}{\bn}}} =
        \cfrac{4}{\xintGCFrac{\xintGCntoGC {5}{\an}{\bn}}} =
\xintTrunc {10}{\xintDiv {4}{\xintGCntoF {5}{\an}{\bn}}}\dots\]
```

We see that the quality of approximation is not fantastic compared to the simple continued fraction of π with about as many terms:

```
\[ \xintFrac{\xintCstoF{3,7,15,1,292,1,1}}=
   \xintGCFrac{3+1/7+1/15+1/1+1/292+1/1+1/1}=
   \xintTrunc{10}{\xintCstoF{3,7,15,1,292,1,1}}\dots\]
```

$$\frac{208341}{66317} = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{292 + \cfrac{1}{1 + \cfrac{1}{1}}}}}} = 3.1415926534\dots$$

To conclude this overview of most of the package functionalities, let us explore the convergents of Euler's number e.

```
\def\cn #1{\the\numexpr\ifcase \numexpr #1+3-3*((#1+2)/3)\relax
                           1\or1\or2*(#1/3)\fi\relax }
% produces the pattern 1,1,2,1,1,4,1,1,6,1,1,8,... which are the
% coefficients of the simple continued fraction of e-1.
\cnta 0
\def\mymacro #1{\advance\cnta by 1
                \noindent
                \hbox to 3em {\hfil\small\texttt{\the\cnta.} }%
                $\xintTrunc {30}{\xintAdd {1[0]}{#1}}\dots=
                 \xintFrac{\xintAdd {1[0]}{#1}}$}%
\xintListWithSep{\vtop to 6pt{}\vbox to 12pt{}\par}
    {\xintApply\mymacro{\xintiCstoCv{\xintCntoCs {35}{\cn}}}}
```

The volume of computation is kept minimal by the following steps:

- a comma separated list of the first 36 coefficients is produced by `\xintCntoCs`,
- this is then given to `\xintiCstoCv` which produces the list of the convergents (there is also `\xintCstoCv`, but our coefficients being integers we used the infinitesimally faster `\xintiCstoCv`),
- then the whole list was converted into a sequence of one-line paragraphs, each convergent becomes the argument to a macro printing it together with its decimal expansion with 30 digits after the decimal point.
- A count register `\cnta` was used to give a line count serving as a visual aid: we could also have done that in an expandable way, but well, let's relax from time to time...

1. $2.000000000000000000000000000000\cdots = 2$

2. $3.000000000000000000000000000000\cdots = 3$

3. $2.666666666666666666666666666666\cdots = \frac{8}{3}$

4. $2.750000000000000000000000000000\cdots = \frac{11}{4}$

5. $2.714285714285714285714285714285\cdots = \frac{19}{7}$

6. $2.718750000000000000000000000000\cdots = \frac{87}{32}$

7. $2.717948717948717948717948717948\cdots = \frac{106}{39}$

8. $2.718309859154929577464788732394\cdots = \frac{193}{71}$

9. $2.718279569892473118279569892473\cdots = \frac{1264}{465}$

10. $2.718283582089552238805970149253\cdots = \frac{1457}{536}$

11. $2.718281718281718281718281718281\cdots = \frac{2721}{1001}$

12. $2.718281835205992509363295880149\cdots = \frac{23225}{8544}$

13. $2.718281822943949711891042430591\cdots = \frac{25946}{9545}$

14. $2.718281828735695726684725523798\cdots = \frac{49171}{18089}$

15. $2.718281828445401318035025074172\cdots = \frac{517656}{190435}$

16. $2.718281828470583721777828930962\cdots = \frac{566827}{208524}$

17. $2.71828182845856341127785060620 2\cdots = \frac{1084483}{398959}$
18. $2.71828182845906511407452954664 8\cdots = \frac{13580623}{4996032}$
19. $2.71828182845902801320706559102 6\cdots = \frac{14665106}{5394991}$
20. $2.71828182845904585140462108494 9\cdots = \frac{28245729}{10391023}$
21. $2.71828182845904521352198375822 1\cdots = \frac{410105312}{150869313}$
22. $2.71828182845904525462479502709 2\cdots = \frac{438351041}{161260336}$
23. $2.71828182845904523475756063147 9\cdots = \frac{848456353}{312129649}$
24. $2.71828182845904523537901337277 2\cdots = \frac{14013652689}{5155334720}$
25. $2.71828182845904523534353553278 7\cdots = \frac{14862109042}{5467464369}$
26. $2.71828182845904523536075323018 8\cdots = \frac{28875761731}{10622799089}$
27. $2.71828182845904523536027459394 1\cdots = \frac{534625820200}{196677847971}$
28. $2.71828182845904523536029912091 1\cdots = \frac{563501581931}{207300647060}$
29. $2.71828182845904523536028717990 0\cdots = \frac{1098127402131}{403978495031}$
30. $2.71828182845904523536028747861 1\cdots = \frac{22526049624551}{8286870547680}$
31. $2.71828182845904523536028746472 6\cdots = \frac{23624177026682}{8690849042711}$
32. $2.71828182845904523536028747150 3\cdots = \frac{46150226651233}{16977719590391}$
33. $2.71828182845904523536028747134 9\cdots = \frac{1038929163353808}{382200680031313}$
34. $2.71828182845904523536028747135 5\cdots = \frac{1085079390005041}{399178399621704}$
35. $2.71828182845904523536028747135 2\cdots = \frac{2124008553358849}{781379079653017}$
36. $2.71828182845904523536028747135 2\cdots = \frac{52061284670617417}{19152276311294112}$

The actual computation of the list of all 36 convergents accounts for only 8% of the total time (total time equal to about 5 hundredths of a second in my testing, on my laptop): another 80% is occupied with the computation of the truncated decimal expansions (and the addition of 1 to everything as the formula gives the continued fraction of $e-1$). One can with no problem compute much bigger convergents. Let's get the 200th convergent. It turns out to have the same first 268 digits after the decimal point as $e-1$. Higher convergents get more and more digits in proportion to their index: the 500th convergent already gets 799 digits correct! To allow speedy compilation of the source of this document when the need arises, I limit here to the 200th convergent (getting the 500th took about 1.2s on my laptop last time I tried, and the 200th convergent is obtained ten times faster).

```
\oodef\z {\xintCntoF {199}{\cn}}%
\begingroup\parindent 0pt \leftskip 2.5cm
\indent\llap {Numerator = }{\printnumber{\xintNumerator\z}\par
\indent\llap {Denominator = }\printnumber{\xintDenominator\z}\par
\indent\llap {Expansion = }\printnumber{\xintTrunc{268}\z}\dots
\par\endgroup
```

Numerator = 56896403887189626759752389231580787529388901766791744605
72320245471922969611182301752438601749953108177313670124
1708609749634329382906

Denominator = 33112381766973761930625636081635675336546882372931443815
620561546324665972858186546133769206314891601955061457059255337661142645217223

Expansion = 1.7182818284590452353602874713526624977572470936999595749669676277240766303535475945713821785251664274274663919320030599218174135966290435729003342952605956307381323286279434907632338298807531952510190115738341879307021540891499348841675092447614606680822648001684774118...

One can also use a centered continued fraction: we get more digits but there are also more computations as the numerators may be either 1 or −1.

30.2 \xintCFrac

Frac *f* `\xintCFrac{f}` is a math-mode only, LaTeX with `amsmath` only, macro which first computes then displays with the help of `\cfrac` the simple continued fraction corresponding to the given fraction. It admits an optional argument which may be `[l]`, `[r]` or (the default) `[c]` to specify the location of the one's in the numerators of the sub-fractions. Each coefficient is typeset using the \xintFrac macro from the xintfrac package. This macro is *f*-expandable in the sense that it prepares expandably the whole expression with the multiple `\cfrac`'s, but it is not completely expandable naturally.

30.3 \xintGCFrac

f ★ `\xintGCFrac{a+b/c+d/e+f/g+h/...}` uses similarly `\cfrac` to typeset a generalized continued fraction in inline format. It admits the same optional argument as `\xintCFrac`.

```
\[\xintGCFrac {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}}\]
```

$$1+\cfrac{3375\cdot 10^{-3}}{\frac{1}{7}-\cfrac{\frac{3}{5}}{720}}$$

As can be seen this is typesetting macro, although it does proceed to the evaluation of the coefficients themselves. See \xintGCtoF if you are impatient to see this fraction computed. Numerators and denominators are made arguments to the \xintFrac macro.

30.4 \xintGCtoGCx

nnf ★ `\xintGCtoGCx{sepa}{sepb}{a+b/c+d/e+f/...+x/y}` returns the list of the coefficients of the generalized continued fraction of `f`, each one within a pair of braces, and separated with the help of `sepa` and `sepb`. Thus

`\xintGCtoGCx :;{1+2/3+4/5+6/7}` gives 1:2;3:4;5:6;7

Plain TeX+`amstex` users may be interested in:

```
$$\xintGCtoGCx {+\cfrac}{\\}{a+b/...}\endcfrac$$
$$\xintGCtoGCx {+\cfrac\xintFwOver}{\\\xintFwOver}{a+b/...}\endcfrac$$
```

30.5 \xintFtoCs

Frac *f* ★ `\xintFtoCs{f}` returns the comma separated list of the coefficients of the simple continued fraction of f.

```
\[ \xintSignedFrac{-5262046/89233} = [\xintFtoCs{-5262046/89233}]\]
```

$$-\frac{5262046}{89233} = [-59, 33, 27, 100]$$

30.6 \xintFtoCx

Frac
n f ★ `\xintFtoCx{sep}{f}` returns the list of the coefficients of the simple continued fraction of f, withing group braces and separated with the help of sep.

```
$$\xintFtoCx {+\cfrac1\\ }{f}\endcfrac$$
```

will display the continued fraction in `\cfrac` format, with Plain TEX and amstex.

30.7 \xintFtoGC

Frac
f ★ `\xintFtoGC{f}` does the same as `\xintFtoCx{+1/}{f}`. Its output may thus be used in the package macros expecting such an 'inline format'. This continued fraction is a *simple* one, not a *generalized* one, but as it is produced in the format used for user input of generalized continued fractions, the macro was called `\xintFtoGC` rather than `\xintFtoC` for example.

```
566827/208524=\xintFtoGC {566827/208524}
```

566827/208524=2+1/1+1/2+1/1+1/1+1/4+1/1+1/1+1/6+1/1+1/1+1/8+1/1+1/1+1/11

30.8 \xintFtoCC

Frac
f ★ `\xintFtoCC{f}` returns the 'centered' continued fraction of f, in 'inline format'.

```
566827/208524=\xintFtoCC {566827/208524}
```

566827/208524=3+-1/4+-1/2+1/5+-1/2+1/7+-1/2+1/9+-1/2+1/11

```
\[\xintFrac{566827/208524} = \xintGCFrac{\xintFtoCC{566827/208524}}\]
```

$$\frac{566827}{208524} = 3 - \cfrac{1}{4 - \cfrac{1}{2 + \cfrac{1}{5 - \cfrac{1}{2 + \cfrac{1}{7 - \cfrac{1}{2 + \cfrac{1}{9 - \cfrac{1}{2 + \cfrac{1}{11}}}}}}}}}$$

30.9 \xintFtoCv

Frac
f ★ `\xintFtoCv{f}` returns the list of the (braced) convergents of f, with no separator. To be treated with `\xintAssignArray` or `\xintListWithSep`.

```
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintFtoCv{5211/3748}}}\]
```

$$1 \to \frac{3}{2} \to \frac{4}{3} \to \frac{7}{5} \to \frac{25}{18} \to \frac{32}{23} \to \frac{57}{41} \to \frac{317}{228} \to \frac{374}{269} \to \frac{691}{497} \to \frac{5211}{3748}$$

30.10 \xintFtoCCv

Frac f ★ \xintFtoCCv{f} returns the list of the (braced) centered convergents of f, with no separator. To be treated with \xintAssignArray or \xintListWithSep.

```
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintFtoCCv{5211/3748}}}\]
```

$$1 \to \frac{4}{3} \to \frac{7}{5} \to \frac{32}{23} \to \frac{57}{41} \to \frac{374}{269} \to \frac{691}{497} \to \frac{5211}{3748}$$

30.11 \xintCstoF

f ★ \xintCstoF{a,b,c,d,...,z} computes the fraction corresponding to the coefficients, which may be fractions or even macros expanding to such fractions. The final fraction may then be highly reducible.

```
\[\xintGCFrac {-1+1/3+1/-5+1/7+1/-9+1/11+1/-13}
=\xintSignedFrac{\xintCstoF {-1,3,-5,7,-9,11,-13}}
=\xintSignedFrac{\xintGCtoF {-1+1/3+1/-5+1/7+1/-9+1/11+1/-13}}\]
```

$$-1+\cfrac{1}{3+\cfrac{1}{-5+\cfrac{1}{7+\cfrac{1}{-9+\cfrac{1}{11+\cfrac{1}{-13}}}}}} = -\frac{75887}{118187} = -\frac{75887}{118187}$$

```
\xintGCFrac{{1/2}+1/{1/3}+1/{1/4}+1/{1/5}}=
\xintFrac{\xintCstoF {1/2,1/3,1/4,1/5}}
```

$$\frac{1}{2}+\cfrac{1}{\frac{1}{3}+\cfrac{1}{\frac{1}{4}+\cfrac{1}{\frac{1}{5}}}} = \frac{159}{66}$$

A generalized continued fraction may produce a reducible fraction (\xintCstoF tries its best not to accumulate in a silly way superfluous factors but will not do simplifications which would be obvious to a human, like simplification by 3 in the result above).

30.12 \xintCstoCv

f ★ \xintCstoCv{a,b,c,d,...,z} returns the list of the corresponding convergents. It is allowed to use fractions as coefficients (the computed convergents have then no reason to be the real convergents of the final fraction). When the coefficients are integers, the convergents are irreducible fractions, but otherwise it is not necessarily the case.

```
\xintListWithSep:{\xintCstoCv{1,2,3,4,5,6}}
1/1:3/2:10/7:43/30:225/157:1393/972
\xintListWithSep:{\xintCstoCv{1,1/2,1/3,1/4,1/5,1/6}}
```

```
1/1:3/1:9/7:45/19:225/159:1575/729
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintCstoCv
{\xintPow {-.3}{-5},7.3/4.57,\xintCstoF{3/4,9,-1/3}}}}\]
```

$$\frac{-100000}{243} \to \frac{-72888949}{177390} \to \frac{-2700356878}{6567804}$$

30.13 \xintCstoGC

$f\star$ `\xintCstoGC{a,b,..,z}` transforms a comma separated list (or something expanding to such a list) into an 'inline format' continued fraction `{a}+1/{b}+1/...+1/{z}`. The coefficients are just copied and put within braces, without expansion. The output can then be used in `\xintGCFrac` for example.

```
\[\xintGCFrac {\xintCstoGC {-1,1/2,-1/3,1/4,-1/5}}
=\xintSignedFrac {\xintCstoF {-1,1/2,-1/3,1/4,-1/5}}\]
```

$$-1+\cfrac{1}{\frac{1}{2}+\cfrac{1}{\frac{-1}{3}+\cfrac{1}{\frac{1}{4}+\cfrac{1}{\frac{-1}{5}}}}} = -\frac{145}{83}$$

30.14 \xintGCtoF

$f\star$ `\xintGCtoF{a+b/c+d/e+f/g+......+v/w+x/y}` computes the fraction defined by the inline generalized continued fraction. Coefficients may be fractions but must then be put within braces. They can be macros. The plus signs are mandatory.

```
\[\xintGCFrac {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}} =
\xintFrac{\xintGCtoF {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}}} =
\xintFrac{\xintIrr{\xintGCtoF
                   {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}}}}\]
```

$$1+\cfrac{3375\cdot 10^{-3}}{\frac{1}{7}-\cfrac{\frac{3}{5}}{720}} = \frac{88629000}{3579000} = \frac{29543}{1193}$$

```
\[ \xintGCFrac{{1/2}+{2/3}/{4/5}+{1/2}/{1/5}+{3/2}/{5/3}} =
   \xintFrac{\xintGCtoF {{1/2}+{2/3}/{4/5}+{1/2}/{1/5}+{3/2}/{5/3}}} \]
```

$$\frac{1}{2}+\cfrac{\frac{2}{3}}{\frac{4}{5}+\cfrac{\frac{1}{2}}{\frac{1}{5}+\cfrac{\frac{3}{2}}{\frac{5}{3}}}} = \frac{4270}{4140}$$

The macro tries its best not to accumulate superfluous factor in the denominators, but doesn't reduce the fraction to irreducible form before returning it and does not do simplifications which would be obvious to a human.

30.15 \xintGCtoCv

$f\star$ `\xintGCtoCv{a+b/c+d/e+f/g+......+v/w+x/y}` returns the list of the corresponding convergents. The coefficients may be fractions, but must then be inside braces. Or they may be macros, too.

The convergents will in the general case be reducible. To put them into irreducible form, one needs one more step, for example it can be done with `\xintApply\xintIrr`.

```
\[\xintListWithSep{,}{\xintApply\xintFrac
                {\xintGCtoCv{3+{-2}/{7/2}+{3/4}/12+{-56}/3}}}\]
\[\xintListWithSep{,}{\xintApply\xintFrac{\xintApply\xintIrr
                {\xintGCtoCv{3+{-2}/{7/2}+{3/4}/12+{-56}/3}}}}\]
```

$$3, \frac{17}{7}, \frac{834}{342}, \frac{1306}{542}$$

$$3, \frac{17}{7}, \frac{139}{57}, \frac{653}{271}$$

30.16 \xintCntoF

$^{\mathrm{num}}_{x} f\star$ `\xintCntoF{N}{\macro}` computes the fraction `f` having coefficients `c(j)=\macro{j}` for `j=0,1,...,N`. The `N` parameter is given to a `\numexpr`. The values of the coefficients, as returned by `\macro` do not have to be positive, nor integers, and it is thus not necessarily the case that the original `c(j)` are the true coefficients of the final `f`.

```
\def\macro #1{\the\numexpr 1+#1*#1\relax}\xintCntoF {5}{\macro}
                        72625/49902[0]
```

30.17 \xintGCntoF

$^{\mathrm{num}}_{x} ff\star$ `\xintGCntoF{N}{\macroA}{\macroB}` returns the fraction `f` corresponding to the inline generalized continued fraction `a0+b0/a1+b1/a2+....+b(N-1)/aN`, with `a(j)=\macroA{j}` and `b(j)=\macroB{j}`. The `N` parameter is given to a `\numexpr`.

$$1+\cfrac{1}{2-\cfrac{1}{3+\cfrac{1}{1-\cfrac{1}{2+\cfrac{1}{3-\cfrac{1}{1}}}}}}=\frac{39}{25}$$

There is also \xintGCntoGC to get the 'inline format' continued fraction. The previous display was obtained with:

```
\def\coeffA #1{\the\numexpr #1+4-3*((#1+2)/3)\relax }%
    \def\coeffB #1{\xintMON{#1}}% (-1)^n
  \[\xintGCFrac{\xintGCntoGC {6}{\coeffA}{\coeffB}}
   = \xintFrac{\xintGCntoF {6}{\coeffA}{\coeffB}}\]
```

30.18 \xintCntoCs

$^{\text{num}}_{x} f \star$ \xintCntoCs{N}{\macro} produces the comma separated list of the corresponding coefficients, from n=0 to n=N. The N is given to a \numexpr.

```
\def\macro #1{\the\numexpr 1+#1*#1\relax}
\xintCntoCs {5}{\macro}->1,2,5,10,17,26
\[\xintFrac{\xintCntoF {5}{\macro}}=\xintCFrac{\xintCntoF {5}{\macro}}\]
```

$$\frac{72625}{49902} = 1 + \cfrac{1}{2 + \cfrac{1}{5 + \cfrac{1}{10 + \cfrac{1}{17 + \cfrac{1}{26}}}}}$$

30.19 \xintCntoGC

$^{\text{num}}_{x} f \star$ \xintCntoGC{N}{\macro} evaluates the c(j)=\macro{j} from j=0 to j=N and returns a continued fraction written in inline format: {c(0)}+1/{c(1)}+1/...+1/{c(N)}. The parameter N is given to a \numexpr. The coefficients, after expansion, are, as shown, being enclosed in an added pair of braces, they may thus be fractions.

```
\def\macro #1{\the\numexpr\ifodd#1 -1-#1\else1+#1\fi\relax/%
              \the\numexpr 1+#1*#1\relax}
\edef\x{\xintCntoGC {5}{\macro}}\meaning\x
macro:->{1/1}+1/{-2/2}+1/{3/5}+1/{-4/10}+1/{5/17}+1/{-6/26}
\[\xintGCFrac{\xintCntoGC {5}{\macro}}\]
```

$$1 + \cfrac{1}{\frac{-2}{2} + \cfrac{1}{\frac{3}{5} + \cfrac{1}{\frac{-4}{10} + \cfrac{1}{\frac{5}{17} + \cfrac{1}{\frac{-6}{26}}}}}}$$

30.20 \xintGCntoGC

$^{\text{num}}_{x} ff \star$ \xintGCntoGC{N}{\macroA}{\macroB} evaluates the coefficients and then returns the corresponding {a0}+{b0}/{a1}+{b1}/{a2}+...+{b(N-1)}/{aN} inline generalized fraction. N is givent to a \numexpr. As shown, the coefficients are enclosed into added pairs of braces, and may thus be fractions.

```
  #1*#1*#1+1\relax}%
\def\bn #1{\the\numexpr \xintiiMON{#1}*(#1+1)\relax}%
$\xintGCntoGC {5}{\an}{\bn}}=\xintGCFrac {\xintGCntoGC {5}{\an}{\bn}} =
\displaystyle\xintFrac {\xintGCntoF {5}{\an}{\bn}}$\par
```

$$1 + 1/2 + -2/9 + 3/28 + -4/65 + 5/126 = 1 + \cfrac{1}{2 - \cfrac{2}{9 + \cfrac{3}{28 - \cfrac{4}{65 + \cfrac{5}{126}}}}} = \frac{5797655}{3712466}$$

30.21 \xintiCstoF, \xintiGCtoF, \xintiCstoCv, \xintiGCtoCv

The same as the corresponding macros without the 'i', but for integer-only input. Infinitesimally faster; to notice the higher efficiency one would need to use them with an input having (at least) hundreds of coefficients.

30.22 \xintGCtoGC

f ★ `\xintGCtoGC{a+b/c+d/e+f/g+......+v/w+x/y}` expands (with the usual meaning) each one of the coefficients and returns an inline continued fraction of the same type, each expanded coefficient being enclosed withing braces.

```
{6}+\xintCstoF {2,-7,-5}/16}} \meaning\x
macro:->{1}+{3375/1[-3]}/{1/7}+{-3/5}/{720}+{67/36}/{16}
```

To be honest I have, it seems, forgotten why I wrote this macro in the first place.

31 Package xinttools implementation

Release 1.09g splits off xinttools.sty from xint.sty.

Contents

31.1 Catcodes, ε-TEX and reload detection

The method for package identification and reload detection is copied verbatim from the packages by Heiko Oberdiek (with some modifications starting with release 1.09b).

The method for catcodes was also inspired by these packages, we proceed slightly differently.

Starting with version 1.06 of the package, also ` must be catcode-protected, because we replace everywhere in the code the twice-expansion done with \expandafter by the systematic use of \romannumeral-`0.

Starting with version 1.06b I decide that I suffer from an indigestion of @ signs, so I replace them all with underscores _, à la LaTeX3.

Release 1.09b is more economical: some macros are defined already in xint.sty (now xinttools.sty) and re-used in other modules. All catcode changes have been unified and \XINT_storecatcodes will be used by each module to redefine \XINT_restorecatcodes_endinput in case catcodes have changed in-between the loading of xint.sty (now xinttools.sty) and the module (not very probable but...).

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
```

```
\endlinechar=13 %
\catcode123=1   % {
\catcode125=2   % }
\catcode64=11   % @
\catcode95=11   % _
\catcode35=6    % #
\catcode44=12   % ,
\catcode45=12   % -
\catcode46=12   % .
\catcode58=12   % :
\expandafter\let\expandafter\x\csname ver@xint.sty\endcsname
\expandafter
  \ifx\csname PackageInfo\endcsname\relax
    \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
  \else
    \def\y#1#2{\PackageInfo{#1}{#2}}%
  \fi
\expandafter
\ifx\csname numexpr\endcsname\relax
   \y{xinttools}{\numexpr not available, aborting input}%
   \aftergroup\endinput
\else
  \ifx\x\relax % plain-TeX, first loading
  \else
    \def\empty {}%
    \ifx\x\empty % LaTeX, first loading,
    % variable is initialized, but \ProvidesPackage not yet seen
    \else
      \y{xinttools}{I was already loaded, aborting input}%
      \aftergroup\endinput
    \fi
  \fi
\fi
\def\ChangeCatcodesIfInputNotAborted
{%
    \endgroup
    \def\XINT_storecatcodes
    {% takes care of all, to allow more economical code in modules
         \catcode34=\the\catcode34   % " xintbinhex, and 1.09k xintexpr
         \catcode63=\the\catcode63   % ? xintexpr
         \catcode124=\the\catcode124 % | xintexpr
         \catcode38=\the\catcode38   % & xintexpr
         \catcode64=\the\catcode64   % @ xintexpr
         \catcode33=\the\catcode33   % ! xintexpr
         \catcode93=\the\catcode93   % ] -, xintfrac, xintseries, xintcfrac
         \catcode91=\the\catcode91   % [ -, xintfrac, xintseries, xintcfrac
         \catcode36=\the\catcode36   % $ xintgcd only
      \catcode94=\the\catcode94   % ^
      \catcode96=\the\catcode96   % `
```

```
      \catcode47=\the\catcode47   % /
      \catcode41=\the\catcode41   % )
      \catcode40=\the\catcode40   % (
      \catcode42=\the\catcode42   % *
      \catcode43=\the\catcode43   % +
      \catcode62=\the\catcode62   % >
      \catcode60=\the\catcode60   % <
      \catcode58=\the\catcode58   % :
      \catcode46=\the\catcode46   % .
      \catcode45=\the\catcode45   % -
      \catcode44=\the\catcode44   % ,
      \catcode35=\the\catcode35   % #
      \catcode95=\the\catcode95   % _
      \catcode125=\the\catcode125 % }
      \catcode123=\the\catcode123 % {
      \endlinechar=\the\endlinechar
      \catcode13=\the\catcode13   % ^^M
      \catcode32=\the\catcode32   %
      \catcode61=\the\catcode61\relax   % =
    }%
    \edef\XINT_restorecatcodes_endinput
    {%
         \XINT_storecatcodes\noexpand\endinput %
    }%
    \def\XINT_setcatcodes
    {%
      \catcode61=12   % =
      \catcode32=10   % space
      \catcode13=5    % ^^M
      \endlinechar=13 %
      \catcode123=1   % {
      \catcode125=2   % }
      \catcode95=11   % _ (replaces @ everywhere, starting with 1.06b)
      \catcode35=6    % #
      \catcode44=12   % ,
      \catcode45=12   % -
      \catcode46=12   % .
      \catcode58=11   % : (made letter for error cs)
      \catcode60=12   % <
      \catcode62=12   % >
      \catcode43=12   % +
      \catcode42=12   % *
      \catcode40=12   % (
      \catcode41=12   % )
      \catcode47=12   % /
      \catcode96=12   % ` (for ubiquitous \romannumeral-`0 and some \catcode )
      \catcode94=11   % ^
    \catcode36=3   % $
    \catcode91=12  % [
```

```
        \catcode93=12  % ]
        \catcode33=11  % !
        \catcode64=11  % @
        \catcode38=12  % &
        \catcode124=12 % |
        \catcode63=11  % ?
        \catcode34=12  % " missing from v < 1.09k although needed in xintbinhex
        }%
        \XINT_setcatcodes
    }%
\ChangeCatcodesIfInputNotAborted
\def\XINTsetupcatcodes {% for use by other modules
        \edef\XINT_restorecatcodes_endinput
        {%
             \XINT_storecatcodes\noexpand\endinput %
        }%
        \XINT_setcatcodes
}%
```

31.2 Package identification

Inspired from Heiko Oberdiek's packages. Modified in 1.09b to allow re-use in the other modules. Also I assume now that if \ProvidesPackage exists it then does define \ver@<pkgname>.sty, code of HO for some reason escaping me (compatibility with LaTeX 2.09 or other things ??) seems to set extra precautions.

1.09c uses e-TeX \ifdefined.

```
\ifdefined\ProvidesPackage
  \let\XINT_providespackage\relax
\else
  \def\XINT_providespackage #1#2[#3]%
              {\immediate\write-1{Package: #2 #3}%
               \expandafter\xdef\csname ver@#2.sty\endcsname{#3}}%
\fi
\XINT_providespackage
\ProvidesPackage {xinttools}%
  [2014/02/13 v1.09kb Expandable and non-expandable utilities (jfB)]%
```

31.3 Token management, constants

In 1.09e \xint_undef replaced everywhere by \xint_bye. Release 1.09h makes most everything \long.

```
\long\def\xint_gobble_     {}%
\long\def\xint_gobble_i    #1{}%
\long\def\xint_gobble_ii   #1#2{}%
\long\def\xint_gobble_iii  #1#2#3{}%
\long\def\xint_gobble_iv   #1#2#3#4{}%
\long\def\xint_gobble_v    #1#2#3#4#5{}%
```

```
\long\def\xint_gobble_vi   #1#2#3#4#5#6{}%
\long\def\xint_gobble_vii  #1#2#3#4#5#6#7{}%
\long\def\xint_gobble_viii #1#2#3#4#5#6#7#8{}%
\long\def\xint_firstofone  #1{#1}%
\xint_firstofone{\let\XINT_sptoken= } %<- space here!
\long\def\xint_firstoftwo  #1#2{#1}%
\long\def\xint_secondoftwo #1#2{#2}%
\long\def\xint_firstoftwo_thenstop  #1#2{ #1}%
\long\def\xint_secondoftwo_thenstop #1#2{ #2}%
\def\xint_minus_thenstop { -}%
\long\def\xint_gob_til_R #1\R {}%
\long\def\xint_gob_til_W #1\W {}%
\long\def\xint_gob_til_Z #1\Z {}%
\long\def\xint_bye #1\xint_bye {}%
\let\xint_relax\relax
\def\xint_brelax {\xint_relax }%
\long\def\xint_gob_til_xint_relax #1\xint_relax {}%
\long\def\xint_afterfi #1#2\fi {\fi #1}%
\chardef\xint_c_    0
\chardef\xint_c_i   1 % 1.09k did not have it, but needed in \xintSeq
\chardef\xint_c_viii 8
\newtoks\XINT_toks
```

31.4 \xintodef, \xintgodef, \odef

1.09i. For use in \xintAssign. No parameter text! 1.09j uses \xint... rather than \XINT_.... \xintAssign [o] will use the preexisting \odef if there was one before xint' loading.

```
\def\xintodef #1{\expandafter\def\expandafter#1\expandafter }%
\ifdefined\odef\else\let\odef\xintodef\fi
\def\xintgodef {\global\xintodef }%
```

31.5 \xintoodef, \xintgoodef, \oodef

1.09i. Can be prefixed with \global. No parameter text. The alternative

```
\def\oodef #1#{\def\XINT_tmpa{#1}
     \expandafter\expandafter\expandafter\expandafter
     \expandafter\expandafter\expandafter\def
     \expandafter\expandafter\expandafter\XINT_tmpa
     \expandafter\expandafter\expandafter }
```

could not be prefixed by \global. Anyhow, macro parameter tokens would have to somehow not be seen by expanded stuff, except if designed for it. \xintAssign [oo] (etc...) uses the pre-existing \oodef if there was one.

```
\def\xintoodef #1{\expandafter\expandafter\expandafter\def
                  \expandafter\expandafter\expandafter#1%
                  \expandafter\expandafter\expandafter }%
```

```
\ifdefined\oodef\else\let\oodef\xintoodef\fi
\def\xintgoodef {\global\xintoodef }%
```

31.6 \xintfdef, \xintgfdef, \fdef

1.09i. No parameter text!

```
\def\xintfdef #1#2{\expandafter\def\expandafter#1\expandafter
                           {\romannumeral-`0#2}}%
\ifdefined\fdef\else\let\fdef\xintfdef\fi
\def\xintgfdef {\global\xintfdef }% should be \global\fdef if \fdef pre-exists?
```

31.7 \xintReverseOrder

\xintReverseOrder: does NOT expand its argument.

```
\def\xintReverseOrder {\romannumeral0\xintreverseorder }%
\long\def\xintreverseorder #1%
{%
    \XINT_rord_main {}#1%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax
}%
\long\def\XINT_rord_main #1#2#3#4#5#6#7#8#9%
{%
    \xint_bye #9\XINT_rord_cleanup\xint_bye
    \XINT_rord_main {#9#8#7#6#5#4#3#2#1}%
}%
\long\edef\XINT_rord_cleanup\xint_bye\XINT_rord_main #1#2\xint_relax
{%
    \noexpand\expandafter\space\noexpand\xint_gob_til_xint_relax #1%
}%
```

31.8 \xintRevWithBraces

New with 1.06. Makes the expansion of its argument and then reverses the resulting tokens or braced tokens, adding a pair of braces to each (thus, maintaining it when it was already there.

As in some other places, 1.09e replaces \Z by \xint_bye, although here it is just for coherence of notation as \Z would be perfectly safe. The reason for \xint_relax, here and in other locations, is in case #1 expands to nothing, the \romannumeral-`0 must be stopped

```
\def\xintRevWithBraces          {\romannumeral0\xintrevwithbraces }%
\def\xintRevWithBracesNoExpand {\romannumeral0\xintrevwithbracesnoexpand }%
\long\def\xintrevwithbraces #1%
```

```
{%
    \expandafter\XINT_revwbr_loop\expandafter{\expandafter}%
    \romannumeral-‘0#1\xint_relax\xint_relax\xint_relax\xint_relax
                      \xint_relax\xint_relax\xint_relax\xint_relax\xint_bye
}%
\long\def\xintrevwithbracesnoexpand #1%
{%
    \XINT_revwbr_loop {}%
    #1\xint_relax\xint_relax\xint_relax\xint_relax
      \xint_relax\xint_relax\xint_relax\xint_relax\xint_bye
}%
\long\def\XINT_revwbr_loop #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint_relax #9\XINT_revwbr_finish_a\xint_relax
    \XINT_revwbr_loop {{#9}{#8}{#7}{#6}{#5}{#4}{#3}{#2}#1}%
}%
\long\def\XINT_revwbr_finish_a\xint_relax\XINT_revwbr_loop #1#2\xint_bye
{%
    \XINT_revwbr_finish_b #2\R\R\R\R\R\R\R\Z #1%
}%
\def\XINT_revwbr_finish_b #1#2#3#4#5#6#7#8\Z
{%
    \xint_gob_til_R
            #1\XINT_revwbr_finish_c 8%
            #2\XINT_revwbr_finish_c 7%
            #3\XINT_revwbr_finish_c 6%
            #4\XINT_revwbr_finish_c 5%
            #5\XINT_revwbr_finish_c 4%
            #6\XINT_revwbr_finish_c 3%
            #7\XINT_revwbr_finish_c 2%
            \R\XINT_revwbr_finish_c 1\Z
}%
\def\XINT_revwbr_finish_c #1#2\Z
{%
    \expandafter\expandafter\expandafter
        \space
    \csname xint_gobble_\romannumeral #1\endcsname
}%
```

31.9 \xintLength

\xintLength does NOT expand its argument.
1.09g adds the missing \xintlength, which was previously called \XINT_length, and suppresses \XINT_Length
1.06: improved code is roughly 20% faster than the one from earlier versions.
1.09a, \xintnum inserted. 1.09e: \Z->\xint_bye as this is called from \xintNthElt, and there it was necessary not to use \Z. Later use of \Z and \W perfectly safe here.

```
\def\xintLength {\romannumeral0\xintlength }%
\long\def\xintlength #1%
{%
    \XINT_length_loop
    {0}#1\xint_relax\xint_relax\xint_relax\xint_relax
         \xint_relax\xint_relax\xint_relax\xint_relax\xint_bye
}%
\long\def\XINT_length_loop #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint_relax #9\XINT_length_finish_a\xint_relax
    \expandafter\XINT_length_loop\expandafter {\the\numexpr #1+8\relax}%
}%
\def\XINT_length_finish_a\xint_relax
    \expandafter\XINT_length_loop\expandafter #1#2\xint_bye
{%
    \XINT_length_finish_b #2\W\W\W\W\W\W\W\Z {#1}%
}%
\def\XINT_length_finish_b #1#2#3#4#5#6#7#8\Z
{%
    \xint_gob_til_W
            #1\XINT_length_finish_c 8%
            #2\XINT_length_finish_c 7%
            #3\XINT_length_finish_c 6%
            #4\XINT_length_finish_c 5%
            #5\XINT_length_finish_c 4%
            #6\XINT_length_finish_c 3%
            #7\XINT_length_finish_c 2%
            \W\XINT_length_finish_c 1\Z
}%
\edef\XINT_length_finish_c #1#2\Z #3%
   {\noexpand\expandafter\space\noexpand\the\numexpr #3-#1\relax}%
```

31.10 \xintZapFirstSpaces

1.09f, written [2013/11/01].

```
\def\xintZapFirstSpaces {\romannumeral0\xintzapfirstspaces }%
```

defined via an \edef in order to inject space tokens inside.

```
\long\edef\xintzapfirstspaces #1%
  {\noexpand\XINT_zapbsp_a \space #1\space\space\noexpand\xint_bye\xint_relax }%
\xint_firstofone {\long\def\XINT_zapbsp_a #1 } %<- space token here
{%
```

If the original #1 started with a space, here #1 will be in fact empty, so the effect will be to remove precisely one space from the original, because the first two space tokens are matched to the ones of the macro parameter text. If the original #1 did not start with a space then the #1 will be this original #1, with its added first space, up to the first <sp><sp> found. The added initial space will stop later the \romannumeral0. And in \xintZapLastSpaces we also carried along a space in order to be able to mix tne two codes in \xintZapSpaces. Testing for \emptiness of #1 is NOT done with an \if test because #1 may contain \if, \fi things (one could use a \detokenize method), and also because xint.sty has a style of its own for doing these things...

```
    \XINT_zapbsp_again? #1\xint_bye\XINT_zapbsp_b {#1}%
```

The #1 above is thus either empty, or it starts with a (char 32) space token followed with a non (char 32) space token and at any rate #1 is protected from brace stripping. It is assumed that the initial input does not contain space tokens of other than 32 as character code.

```
}%
\long\def\XINT_zapbsp_again? #1{\xint_bye #1\XINT_zapbsp_again }%
```

In the "empty" situation above, here #1=\xint_bye, else #1 could be some brace things, but unbracing will anyhow not reveal any \xint_bye. When we do below \XINT_zapbsp_again we recall that we have stripped two spaces out of <sp><original #1>, so we have one <sp> less in #1, and when we loop we better not forget to reinsert one initial <sp>.

```
\edef\XINT_zapbsp_again\XINT_zapbsp_b #1{\noexpand\XINT_zapbsp_a\space }%
```

We now have now gotten rid of the initial spaces, but #1 perhaps extend only to some initial chunk which was delimited by <sp><sp>.

```
\long\def\XINT_zapbsp_b #1#2\xint_relax
   {\XINT_zapbsp_end? #2\XINT_zapbsp_e\empty #2{#1}}%
```

If the initial chunk up to <sp><sp> (after stripping away the first spaces) was maximal, then #2 above is some spaces followed by \xint_bye, so in the \XINT_zapbsp_end? below it appears as \xint_bye, else the #1 below will not be nor give rise after brace removal to \xint_bye. And then the original \xint_bye in #2 will have the effect that all is swallowed and we continue with \XINT_zapbsp_e. If the chunk was maximal, then the #2 above contains as many space tokens as there were originally at the end.

```
\long\def\XINT_zapbsp_end? #1{\xint_bye #1\XINT_zapbsp_end }%
```

The #2 starts with a space which stops the \romannumeral. The #1 contains the same number of space tokens there was originally.

```
\long\def\XINT_zapbsp_end\XINT_zapbsp_e\empty #1\xint_bye #2{#2#1}%
```

Here the initial chunk was not maximal. So we need to get a second piece all the way up to \xint_bye, we take this opportunity to remove the two initially added ending space tokens. We inserted an \empty to prevent brace removal. The \expandafter get rid of the \empty.

```
\xint_firstofone{\long\def\XINT_zapbsp_e #1 } \xint_bye
    {\expandafter\XINT_zapbsp_f \expandafter{#1}}%
```

Let's not forget when we glue to reinsert the two intermediate space tokens.

```
\long\edef\XINT_zapbsp_f #1#2{#2\space\space #1}%
```

31.11 \xintZapLastSpaces

1.09f, written [2013/11/01].

```
\def\xintZapLastSpaces {\romannumeral0\xintzaplastspaces }%
```

Next macro is defined via an \edef for the space tokens.

```
\long\edef\xintzaplastspaces #1{\noexpand\XINT_zapesp_a {\space}\noexpand\empty
                                #1\space\space\noexpand\xint_bye \xint_relax}%
```

This creates a delimited macro with two space tokens:

```
\xint_firstofone {\long\def\XINT_zapesp_a #1#2 } %<- second space here
    {\expandafter\XINT_zapesp_b\expandafter{#2}{#1}}%
```

The \empty from \xintzaplastspaces is to prevent brace removal in the #2 above. The \expandafter chain removes it.

```
\long\def\XINT_zapesp_b #1#2#3\xint_relax
    {\XINT_zapesp_end? #3\XINT_zapesp_e {#2#1}\empty #3\xint_relax }%
```

When we have reached the ending space tokens, #3 is a bunch of spaces followed by \xint_bye. So the #1 below will be \xint_bye. In all other cases #1 can not be \xint_bye nor can it give birth to it via brace stripping.

```
\long\def\XINT_zapesp_end? #1{\xint_bye #1\XINT_zapesp_end }%
```

We are done. The #1 here has accumulated all the previous material. It started with a space token which stops the \romannumeral0. The reason for the space is the recycling of this code in \xintZapSpaces.

```
\long\def\XINT_zapesp_end\XINT_zapesp_e #1#2\xint_relax {#1}%
```

We haven't yet reached the end, so we need to re-inject two space tokens after what we have gotten so far. Then we loop. We might wonder why in \XINT_zapesp_b we scooped everything up to the end, rather than trying to test if the next thing was a bunch of spaces followed by \xint_bye\xint_relax. But how can we expandably examine what comes next? if we pick up something as undelimited parameter token we risk brace removal and we will never know about it so we cannot reinsert correctly; the only way is to gather a delimited macro parameter and be sure some token will be inside to forbid brace removal. I do not see (so far) any other way than scooping everything up to the end. Anyhow, 99% of the use cases will NOT have <sp><sp> inside!.

```
\long\edef\XINT_zapesp_e #1{\noexpand \XINT_zapesp_a {#1\space\space}}%
```

31.12 \xintZapSpaces

1.09f, written [2013/11/01].

```
\def\xintZapSpaces {\romannumeral0\xintzapspaces }%
```

We start like \xintZapStartSpaces.

```
\long\edef\xintzapspaces #1%
  {\noexpand\XINT_zapsp_a \space #1\space\space\noexpand\xint_bye\xint_relax}%
```

Once the loop stripping the starting spaces is done, we plug into the \xintZapLastSpaces code via \XINT_zapesp_b. As our #1 will always have an initial space, this is why we arranged code of \xintZapLastSpaces to do the same.

```
\xint_firstofone {\long\def\XINT_zapsp_a #1 } %<- space token here
{%
    \XINT_zapsp_again? #1\xint_bye\XINT_zapesp_b {#1}{}%
}%
\long\def\XINT_zapsp_again? #1{\xint_bye #1\XINT_zapsp_again }%
\long\edef\XINT_zapsp_again\XINT_zapesp_b #1#2{\noexpand\XINT_zapsp_a\space }%
```

31.13 \xintZapSpacesB

1.09f, written [2013/11/01].

```
\def\xintZapSpacesB {\romannumeral0\xintzapspacesb }%
\long\def\xintzapspacesb #1{\XINT_zapspb_one? #1\xint_relax\xint_relax
                            \xint_bye\xintzapspaces {#1}}%
\long\def\XINT_zapspb_one? #1#2%
   {\xint_gob_til_xint_relax #1\XINT_zapspb_onlyspaces\xint_relax
    \xint_gob_til_xint_relax #2\XINT_zapspb_bracedorone\xint_relax
    \xint_bye {#1}}%
\def\XINT_zapspb_onlyspaces\xint_relax
    \xint_gob_til_xint_relax\xint_relax\XINT_zapspb_bracedorone\xint_relax
    \xint_bye #1\xint_bye\xintzapspaces #2{ }%
\long\def\XINT_zapspb_bracedorone\xint_relax
    \xint_bye #1\xint_relax\xint_bye\xintzapspaces #2{ #1}%
```

31.14 \xintCSVtoList, \xintCSVtoListNonStripped

\xintCSVtoList transforms a,b,..,z into {a}{b}...{z}. The comma separated list may be a macro which is first expanded (protect the first item with a space if it is not to be expanded). First included in release 1.06. Here, use of \Z (and \R) perfectly safe.

[2013/11/02]: Starting with 1.09f, automatically filters items through \xintZapSpacesB to strip off all spaces around commas, and spaces at the start and end of the list. The original is kept as \xintCSVtoListNonStripped, and is faster. But ... it doesn't strip spaces.

```
\def\xintCSVtoList {\romannumeral0\xintcsvtolist }%
\long\def\xintcsvtolist #1{\expandafter\xintApply
           \expandafter\xintzapspacesb
           \expandafter{\romannumeral0\xintcsvtolistnonstripped{#1}}}%
\def\xintCSVtoListNoExpand {\romannumeral0\xintcsvtolistnoexpand }%
\long\def\xintcsvtolistnoexpand #1{\expandafter\xintApply
           \expandafter\xintzapspacesb
           \expandafter{\romannumeral0\xintcsvtolistnonstrippednoexpand{#1}}}%
\def\xintCSVtoListNonStripped {\romannumeral0\xintcsvtolistnonstripped }%
\def\xintCSVtoListNonStrippedNoExpand
         {\romannumeral0\xintcsvtolistnonstrippednoexpand }%
\long\def\xintcsvtolistnonstripped #1%
{%
    \expandafter\XINT_csvtol_loop_a\expandafter
    {\expandafter}\romannumeral-`0#1%
        ,\xint_bye,\xint_bye,\xint_bye,\xint_bye
        ,\xint_bye,\xint_bye,\xint_bye,\xint_bye,\Z
}%
\long\def\xintcsvtolistnonstrippednoexpand #1%
{%
    \XINT_csvtol_loop_a
    {}#1,\xint_bye,\xint_bye,\xint_bye,\xint_bye
        ,\xint_bye,\xint_bye,\xint_bye,\xint_bye,\Z
}%
\long\def\XINT_csvtol_loop_a #1#2,#3,#4,#5,#6,#7,#8,#9,%
{%
    \xint_bye #9\XINT_csvtol_finish_a\xint_bye
    \XINT_csvtol_loop_b {#1}{{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}}%
}%
\long\def\XINT_csvtol_loop_b #1#2{\XINT_csvtol_loop_a {#1#2}}%
\long\def\XINT_csvtol_finish_a\xint_bye\XINT_csvtol_loop_b #1#2#3\Z
{%
    \XINT_csvtol_finish_b #3\R,\R,\R,\R,\R,\R,\R,\Z #2{#1}%
}%
\def\XINT_csvtol_finish_b #1,#2,#3,#4,#5,#6,#7,#8\Z
{%
    \xint_gob_til_R
            #1\XINT_csvtol_finish_c 8%
            #2\XINT_csvtol_finish_c 7%
            #3\XINT_csvtol_finish_c 6%
            #4\XINT_csvtol_finish_c 5%
            #5\XINT_csvtol_finish_c 4%
            #6\XINT_csvtol_finish_c 3%
            #7\XINT_csvtol_finish_c 2%
            \R\XINT_csvtol_finish_c 1\Z
}%
\def\XINT_csvtol_finish_c #1#2\Z
{%
    \csname XINT_csvtol_finish_d\romannumeral #1\endcsname
```

```
}%
\long\def\XINT_csvtol_finish_dviii #1#2#3#4#5#6#7#8#9{ #9}%
\long\def\XINT_csvtol_finish_dvii  #1#2#3#4#5#6#7#8#9{ #9{#1}}%
\long\def\XINT_csvtol_finish_dvi   #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}}%
\long\def\XINT_csvtol_finish_dv    #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}}%
\long\def\XINT_csvtol_finish_div   #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}}%
\long\def\XINT_csvtol_finish_diii  #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}{#5}}%
\long\def\XINT_csvtol_finish_dii   #1#2#3#4#5#6#7#8#9%
                                            { #9{#1}{#2}{#3}{#4}{#5}{#6}}%
\long\def\XINT_csvtol_finish_di    #1#2#3#4#5#6#7#8#9%
                                            { #9{#1}{#2}{#3}{#4}{#5}{#6}{#7}}%
```

31.15 \xintListWithSep

\xintListWithSep {\sep}{{a}{b}...{z}} returns a \sep b \sep .... \sep z
Included in release 1.04. The 'sep' can be \par's: the macro xintlistwithsep etc... are all declared long. 'sep' does not have to be a single token. It is not expanded. The list may be a macro and it is expanded. 1.06 modifies the 'feature' of returning sep if the list is empty: the output is now empty in that case. (sep was not used for a one element list, but strangely it was for a zero-element list).

Use of \Z as delimiter was objectively an error, which I fix here in 1.09e, now the code uses \xint_bye.

```
\def\xintListWithSep {\romannumeral0\xintlistwithsep }%
\def\xintListWithSepNoExpand {\romannumeral0\xintlistwithsepnoexpand }%
\long\def\xintlistwithsep #1#2%
    {\expandafter\XINT_lws\expandafter {\romannumeral-`0#2}{#1}}%
\long\def\XINT_lws #1#2{\XINT_lws_start {#2}#1\xint_bye }%
\long\def\xintlistwithsepnoexpand #1#2{\XINT_lws_start {#1}#2\xint_bye }%
\long\def\XINT_lws_start #1#2%
{%
    \xint_bye #2\XINT_lws_dont\xint_bye
    \XINT_lws_loop_a {#2}{#1}%
}%
\long\def\XINT_lws_dont\xint_bye\XINT_lws_loop_a #1#2{ }%
\long\def\XINT_lws_loop_a #1#2#3%
{%
    \xint_bye #3\XINT_lws_end\xint_bye
    \XINT_lws_loop_b {#1}{#2#3}{#2}%
}%
\long\def\XINT_lws_loop_b #1#2{\XINT_lws_loop_a {#1#2}}%
\long\def\XINT_lws_end\xint_bye\XINT_lws_loop_b #1#2#3{ #1}%
```

31.16 \xintNthElt

\xintNthElt {i}{{a}{b}...{z}} (or 'tokens' abcd...z) returns the i th element (one pair of braces removed). The list is first expanded. First included in release 1.06. With 1.06a, a value of i = 0 (or negative) makes the macro return the length.

This is different from \xintLen which is for numbers (checks sign) and different from \xintLength which does not first expand its argument. With 1.09b, only i=0 gives the length, negative values return the i th element from the end. 1.09c has some slightly less quick initial preparation (if #2 is very long, not good to have it twice), I wanted to respect the noexpand directive in all cases, and the alternative would be to define more macros.

At some point I turned the \W's into \xint_relax's but forgot to modify accordingly \XINT_nthelt_finish. So in case the index is larger than the number of items the macro returned was an \xint_relax token rather than nothing. Fixed in 1.09e. I also take the opportunity of this fix to replace uses of \Z by \xint_bye. (and as a result I must do the change also in \XINT_length_loop and related macros).

```
\def\xintNthElt         {\romannumeral0\xintnthelt }%
\def\xintNthEltNoExpand {\romannumeral0\xintntheltnoexpand }%
\def\xintnthelt #1%
{%
    \expandafter\XINT_nthelt_a\expandafter {\the\numexpr #1}%
}%
\def\xintntheltnoexpand #1%
{%
    \expandafter\XINT_ntheltnoexpand_a\expandafter {\the\numexpr #1}%
}%
\long\def\XINT_nthelt_a #1#2%
{%
     \ifnum #1<0
         \xint_afterfi{\expandafter\XINT_nthelt_c\expandafter
                         {\romannumeral0\xintrevwithbraces {#2}}{-#1}}%
     \else
         \xint_afterfi{\expandafter\XINT_nthelt_c\expandafter
                           {\romannumeral-`0#2}{#1}}%
     \fi
}%
\long\def\XINT_ntheltnoexpand_a #1#2%
{%
     \ifnum #1<0
         \xint_afterfi{\expandafter\XINT_nthelt_c\expandafter
                         {\romannumeral0\xintrevwithbracesnoexpand {#2}}{-#1}}%
     \else
         \xint_afterfi{\expandafter\XINT_nthelt_c\expandafter
                           {#2}{#1}}%
     \fi
}%
\long\def\XINT_nthelt_c #1#2%
{%
    \ifnum #2>\xint_c_
          \expandafter\XINT_nthelt_loop_a
    \else
          \expandafter\XINT_length_loop
    \fi {#2}#1\xint_relax\xint_relax\xint_relax\xint_relax
```

```
        \xint_relax\xint_relax\xint_relax\xint_relax\xint_bye
}%
\def\XINT_nthelt_loop_a #1%
{%
    \ifnum #1>\xint_c_viii
        \expandafter\XINT_nthelt_loop_b
    \else
        \expandafter\XINT_nthelt_getit
    \fi
    {#1}%
}%
\long\def\XINT_nthelt_loop_b #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint_relax #9\XINT_nthelt_silentend\xint_relax
    \expandafter\XINT_nthelt_loop_a\expandafter{\the\numexpr #1-8}%
}%
\def\XINT_nthelt_silentend #1\xint_bye { }%
\def\XINT_nthelt_getit #1%
{%
    \expandafter\expandafter\expandafter\XINT_nthelt_finish
    \csname xint_gobble_\romannumeral\numexpr#1-1\endcsname
}%
\long\edef\XINT_nthelt_finish #1#2\xint_bye
   {\noexpand\xint_gob_til_xint_relax #1\noexpand\expandafter\space
                              \noexpand\xint_gobble_iii\xint_relax\space #1}%
```

31.17 \xintApply

\xintApply {\macro}{{a}{b}...{z}} returns {\macro{a}}...{\macro{b}} where each instance of \macro is ff-expanded. The list is first expanded and may thus be a macro. Introduced with release 1.04.

Modified in 1.09e to not use \Z but rather \xint_bye.

```
\def\xintApply          {\romannumeral0\xintapply }%
\def\xintApplyNoExpand {\romannumeral0\xintapplynoexpand }%
\long\def\xintapply #1#2%
{%
    \expandafter\XINT_apply\expandafter {\romannumeral-`0#2}%
    {#1}%
}%
\long\def\XINT_apply #1#2{\XINT_apply_loop_a {}{#2}#1\xint_bye }%
\long\def\xintapplynoexpand #1#2{\XINT_apply_loop_a {}{#1}#2\xint_bye }%
\long\def\XINT_apply_loop_a #1#2#3%
{%
    \xint_bye #3\XINT_apply_end\xint_bye
    \expandafter
    \XINT_apply_loop_b
    \expandafter {\romannumeral-`0#2{#3}}{#1}{#2}%
}%
```

```
\long\def\XINT_apply_loop_b #1#2{\XINT_apply_loop_a {#2{#1}}}%
\long\def\XINT_apply_end\xint_bye\expandafter\XINT_apply_loop_b
    \expandafter #1#2#3{ #2}%
```

31.18 \xintApplyUnbraced

\xintApplyUnbraced {\macro}{{a}{b}...{z}} returns \macro{a}...\macro{z} where each instance of \macro is expanded using \romannumeral-'0. The second argument may be a macro as it is first expanded itself (fully). No braces are added: this allows for example a non-expandable \def in \macro, without having to do \gdef. The list is first expanded. Introduced with release 1.06b. Define \macro to start with a space if it is not expandable or its execution should be delayed only when all of \macro{a}...\macro{z} is ready.

Modified in 1.09e to use \xint_bye rather than \Z.

```
\def\xintApplyUnbraced {\romannumeral0\xintapplyunbraced }%
\def\xintApplyUnbracedNoExpand {\romannumeral0\xintapplyunbracednoexpand }%
\long\def\xintapplyunbraced #1#2%
{%
    \expandafter\XINT_applyunbr\expandafter {\romannumeral-`0#2}%
    {#1}%
}%
\long\def\XINT_applyunbr #1#2{\XINT_applyunbr_loop_a {}{#2}#1\xint_bye }%
\long\def\xintapplyunbracednoexpand #1#2%
   {\XINT_applyunbr_loop_a {}{#1}#2\xint_bye }%
\long\def\XINT_applyunbr_loop_a #1#2#3%
{%
    \xint_bye #3\XINT_applyunbr_end\xint_bye
    \expandafter\XINT_applyunbr_loop_b
    \expandafter {\romannumeral-`0#2{#3}}{#1}{#2}%
}%
\long\def\XINT_applyunbr_loop_b #1#2{\XINT_applyunbr_loop_a {#2#1}}%
\long\def\XINT_applyunbr_end\xint_bye\expandafter\XINT_applyunbr_loop_b
    \expandafter #1#2#3{ #2}%
```

31.19 \xintSeq

1.09c. Without the optional argument puts stress on the input stack, should not be used to generated thousands of terms then. Here also, let's use \xint_bye rather than \Z as delimiter (1.09e; necessary change as #1 is used prior to being expanded, thus \Z might very well arise here as a macro).

```
\def\xintSeq {\romannumeral0\xintseq }%
\def\xintseq #1{\XINT_seq_chkopt  #1\xint_bye }%
\def\XINT_seq_chkopt #1%
{%
    \ifx [#1\expandafter\XINT_seq_opt
       \else\expandafter\XINT_seq_noopt
```

```
    \fi  #1%
}%
\def\XINT_seq_noopt #1\xint_bye #2%
{%
    \expandafter\XINT_seq\expandafter
       {\the\numexpr#1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_seq #1#2%
{%
   \ifcase\ifnum #1=#2 0\else\ifnum #2>#1 1\else -1\fi\fi\space
      \expandafter\xint_firstoftwo_thenstop
   \or
      \expandafter\XINT_seq_p
   \else
      \expandafter\XINT_seq_n
   \fi
   {#2}{#1}%
}%
\def\XINT_seq_p #1#2%
{%
    \ifnum #1>#2
      \expandafter\expandafter\expandafter\XINT_seq_p
    \else
      \expandafter\XINT_seq_e
    \fi
    \expandafter{\the\numexpr #1-\xint_c_i}{#2}{#1}%
}%
\def\XINT_seq_n #1#2%
{%
    \ifnum #1<#2
      \expandafter\expandafter\expandafter\XINT_seq_n
    \else
      \expandafter\XINT_seq_e
    \fi
     \expandafter{\the\numexpr #1+\xint_c_i}{#2}{#1}%
}%
\def\XINT_seq_e #1#2#3{ }%
\def\XINT_seq_opt [\xint_bye #1]#2#3%
{%
    \expandafter\XINT_seqo\expandafter
    {\the\numexpr #2\expandafter}\expandafter
    {\the\numexpr #3\expandafter}\expandafter
    {\the\numexpr #1}%
}%
\def\XINT_seqo #1#2%
{%
   \ifcase\ifnum #1=#2 0\else\ifnum #2>#1 1\else -1\fi\fi\space
      \expandafter\XINT_seqo_a
   \or
```

```
      \expandafter\XINT_seqo_pa
   \else
      \expandafter\XINT_seqo_na
   \fi
   {#1}{#2}%
}%
\def\XINT_seqo_a #1#2#3{ {#1}}%
\def\XINT_seqo_o #1#2#3#4{ #4}%
\def\XINT_seqo_pa #1#2#3%
{%
    \ifcase\ifnum #3=\xint_c_ 0\else\ifnum #3>\xint_c_ 1\else -1\fi\fi\space
           \expandafter\XINT_seqo_o
    \or
           \expandafter\XINT_seqo_pb
    \else
           \xint_afterfi{\expandafter\space\xint_gobble_iv}%
    \fi
    {#1}{#2}{#3}{{#1}}%
}%
\def\XINT_seqo_pb #1#2#3%
{%
    \expandafter\XINT_seqo_pc\expandafter{\the\numexpr #1+#3}{#2}{#3}%
}%
\def\XINT_seqo_pc #1#2%
{%
    \ifnum #1>#2
        \expandafter\XINT_seqo_o
    \else
        \expandafter\XINT_seqo_pd
    \fi
    {#1}{#2}%
}%
\def\XINT_seqo_pd #1#2#3#4{\XINT_seqo_pb {#1}{#2}{#3}{#4{#1}}}%
\def\XINT_seqo_na #1#2#3%
{%
    \ifcase\ifnum #3=\xint_c_ 0\else\ifnum #3>\xint_c_ 1\else -1\fi\fi\space
        \expandafter\XINT_seqo_o
    \or
        \xint_afterfi{\expandafter\space\xint_gobble_iv}%
    \else
        \expandafter\XINT_seqo_nb
    \fi
    {#1}{#2}{#3}{{#1}}%
}%
\def\XINT_seqo_nb #1#2#3%
{%
    \expandafter\XINT_seqo_nc\expandafter{\the\numexpr #1+#3}{#2}{#3}%
}%
\def\XINT_seqo_nc #1#2%
```

```
{%
    \ifnum #1<#2
        \expandafter\XINT_seqo_o
    \else
        \expandafter\XINT_seqo_nd
    \fi
    {#1}{#2}%
}%
\def\XINT_seqo_nd #1#2#3#4{\XINT_seqo_nb {#1}{#2}{#3}{#4{#1}}}%
```

31.20 \xintloop, \xintbreakloop, \xintbreakloopanddo, \xintloopskiptonext

1.09g [2013/11/22]. Made long with 1.09h.

```
\long\def\xintloop #1#2\repeat {#1#2\xintloop_again\fi\xint_gobble_i {#1#2}}%
\long\def\xintloop_again\fi\xint_gobble_i #1{\fi
                             #1\xintloop_again\fi\xint_gobble_i {#1}}%
\long\def\xintbreakloop #1\xintloop_again\fi\xint_gobble_i #2{}%
\long\def\xintbreakloopanddo #1#2\xintloop_again\fi\xint_gobble_i #3{#1}%
\long\def\xintloopskiptonext #1\xintloop_again\fi\xint_gobble_i #2{%
                        #2\xintloop_again\fi\xint_gobble_i {#2}}%
```

31.21 \xintiloop, \xintiloopindex, \xintouteriloopindex, \xintbreakiloop, \xintbreakiloopanddo, \xintiloopskiptonext, \xintiloopskipandredo

1.09g [2013/11/22]. Made long with 1.09h.

```
\def\xintiloop [#1+#2]{%
    \expandafter\xintiloop_a\the\numexpr #1\expandafter.\the\numexpr #2.}%
\long\def\xintiloop_a #1.#2.#3#4\repeat{%
    #3#4\xintiloop_again\fi\xint_gobble_iii {#1}{#2}{#3#4}}%
\def\xintiloop_again\fi\xint_gobble_iii #1#2{%
    \fi\expandafter\xintiloop_again_b\the\numexpr#1+#2.#2.}%
\long\def\xintiloop_again_b #1.#2.#3{%
    #3\xintiloop_again\fi\xint_gobble_iii {#1}{#2}{#3}}%
\long\def\xintbreakiloop #1\xintiloop_again\fi\xint_gobble_iii #2#3#4{}%
\long\def\xintbreakiloopanddo
     #1.#2\xintiloop_again\fi\xint_gobble_iii #3#4#5{#1}%
\long\def\xintiloopindex #1\xintiloop_again\fi\xint_gobble_iii #2%
                {#2#1\xintiloop_again\fi\xint_gobble_iii {#2}}%
\long\def\xintouteriloopindex #1\xintiloop_again
                         #2\xintiloop_again\fi\xint_gobble_iii #3%
   {#3#1\xintiloop_again #2\xintiloop_again\fi\xint_gobble_iii {#3}}%
\long\def\xintiloopskiptonext #1\xintiloop_again\fi\xint_gobble_iii #2#3{%
    \expandafter\xintiloop_again_b \the\numexpr#2+#3.#3.}%
\long\def\xintiloopskipandredo #1\xintiloop_again\fi\xint_gobble_iii #2#3#4{%
    #4\xintiloop_again\fi\xint_gobble_iii {#2}{#3}{#4}}%
```

31.22 \XINT_xflet

1.09e [2013/10/29]: we expand fully unbraced tokens and swallow arising space tokens until the dust settles. For treating cases {<blank>\x<blank>\y...}, with guaranteed expansion of the \x (which may itself give space tokens), a simpler approach is possible with doubled \romannumeral-`0, this is what I first did, but it had the feature that <sptoken><sptoken>\x would not expand the \x. At any rate, <sptoken>'s before the list terminator z were all correctly moved out of the way, hence the stuff was robust for use in (the then current versions of) \xintApplyInline and \xintFor. Although *two* space tokens would need devilishly prepared input, nevertheless I decided to also survive that, so here the method is a bit more complicated. But it simplifies things on the caller side.

```
\def\XINT_xflet #1%
{%
    \def\XINT_xflet_macro {#1}\XINT_xflet_zapsp
}%
\def\XINT_xflet_zapsp
{%
    \expandafter\futurelet\expandafter\XINT_token
    \expandafter\XINT_xflet_sp?\romannumeral-`0%
}%
\def\XINT_xflet_sp?
{%
    \ifx\XINT_token\XINT_sptoken
         \expandafter\XINT_xflet_zapsp
    \else\expandafter\XINT_xflet_zapspB
    \fi
}%
\def\XINT_xflet_zapspB
{%
    \expandafter\futurelet\expandafter\XINT_tokenB
    \expandafter\XINT_xflet_spB?\romannumeral-`0%
}%
\def\XINT_xflet_spB?
{%
    \ifx\XINT_tokenB\XINT_sptoken
         \expandafter\XINT_xflet_zapspB
    \else\expandafter\XINT_xflet_eq?
    \fi
}%
\def\XINT_xflet_eq?
{%
    \ifx\XINT_token\XINT_tokenB
         \expandafter\XINT_xflet_macro
    \else\expandafter\XINT_xflet_zapsp
    \fi
}%
```

31.23 \xintApplyInline

1.09a: \xintApplyInline\macro{{a}{b}...{z}} has the same effect as executing \macro{a} and then applying again \xintApplyInline to the shortened list {{b}...{z}} until nothing is left. This is a non-expandable command which will result in quicker code than using \xintApplyUnbraced. It expands (fully) its second (list) argument first, which may thus be encapsulated in a macro.

Release 1.09c has a new \xintApplyInline: the new version, while not expandable, is designed to survive when the applied macro closes a group, as is the case in alignemnts when it contains a & or \\. It uses catcode 3 Z as list terminator. Don't use it among the list items.

1.09d: the bug which was discovered in \xintFor* regarding space tokens at the very end of the item list also was in \xintApplyInline. The new version will expand unbraced item elements and this is in fact convenient to simulate insertion of lists in others.

1.09e: the applied macro is allowed to be long, with items (or the first fixed arguments of he macro, passed together with it as #1 to \xintApplyInline) containing explicit \par's. (1.09g: some missing \long's added)

1.09f: terminator used to be z, now Z (still catcode 3).

```
\catcode`Z 3
\long\def\xintApplyInline #1#2%
{%
  \long\expandafter\def\expandafter\XINT_inline_macro
  \expandafter ##\expandafter 1\expandafter {#1{##1}}%
  \XINT_xflet\XINT_inline_b #2Z% this Z has catcode 3
}%
\def\XINT_inline_b
{%
    \ifx\XINT_token Z\expandafter\xint_gobble_i
    \else\expandafter\XINT_inline_d\fi
}%
\long\def\XINT_inline_d #1%
{%
  \long\def\XINT_item{{#1}}\XINT_xflet\XINT_inline_e
}%
\def\XINT_inline_e
{%
    \ifx\XINT_token Z\expandafter\XINT_inline_w
    \else\expandafter\XINT_inline_f\fi
}%
\def\XINT_inline_f
{%
  \expandafter\XINT_inline_g\expandafter{\XINT_inline_macro {##1}}%
}%
\long\def\XINT_inline_g #1%
{%
   \expandafter\XINT_inline_macro\XINT_item
   \long\def\XINT_inline_macro ##1{#1}\XINT_inline_d
```

```
}%
\def\XINT_inline_w #1%
{%
   \expandafter\XINT_inline_macro\XINT_item
}%
```

31.24 \xintFor, \xintFor*, \xintBreakFor, \xintBreakForAndDo

1.09c [2013/10/09]: a new kind of loop which uses macro parameters #1, #2, #3, #4 rather than macros; while not expandable it survives executing code closing groups, like what happens in an alignment with the & character. When inserted in a macro for later use, the # character must be doubled.

The non-star variant works on a csv list, which it expands once, the star variant works on a token list, expanded fully.

1.09d: [2013/10/22] \xintFor* crashed when a space token was at the very end of the list. It is crucial in this code to not let the ending Z be picked up as a macro parameter without knowing in advance that it is its turn. So, we conscientiously clean out of the way space tokens, but also we ff-expand with \romannumeral-‘0 (unbraced) items, a process which may create new space tokens, so it is iterated. As unbraced items are expanded, it is easy to simulate insertion of a list in another. Unbraced items consecutive to an even (non-zero) number of space tokens will not get expanded.

1.09e: [2013/10/29] does this better, no difference between an even or odd number of explicit consecutive space tokens. Normal situations anyhow only create at most one space token, but well. There was a feature in \xintFor (not \xintFor*) from 1.09c that it treated an empty list as a list with one, empty, item. This feature is kept in 1.09e, knowingly... Also, macros are made long, hence the iterated text may contain \par and also the looped over items. I thought about providing some macro expanding to the loop count, but as the \xintFor is not expandable anyhow, there is no loss of generality if the iterated commands do themselves the bookkeeping using a count or a LaTeX counter, and deal with nesting or other problems. I can't do *everything*!

1.09e adds \XINT_forever with \xintintegers, \xintdimensions, \xintrationals and \xintBreakFor, \xintBreakForAndDo, \xintifForFirst, \xintifForLast. On this occasion \xint_firstoftwo and \xint_secondoftwo are made long.

1.09f: rewrites large parts of \xintFor code in order to filter the comma separated list via \xintCSVtoList which gets rid of spaces. Compatibility with \XINT_forever, the necessity to prevent unwanted brace stripping, and shared code with \xintFor*, make this all a delicate balancing act. The #1 in \XINT_for_forever? has an initial space token which serves two purposes: preventing brace stripping, and stopping the expansion made by \xintcsvtolist. If the \XINT_forever branch is taken, the added space will not be a problem there.

1.09f rewrites (2013/11/03) the code which now allows all macro parameters from #1 to #9 in \xintFor, \xintFor*, and \XINT_forever.

The 1.09f \xintFor and \xintFor* modified the value of \count 255 which was silly, 1.09g used \XINT_count, but requiring a \count only for that was also silly, 1.09h just uses \numexpr (all of that was only to get rid simply of a possibly space in #2...).

1.09ka [2014/02/05] corrects the following bug: \xintBreakFor and \xintBreak-ForAndDo could not be used in the last iteration.

```
\def\XINT_tmpa #1#2{\ifnum #2<#1 \xint_afterfi {{#########2}}\fi}%
\def\XINT_tmpb #1#2{\ifnum #1<#2 \xint_afterfi {{#########2}}\fi}%
\def\XINT_tmpc #1%
{%
    \expandafter\edef \csname XINT_for_left#1\endcsname
                {\xintApplyUnbraced {\XINT_tmpa #1}{123456789}}%
    \expandafter\edef \csname XINT_for_right#1\endcsname
                {\xintApplyUnbraced {\XINT_tmpb #1}{123456789}}%
}%
\xintApplyInline \XINT_tmpc {123456789}%
\long\def\xintBreakFor      #1Z{}%
\long\def\xintBreakForAndDo #1#2Z{#1}%
\def\xintFor {\let\xintifForFirst\xint_firstoftwo
              \futurelet\XINT_token\XINT_for_ifstar }%
\def\XINT_for_ifstar {\ifx\XINT_token*\expandafter\XINT_forx
                                  \else\expandafter\XINT_for \fi }%
\catcode`U 3 % with numexpr
\catcode`V 3 % with xintfrac.sty (xint.sty not enough)
\catcode`D 3 % with dimexpr
% \def\XINT_flet #1%
% {%
%     \def\XINT_flet_macro {#1}\XINT_flet_zapsp
% }%
\def\XINT_flet_zapsp
{%
    \futurelet\XINT_token\XINT_flet_sp?
}%
\def\XINT_flet_sp?
{%
    \ifx\XINT_token\XINT_sptoken
         \xint_afterfi{\expandafter\XINT_flet_zapsp\romannumeral0}%
    \else\expandafter\XINT_flet_macro
    \fi
}%
\long\def\XINT_for #1#2in#3#4#5%
{%
    \expandafter\XINT_toks\expandafter
        {\expandafter\XINT_for_d\the\numexpr #2\relax {#5}}%
    \def\XINT_flet_macro {\expandafter\XINT_for_forever?\space}%
    \expandafter\XINT_flet_zapsp #3Z%
}%
\def\XINT_for_forever? #1Z%
{%
    \ifx\XINT_token U\XINT_to_forever\fi
    \ifx\XINT_token V\XINT_to_forever\fi
    \ifx\XINT_token D\XINT_to_forever\fi
```

```
    \expandafter\the\expandafter\XINT_toks\romannumeral0\xintcsvtolist {#1}Z%
}%
\def\XINT_to_forever\fi #1\xintcsvtolist #2{\fi \XINT_forever #2}%
\long\def\XINT_forx *#1#2in#3#4#5%
{%
    \expandafter\XINT_toks\expandafter
       {\expandafter\XINT_forx_d\the\numexpr #2\relax {#5}}%
    \XINT_xflet\XINT_forx_forever? #3Z%
}%
\def\XINT_forx_forever?
{%
    \ifx\XINT_token U\XINT_to_forxever\fi
    \ifx\XINT_token V\XINT_to_forxever\fi
    \ifx\XINT_token D\XINT_to_forxever\fi
    \XINT_forx_empty?
}%
\def\XINT_to_forxever\fi #1\XINT_forx_empty? {\fi \XINT_forever }%
\catcode`U 11
\catcode`D 11
\catcode`V 11
\def\XINT_forx_empty?
{%
    \ifx\XINT_token Z\expandafter\xintBreakFor\fi
    \the\XINT_toks
}%
\long\def\XINT_for_d #1#2#3%
{%
  \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
  \XINT_toks {{#3}}%
  \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
                      \the\XINT_toks   \csname XINT_for_right#1\endcsname }%
  \XINT_toks {\XINT_x\let\xintifForFirst\xint_secondoftwo\XINT_for_d #1{#2}}%
  \futurelet\XINT_token\XINT_for_last?
}%
\long\def\XINT_forx_d #1#2#3%
{%
  \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
  \XINT_toks {{#3}}%
  \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
                      \the\XINT_toks   \csname XINT_for_right#1\endcsname }%
  \XINT_toks {\XINT_x\let\xintifForFirst\xint_secondoftwo\XINT_forx_d #1{#2}}%
  \XINT_xflet\XINT_for_last?
}%
\def\XINT_for_last?
{%
    \let\xintifForLast\xint_secondoftwo
    \ifx\XINT_token Z\let\xintifForLast\xint_firstoftwo
        \xint_afterfi{\xintBreakForAndDo{\XINT_x\xint_gobble_i Z}}\fi
    \the\XINT_toks
```

```
}%
```

31.25 \XINT_forever, \xintintegers, \xintdimensions, \xintrationals

New with 1.09e. But this used inadvertently \xintiadd/\xintimul which have the unnecessary \xintnum overhead. Changed in 1.09f to use \xintiiadd/\xintiimul which do not have this overhead. Also 1.09f has \xintZapSpacesB which helps getting rid of spaces for the \xintrationals case (the other cases end up inside a \numexpr, or \dimexpr, so not necessary).

```
\catcode`U 3
\catcode`D 3
\catcode`V 3
\let\xintegers      U%
\let\xintintegers   U%
\let\xintdimensions D%
\let\xintrationals  V%
\def\XINT_forever #1%
{%
  \expandafter\XINT_forever_a
  \csname XINT_?expr_\ifx#1UU\else\ifx#1DD\else V\fi\fi a\expandafter\endcsname
  \csname XINT_?expr_\ifx#1UU\else\ifx#1DD\else V\fi\fi i\expandafter\endcsname
  \csname XINT_?expr_\ifx#1UU\else\ifx#1DD\else V\fi\fi \endcsname
}%
\catcode`U 11
\catcode`D 11
\catcode`V 11
\def\XINT_?expr_Ua #1#2%
   {\expandafter{\expandafter\numexpr\the\numexpr #1\expandafter\relax
                               \expandafter\relax\expandafter}%
    \expandafter{\the\numexpr #2}}%
\def\XINT_?expr_Da #1#2%
   {\expandafter{\expandafter\dimexpr\number\dimexpr #1\expandafter\relax
                 \expandafter s\expandafter p\expandafter\relax\expandafter}%
    \expandafter{\number\dimexpr #2}}%
\catcode`Z 11
\def\XINT_?expr_Va #1#2%
{%
    \expandafter\XINT_?expr_Vb\expandafter
          {\romannumeral-`0\xintrawwithzeros{\xintZapSpacesB{#2}}}%
          {\romannumeral-`0\xintrawwithzeros{\xintZapSpacesB{#1}}}%
}%
\catcode`Z 3
\def\XINT_?expr_Vb #1#2{\expandafter\XINT_?expr_Vc #2.#1.}%
\def\XINT_?expr_Vc #1/#2.#3/#4.%
{%
     \xintifEq {#2}{#4}%
       {\XINT_?expr_Vf {#3}{#1}{#2}}%
       {\expandafter\XINT_?expr_Vd\expandafter
```

```
        {\romannumeral0\xintiimul {#2}{#4}}%
        {\romannumeral0\xintiimul {#1}{#4}}%
        {\romannumeral0\xintiimul {#2}{#3}}%
       }%
}%
\def\XINT_?expr_Vd #1#2#3{\expandafter\XINT_?expr_Ve\expandafter {#2}{#3}{#1}}%
\def\XINT_?expr_Ve #1#2{\expandafter\XINT_?expr_Vf\expandafter {#2}{#1}}%
\def\XINT_?expr_Vf #1#2#3{{#2/#3}{{0}{#1}{#2}{#3}}}%
\def\XINT_?expr_Ui {{\numexpr 1\relax}{1}}%
\def\XINT_?expr_Di {{\dimexpr 0pt\relax}{65536}}%
\def\XINT_?expr_Vi {{1/1}{0111}}%
\def\XINT_?expr_U #1#2%
   {\expandafter{\expandafter\numexpr\the\numexpr #1+#2\relax\relax}{#2}}%
\def\XINT_?expr_D #1#2%
   {\expandafter{\expandafter\dimexpr\the\numexpr #1+#2\relax sp\relax}{#2}}%
\def\XINT_?expr_V #1#2{\XINT_?expr_Vx #2}%
\def\XINT_?expr_Vx #1#2%
{%
     \expandafter\XINT_?expr_Vy\expandafter
        {\romannumeral0\xintiiadd {#1}{#2}}{#2}%
}%
\def\XINT_?expr_Vy #1#2#3#4%
{%
     \expandafter{\romannumeral0\xintiiadd {#3}{#1}/#4}{{#1}{#2}{#3}{#4}}%
}%
\def\XINT_forever_a #1#2#3#4%
{%
    \ifx #4[\expandafter\XINT_forever_opt_a
       \else\expandafter\XINT_forever_b
    \fi #1#2#3#4%
}%
\def\XINT_forever_b #1#2#3Z{\expandafter\XINT_forever_c\the\XINT_toks #2#3}%
\long\def\XINT_forever_c #1#2#3#4#5%
    {\expandafter\XINT_forever_d\expandafter #2#4#5{#3}Z}%
\def\XINT_forever_opt_a #1#2#3[#4+#5]#6Z%
{%
    \expandafter\expandafter\expandafter
    \XINT_forever_opt_c\expandafter\the\expandafter\XINT_toks
    \romannumeral-`0#1{#4}{#5}#3%
}%
\long\def\XINT_forever_opt_c #1#2#3#4#5#6{\XINT_forever_d #2{#4}{#5}#6{#3}Z}%
\long\def\XINT_forever_d #1#2#3#4#5%
{%
  \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#5}%
  \XINT_toks {{#2}}%
  \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
                      \the\XINT_toks   \csname XINT_for_right#1\endcsname }%
  \XINT_x
  \let\xintifForFirst\xint_secondoftwo
```

```
  \expandafter\XINT_forever_d\expandafter #1\romannumeral-`0#4{#2}{#3}#4{#5}%
}%
```

31.26 \xintForpair, \xintForthree, \xintForfour

1.09c: I don't know yet if {a}{b} is better for the user or worse than (a,b). I prefer the former. I am not very motivated to deal with spaces in the (a,b) approach which is the one (currently) followed here.

[2013/11/02] 1.09f: I may not have been very motivated in 1.09c, but since then I developped the \xintZapSpaces/\xintZapSpacesB tools (much to my satisfaction). Based on this, and better parameter texts, \xintForpair and its cousins now handle spaces very satisfactorily (this relies partly on the new \xintCSVtoList which makes use of \xintZapSpacesB). Does not share code with \xintFor anymore.

[2013/11/03] 1.09f: \xintForpair extended to accept #1#2, #2#3 etc... up to #8#9, \xintForthree, #1#2#3 up to #7#8#9, \xintForfour id.

```
\catcode`j 3
\long\def\xintForpair #1#2#3in#4#5#6%
{%
    \let\xintifForFirst\xint_firstoftwo
    \XINT_toks  {\XINT_forpair_d #2{#6}}%
    \expandafter\the\expandafter\XINT_toks #4jZ%
}%
\long\def\XINT_forpair_d #1#2#3(#4)#5%
{%
  \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
  \XINT_toks \expandafter{\romannumeral0\xintcsvtolist{ #4}}%
  \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
        \the\XINT_toks \csname XINT_for_right\the\numexpr#1+1\endcsname}%
  \let\xintifForLast\xint_secondoftwo
  \ifx #5j\expandafter\xint_firstoftwo
     \else\expandafter\xint_secondoftwo
  \fi
  {\let\xintifForLast\xint_firstoftwo
   \xintBreakForAndDo {\XINT_x \xint_gobble_i Z}}%
  \XINT_x
  \let\xintifForFirst\xint_secondoftwo\XINT_forpair_d #1{#2}%
}%
\long\def\xintForthree #1#2#3in#4#5#6%
{%
    \let\xintifForFirst\xint_firstoftwo
    \XINT_toks  {\XINT_forthree_d #2{#6}}%
    \expandafter\the\expandafter\XINT_toks #4jZ%
}%
\long\def\XINT_forthree_d #1#2#3(#4)#5%
{%
  \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
  \XINT_toks \expandafter{\romannumeral0\xintcsvtolist{ #4}}%
  \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
```

```
      \the\XINT_toks \csname XINT_for_right\the\numexpr#1+2\endcsname}%
  \let\xintifForLast\xint_secondoftwo
  \ifx #5j\expandafter\xint_firstoftwo
     \else\expandafter\xint_secondoftwo
  \fi
  {\let\xintifForLast\xint_firstoftwo
   \xintBreakForAndDo {\XINT_x \xint_gobble_i Z}}%
  \XINT_x
  \let\xintifForFirst\xint_secondoftwo\XINT_forthree_d #1{#2}%
}%
\long\def\xintForfour #1#2#3in#4#5#6%
{%
    \let\xintifForFirst\xint_firstoftwo
    \XINT_toks  {\XINT_forfour_d #2{#6}}%
    \expandafter\the\expandafter\XINT_toks #4jZ%
}%
\long\def\XINT_forfour_d #1#2#3(#4)#5%
{%
  \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
  \XINT_toks \expandafter{\romannumeral0\xintcsvtolist{ #4}}%
  \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
      \the\XINT_toks \csname XINT_for_right\the\numexpr#1+3\endcsname}%
  \let\xintifForLast\xint_secondoftwo
  \ifx #5j\expandafter\xint_firstoftwo
     \else\expandafter\xint_secondoftwo
  \fi
  {\let\xintifForLast\xint_firstoftwo
   \xintBreakForAndDo {\XINT_x \xint_gobble_i Z}}%
  \XINT_x
  \let\xintifForFirst\xint_secondoftwo\XINT_forfour_d #1{#2}%
}%
\catcode`Z 11
\catcode`j 11
```

31.27 \xintAssign, \xintAssignArray, \xintDigitsOf

\xintAssign {a}{b}..{z}\to\A\B...\Z,
\xintAssignArray {a}{b}..{z}\to\U

version 1.01 corrects an oversight in 1.0 related to the value of \escapechar at the time of using \xintAssignArray or \xintRelaxArray These macros are non-expandable.

In version 1.05a I suddenly see some incongruous \expandafter's in (what is called now) \XINT_assignarray_end_c, which I remove.

Release 1.06 modifies the macros created by \xintAssignArray to feed their argument to a \numexpr. Release 1.06a detects an incredible typo in 1.01, (bad copy-paste from \xintRelaxArray) which caused \xintAssignArray to use #1 rather than the #2 as in the correct earlier 1.0 version!!! This went through undetected because \xint_arrayname, although weird, was still usable: the probability to overwrite something was almost zero. The bug got finally revealed doing \xintAssignArray {}{}{}\to\Stuff.

With release 1.06b an empty argument (or expanding to empty) to \xintAssignArray is ok.

1.09h simplifies the coding of \xintAssignArray (no more _end_a, _end_b, etc...), and no use of a \count register anymore, and uses \xintiloop in \xintRelaxArray. Furthermore, macros are made long.

1.09i allows an optional parameter \xintAssign [oo] for example, then \oodef rather than \edef is used. Idem for \xintAssignArray. However in the latter case, the global variant is not available, one should use \globaldefs for that.

1.09j: I decide that the default behavior of \xintAssign should be to use \def, not \edef when assigning to a cs an item of the list. This is a breaking change but I don't think anybody on earth is using xint anyhow. Also use of the optional parameter was broken if it was [], [g], [e], [x] as the corresponding \XINT_... macros had not been defined (in the initial version I did not have the XINT_ prefix; then I added it in case \oodef was pre-existing and thus was not redefined by the package which instead had \XINT_oodef, now \xintoodef.)

```
\def\xintAssign{\def\XINT_flet_macro {\XINT_assign_fork}\XINT_flet_zapsp }%
\def\XINT_assign_fork
{%
    \let\XINT_assign_def\def
    \ifx\XINT_token[\expandafter\XINT_assign_opt
                   \else\expandafter\XINT_assign_a
    \fi
}%
\def\XINT_assign_opt [#1]%
{%
    \ifcsname #1def\endcsname
      \expandafter\let\expandafter\XINT_assign_def \csname #1def\endcsname
    \else
      \expandafter\let\expandafter\XINT_assign_def \csname xint#1def\endcsname
    \fi
    \XINT_assign_a
}%
\long\def\XINT_assign_a #1\to
{%
    \expandafter\XINT_assign_b\romannumeral-`0#1{}\to
}%
\long\def\XINT_assign_b #1% attention to the # at the beginning of next line
#{%
    \def\xint_temp {#1}%
    \ifx\empty\xint_temp
        \expandafter\XINT_assign_c
    \else
        \expandafter\XINT_assign_d
    \fi
}%
\long\def\XINT_assign_c #1#2\to #3%
```

```
{%
    \XINT_assign_def #3{#1}%
    \def\xint_temp {#2}%
    \unless\ifx\empty\xint_temp\xint_afterfi{\XINT_assign_b #2\to }\fi
}%
\def\XINT_assign_d #1\to #2% normally #1 is {} here.
{%
    \expandafter\XINT_assign_def\expandafter #2\expandafter{\xint_temp}%
}%
\def\xintRelaxArray #1%
{%
    \edef\XINT_restoreescapechar {\escapechar\the\escapechar\relax}%
    \escapechar -1
    \expandafter\def\expandafter\xint_arrayname\expandafter {\string #1}%
    \XINT_restoreescapechar
    \xintiloop [\csname\xint_arrayname 0\endcsname+-1]
      \global
      \expandafter\let\csname\xint_arrayname\xintiloopindex\endcsname\relax
      \ifnum \xintiloopindex > \xint_c_
    \repeat
    \global\expandafter\let\csname\xint_arrayname 00\endcsname\relax
    \global\let #1\relax
}%
\def\xintAssignArray{\def\XINT_flet_macro {\XINT_assignarray_fork}%
                     \XINT_flet_zapsp }%
\def\XINT_assignarray_fork
{%
    \let\XINT_assignarray_def\def
    \ifx\XINT_token[\expandafter\XINT_assignarray_opt
              \else\expandafter\XINT_assignarray
    \fi
}%
\def\XINT_assignarray_opt [#1]%
{%
    \ifcsname #1def\endcsname
      \expandafter\let\expandafter\XINT_assignarray_def \csname #1def\endcsname
    \else
      \expandafter\let\expandafter\XINT_assignarray_def
                              \csname xint#1def\endcsname
    \fi
    \XINT_assignarray
}%
\long\def\XINT_assignarray #1\to #2%
{%
    \edef\XINT_restoreescapechar {\escapechar\the\escapechar\relax }%
    \escapechar -1
    \expandafter\def\expandafter\xint_arrayname\expandafter {\string #2}%
    \XINT_restoreescapechar
    \def\xint_itemcount {0}%
```

```
    \expandafter\XINT_assignarray_loop \romannumeral-`0#1\xint_relax
    \csname\xint_arrayname 00\expandafter\endcsname
    \csname\xint_arrayname 0\expandafter\endcsname
    \expandafter {\xint_arrayname}#2%
}%
\long\def\XINT_assignarray_loop #1%
{%
    \def\xint_temp {#1}%
    \ifx\xint_brelax\xint_temp
       \expandafter\def\csname\xint_arrayname 0\expandafter\endcsname
                  \expandafter{\the\numexpr\xint_itemcount}%
       \expandafter\expandafter\expandafter\XINT_assignarray_end
    \else
       \expandafter\def\expandafter\xint_itemcount\expandafter
                  {\the\numexpr\xint_itemcount+\xint_c_i}%
       \expandafter\XINT_assignarray_def
          \csname\xint_arrayname\xint_itemcount\expandafter\endcsname
             \expandafter{\xint_temp }%
       \expandafter\XINT_assignarray_loop
    \fi
}%
\def\XINT_assignarray_end #1#2#3#4%
{%
    \def #4##1%
    {%
        \romannumeral0\expandafter #1\expandafter{\the\numexpr ##1}%
    }%
    \def #1##1%
    {%
        \ifnum ##1<\xint_c_
            \xint_afterfi {\xintError:ArrayIndexIsNegative\space }%
        \else
        \xint_afterfi {%
               \ifnum ##1>#2
                   \xint_afterfi {\xintError:ArrayIndexBeyondLimit\space }%
               \else\xint_afterfi
       {\expandafter\expandafter\expandafter\space\csname #3##1\endcsname}%
               \fi}%
        \fi
     }%
}%
\let\xintDigitsOf\xintAssignArray
\let\XINT_tmpa\relax \let\XINT_tmpb\relax \let\XINT_tmpc\relax
\XINT_restorecatcodes_endinput%
```

32 Package xint implementation

With release 1.09a all macros doing arithmetic operations and a few more apply systematically \xintnum to their arguments; this adds a little overhead but this is more convenient for using count registers even with infix notation; also this is what xintfrac.sty did all along. Simplifies the discussion in the documentation too.

Contents

32.1 Catcodes, ε-TEX and reload detection

The code for reload detection is copied from Heiko Oberdiek's packages, and adapted here to check for previous loading of the master **xint** package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \def\space { }%
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xint.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xinttools.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xint}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xint.sty
      \ifx\w\relax % but xinttools.sty not yet loaded.
         \y{xint}{now issuing \string\input\space xinttools.sty}%
         \def\z{\endgroup\input xinttools.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xinttools.sty not yet loaded.
            \y{xint}{now issuing \string\RequirePackage{xinttools}}%
            \def\z{\endgroup\RequirePackage{xinttools}}%
          \fi
```

```
      \else
        \y{xint}{I was already loaded, aborting input}%
        \aftergroup\endinput
      \fi
    \fi
  \fi
\z%
```

32.2 Confirmation of xinttools loading

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \ifdefined\PackageInfo
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \else
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
  \fi
  \def\empty {}%
  \expandafter\let\expandafter\w\csname ver@xinttools.sty\endcsname
  \ifx\w\relax % Plain TeX, user gave a file name at the prompt
      \y{xint}{Loading of package xinttools failed, aborting input}%
      \aftergroup\endinput
  \fi
  \ifx\w\empty % LaTeX, user gave a file name at the prompt
      \y{xint}{Loading of package xinttools failed, aborting input}%
      \aftergroup\endinput
  \fi
\endgroup%
```

32.3 Catcodes

```
\XINTsetupcatcodes%
```

32.4 Package identification

```
\XINT_providespackage
\ProvidesPackage{xint}%
  [2014/02/13 v1.09kb Expandable operations on long numbers (jfB)]%
```

32.5 Token management, constants

```
\long\def\xint_firstofthree  #1#2#3{#1}%
```

```
\long\def\xint_secondofthree #1#2#3{#2}%
\long\def\xint_thirdofthree  #1#2#3{#3}%
\long\def\xint_firstofthree_thenstop  #1#2#3{ #1}% 1.09i
\long\def\xint_secondofthree_thenstop #1#2#3{ #2}%
\long\def\xint_thirdofthree_thenstop  #1#2#3{ #3}%
\def\xint_gob_til_zero  #10{}%
\def\xint_gob_til_zeros_iii #1000{}%
\def\xint_gob_til_zeros_iv  #10000{}%
\def\xint_gob_til_one   #11{}%
\def\xint_gob_til_G     #1G{}%
\def\xint_gob_til_minus #1-{}%
\def\xint_gob_til_relax #1\relax {}%
\def\xint_exchangetwo_keepbraces          #1#2{{#2}{#1}}%
\def\xint_exchangetwo_keepbraces_thenstop #1#2{ {#2}{#1}}%
\def\xint_UDzerofork      #10#2#3\krof {#2}%
\def\xint_UDsignfork      #1-#2#3\krof {#2}%
\def\xint_UDwfork        #1\W#2#3\krof {#2}%
\def\xint_UDzerosfork    #100#2#3\krof {#2}%
\def\xint_UDonezerofork  #110#2#3\krof {#2}%
\def\xint_UDzerominusfork #10-#2#3\krof {#2}%
\def\xint_UDsignsfork    #1--#2#3\krof {#2}%
% \chardef\xint_c_   0 % already done in xinttools
% \chardef\xint_c_i  1 % already done in xinttools
\chardef\xint_c_ii   2
\chardef\xint_c_iii  3
\chardef\xint_c_iv   4
\chardef\xint_c_v    5
% \chardef\xint_c_vi   6 % will be done in xintfrac
% \chardef\xinf_c_vii  7 % will be done in xintfrac
% \chardef\xint_c_viii 8 % already done in xinttools
\chardef\xint_c_ix     9
\chardef\xint_c_x     10
\chardef\xint_c_ii^v  32 % not used in xint, common to xintfrac and xintbinhex
\chardef\xint_c_ii^vi 64
\mathchardef\xint_c_ixixixix 9999
\mathchardef\xint_c_x^iv    10000
\newcount\xint_c_x^viii  \xint_c_x^viii 100000000
```

32.6 \xintRev

\xintRev: expands fully its argument \romannumeral-`0, and checks the sign. However this last aspect does not appear like a very useful thing. And despite the fact that a special check is made for a sign, actually the input is not given to \xintnum, contrarily to \xintLen. This is all a bit incoherent. Should be fixed.

```
\def\xintRev {\romannumeral0\xintrev }%
\def\xintrev #1%
{%
    \expandafter\XINT_rev_fork
    \romannumeral-`0#1\xint_relax % empty #1 ok, \xint_relax stops expansion
```

```
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
    \xint_relax
}%
\def\XINT_rev_fork #1%
{%
    \xint_UDsignfork
    #1{\expandafter\xint_minus_thenstop\romannumeral0\XINT_rord_main {}}%
     -{\XINT_rord_main {}#1}%
    \krof
}%
```

32.7 \xintLen

\xintLen is ONLY for (possibly long) integers. Gets extended to fractions by xint-frac.sty

```
\def\xintLen {\romannumeral0\xintlen }%
\def\xintlen #1%
{%
    \expandafter\XINT_len_fork
    \romannumeral0\xintnum{#1}\xint_relax\xint_relax\xint_relax\xint_relax
                      \xint_relax\xint_relax\xint_relax\xint_relax\xint_bye
}%
\def\XINT_Len #1% variant which does not expand via \xintnum.
{%
    \romannumeral0\XINT_len_fork
    #1\xint_relax\xint_relax\xint_relax\xint_relax
      \xint_relax\xint_relax\xint_relax\xint_relax\xint_bye
}%
\def\XINT_len_fork #1%
{%
    \expandafter\XINT_length_loop
    \xint_UDsignfork
      #1{{0}}%
       -{{0}#1}%
    \krof
}%
```

32.8 \XINT_RQ

cette macro renverse et ajoute le nombre minimal de zéros à la fin pour que la longueur soit alors multiple de 4
\romannumeral0\XINT_RQ {}<le truc à renverser>\R\R\R\R\R\R\R\R\Z
Attention, ceci n'est utilisé que pour des chaînes de chiffres, et donc le comportement avec des {..} ou autres espaces n'a fait l'objet d'aucune attention

```
\def\XINT_RQ #1#2#3#4#5#6#7#8#9%
```

```
{%
    \xint_gob_til_R #9\XINT_RQ_end_a\R\XINT_RQ {#9#8#7#6#5#4#3#2#1}%
}%
\def\XINT_RQ_end_a\R\XINT_RQ #1#2\Z
{%
    \XINT_RQ_end_b #1\Z
}%
\def\XINT_RQ_end_b #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_R
            #8\XINT_RQ_end_viii
            #7\XINT_RQ_end_vii
            #6\XINT_RQ_end_vi
            #5\XINT_RQ_end_v
            #4\XINT_RQ_end_iv
            #3\XINT_RQ_end_iii
            #2\XINT_RQ_end_ii
            \R\XINT_RQ_end_i
            \Z #2#3#4#5#6#7#8%
}%
\def\XINT_RQ_end_viii #1\Z #2#3#4#5#6#7#8#9\Z { #9}%
\def\XINT_RQ_end_vii  #1\Z #2#3#4#5#6#7#8#9\Z { #8#9000}%
\def\XINT_RQ_end_vi   #1\Z #2#3#4#5#6#7#8#9\Z { #7#8#900}%
\def\XINT_RQ_end_v    #1\Z #2#3#4#5#6#7#8#9\Z { #6#7#8#90}%
\def\XINT_RQ_end_iv   #1\Z #2#3#4#5#6#7#8#9\Z { #5#6#7#8#9}%
\def\XINT_RQ_end_iii  #1\Z #2#3#4#5#6#7#8#9\Z { #4#5#6#7#8#9000}%
\def\XINT_RQ_end_ii   #1\Z #2#3#4#5#6#7#8#9\Z { #3#4#5#6#7#8#900}%
\def\XINT_RQ_end_i      \Z #1#2#3#4#5#6#7#8\Z { #1#2#3#4#5#6#7#80}%
\def\XINT_SQ #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_R #8\XINT_SQ_end_a\R\XINT_SQ {#8#7#6#5#4#3#2#1}%
}%
\def\XINT_SQ_end_a\R\XINT_SQ #1#2\Z
{%
    \XINT_SQ_end_b #1\Z
}%
\def\XINT_SQ_end_b #1#2#3#4#5#6#7%
{%
    \xint_gob_til_R
            #7\XINT_SQ_end_vii
            #6\XINT_SQ_end_vi
            #5\XINT_SQ_end_v
            #4\XINT_SQ_end_iv
            #3\XINT_SQ_end_iii
            #2\XINT_SQ_end_ii
            \R\XINT_SQ_end_i
            \Z #2#3#4#5#6#7%
}%
\def\XINT_SQ_end_vii  #1\Z #2#3#4#5#6#7#8\Z { #8}%
```

```
\def\XINT_SQ_end_vi   #1\Z #2#3#4#5#6#7#8\Z { #7#8000000}%
\def\XINT_SQ_end_v    #1\Z #2#3#4#5#6#7#8\Z { #6#7#800000}%
\def\XINT_SQ_end_iv   #1\Z #2#3#4#5#6#7#8\Z { #5#6#7#80000}%
\def\XINT_SQ_end_iii  #1\Z #2#3#4#5#6#7#8\Z { #4#5#6#7#8000}%
\def\XINT_SQ_end_ii   #1\Z #2#3#4#5#6#7#8\Z { #3#4#5#6#7#800}%
\def\XINT_SQ_end_i      \Z #1#2#3#4#5#6#7\Z { #1#2#3#4#5#6#70}%
\def\XINT_OQ #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_R #9\XINT_OQ_end_a\R\XINT_OQ {#9#8#7#6#5#4#3#2#1}%
}%
\def\XINT_OQ_end_a\R\XINT_OQ #1#2\Z
{%
    \XINT_OQ_end_b #1\Z
}%
\def\XINT_OQ_end_b #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_R
            #8\XINT_OQ_end_viii
            #7\XINT_OQ_end_vii
            #6\XINT_OQ_end_vi
            #5\XINT_OQ_end_v
            #4\XINT_OQ_end_iv
            #3\XINT_OQ_end_iii
            #2\XINT_OQ_end_ii
            \R\XINT_OQ_end_i
            \Z #2#3#4#5#6#7#8%
}%
\def\XINT_OQ_end_viii #1\Z #2#3#4#5#6#7#8#9\Z { #9}%
\def\XINT_OQ_end_vii  #1\Z #2#3#4#5#6#7#8#9\Z { #8#90000000}%
\def\XINT_OQ_end_vi   #1\Z #2#3#4#5#6#7#8#9\Z { #7#8#9000000}%
\def\XINT_OQ_end_v    #1\Z #2#3#4#5#6#7#8#9\Z { #6#7#8#900000}%
\def\XINT_OQ_end_iv   #1\Z #2#3#4#5#6#7#8#9\Z { #5#6#7#8#90000}%
\def\XINT_OQ_end_iii  #1\Z #2#3#4#5#6#7#8#9\Z { #4#5#6#7#8#9000}%
\def\XINT_OQ_end_ii   #1\Z #2#3#4#5#6#7#8#9\Z { #3#4#5#6#7#8#900}%
\def\XINT_OQ_end_i      \Z #1#2#3#4#5#6#7#8\Z { #1#2#3#4#5#6#7#80}%
```

32.9 \XINT_cuz

```
\edef\xint_cleanupzeros_andstop #1#2#3#4%
{%
    \noexpand\expandafter\space\noexpand\the\numexpr #1#2#3#4\relax
}%
\def\xint_cleanupzeros_nostop #1#2#3#4%
{%
    \the\numexpr #1#2#3#4\relax
}%
\def\XINT_rev_andcuz #1%
{%
    \expandafter\xint_cleanupzeros_andstop
```

```
    \romannumeral0\XINT_rord_main {}#1%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax
}%
```

routine CleanUpZeros. Utilisée en particulier par la soustraction.
INPUT: longueur **multiple de 4** (<-- ATTENTION)
OUTPUT: on a retiré tous les leading zéros, on n'est **plus* nécessairement de longueur 4n
Délimiteur pour _main: \W\W\W\W\W\W\W\Z avec SEPT \W

```
\def\XINT_cuz #1%
{%
    \XINT_cuz_loop #1\W\W\W\W\W\W\W\Z%
}%
\def\XINT_cuz_loop #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_W #8\xint_cuz_end_a\W
    \xint_gob_til_Z #8\xint_cuz_end_A\Z
    \XINT_cuz_check_a {#1#2#3#4#5#6#7#8}%
}%
\def\xint_cuz_end_a #1\XINT_cuz_check_a #2%
{%
    \xint_cuz_end_b #2%
}%
\edef\xint_cuz_end_b #1#2#3#4#5\Z
{%
    \noexpand\expandafter\space\noexpand\the\numexpr #1#2#3#4\relax
}%
\def\xint_cuz_end_A \Z\XINT_cuz_check_a #1{ 0}%
\def\XINT_cuz_check_a #1%
{%
    \expandafter\XINT_cuz_check_b\the\numexpr #1\relax
}%
\def\XINT_cuz_check_b #1%
{%
    \xint_gob_til_zero #1\xint_cuz_backtoloop 0\XINT_cuz_stop #1%
}%
\def\XINT_cuz_stop #1\W #2\Z{ #1}%
\def\xint_cuz_backtoloop 0\XINT_cuz_stop 0{\XINT_cuz_loop }%
```

32.10 \xintIsOne

Added in 1.03. Attention: \XINT_isOne does not do any expansion. Release 1.09a defines \xintIsOne which is more user-friendly. Will be modified if xintfrac is loaded.

```
\def\xintIsOne   {\romannumeral0\xintisone }%
```

```
\def\xintisone #1{\expandafter\XINT_isone\romannumeral0\xintnum{#1}\W\Z }%
\def\XINT_isOne #1{\romannumeral0\XINT_isone #1\W\Z }%
\def\XINT_isone #1#2%
{%
    \xint_gob_til_one #1\XINT_isone_b 1%
    \expandafter\space\expandafter 0\xint_gob_til_Z #2%
}%
\def\XINT_isone_b #1\xint_gob_til_Z #2%
{%
    \xint_gob_til_W #2\XINT_isone_yes \W
    \expandafter\space\expandafter 0\xint_gob_til_Z
}%
\def\XINT_isone_yes #1\Z { 1}%
```

32.11 \xintNum

For example \xintNum {----+-+++---+----000000000000003}
1.05 defines \xintiNum, which allows redefinition of \xintNum by xintfrac.sty Slightly modified in 1.06b (\R->\xint_relax) to avoid initial re-scan of input stack (while still allowing empty #1). In versions earlier than 1.09a it was entirely up to the user to apply \xintnum; starting with 1.09a arithmetic macros of xint.sty (like earlier already xintfrac.sty with its own \xintnum) make use of \xintnum. This allows arguments to be count registers, or even \numexpr arbitrary long expressions (with the trick of braces, see the user documentation).

```
\def\xintiNum {\romannumeral0\xintinum }%
\def\xintinum #1%
{%
    \expandafter\XINT_num_loop
    \romannumeral-`0#1\xint_relax\xint_relax\xint_relax\xint_relax
                      \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\let\xintNum\xintiNum \let\xintnum\xintinum
\def\XINT_num #1%
{%
    \XINT_num_loop #1\xint_relax\xint_relax\xint_relax\xint_relax
                     \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\def\XINT_num_loop #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_xint_relax #8\XINT_num_end\xint_relax
    \XINT_num_NumEight #1#2#3#4#5#6#7#8%
}%
\edef\XINT_num_end\xint_relax\XINT_num_NumEight #1\xint_relax #2\Z
{%
    \noexpand\expandafter\space\noexpand\the\numexpr #1+0\relax
}%
\def\XINT_num_NumEight #1#2#3#4#5#6#7#8%
{%
```

```
    \ifnum \numexpr #1#2#3#4#5#6#7#8+0= 0
      \xint_afterfi {\expandafter\XINT_num_keepsign_a
                     \the\numexpr #1#2#3#4#5#6#7#81\relax}%
    \else
      \xint_afterfi {\expandafter\XINT_num_finish
                     \the\numexpr #1#2#3#4#5#6#7#8\relax}%
    \fi
}%
\def\XINT_num_keepsign_a #1%
{%
    \xint_gob_til_one#1\XINT_num_gobacktoloop 1\XINT_num_keepsign_b
}%
\def\XINT_num_gobacktoloop 1\XINT_num_keepsign_b {\XINT_num_loop }%
\def\XINT_num_keepsign_b #1{\XINT_num_loop -}%
\def\XINT_num_finish #1\xint_relax #2\Z { #1}%
```

32.12 \xintSgn, \xintiiSgn, \XINT_Sgn, \XINT_cntSgn

Changed in 1.05. Earlier code was unnecessarily strange. 1.09a with \xintnum
  1.09i defines \XINT_Sgn and \XINT_cntSgn (was \XINT__Sgn in 1.09i) for reasons
of internal optimizations

```
\def\xintiiSgn {\romannumeral0\xintiisgn }%
\def\xintiisgn #1%
{%
    \expandafter\XINT_sgn \romannumeral-`0#1\Z%
}%
\def\xintSgn {\romannumeral0\xintsgn }%
\def\xintsgn #1%
{%
    \expandafter\XINT_sgn \romannumeral0\xintnum{#1}\Z%
}%
\def\XINT_sgn #1#2\Z
{%
    \xint_UDzerominusfork
      #1-{ 0}%
      0#1{ -1}%
       0-{ 1}%
    \krof
}%
\def\XINT_Sgn #1#2\Z
{%
    \xint_UDzerominusfork
      #1-{0}%
      0#1{-1}%
       0-{1}%
    \krof
}%
\def\XINT_cntSgn #1#2\Z
```

```
{%
    \xint_UDzerominusfork
      #1-\z@
      0#1\m@ne
       0-\@ne
    \krof
}%
```

32.13 \xintBool, \xintToggle

1.09c

```
\def\xintBool #1{\romannumeral-‘0%
                 \csname if#1\endcsname\expandafter1\else\expandafter0\fi }%
\def\xintToggle #1{\romannumeral-‘0\iftoggle{#1}{1}{0}}%
```

32.14 \xintSgnFork

Expandable three-way fork added in 1.07. The argument #1 must expand to -1,0 or 1. 1.09i has _afterstop, renamed _thenstop later, for efficiency.

```
\def\xintSgnFork {\romannumeral0\xintsgnfork }%
\def\xintsgnfork #1%
{%
    \ifcase #1 \expandafter\xint_secondofthree_thenstop
            \or\expandafter\xint_thirdofthree_thenstop
          \else\expandafter\xint_firstofthree_thenstop
    \fi
}%
```

32.15 \XINT_cntSgnFork

1.09i. Used internally, #1 must expand to \m@ne, \z@, or \@ne or equivalent. Does not insert a space token to stop a romannumeral0 expansion.

```
\def\XINT_cntSgnFork #1%
{%
    \ifcase #1\expandafter\xint_secondofthree
            \or\expandafter\xint_thirdofthree
          \else\expandafter\xint_firstofthree
    \fi
}%
```

32.16 \xintifSgn

Expandable three-way fork added in 1.09a. Branches expandably depending on whether <0, =0, >0. Choice of branch guaranteed in two steps.

The use of \romannumeral0\xintsgn rather than \xintSgn is for matters related to the transformation of the ternary operator : in \xintNewExpr. I hope I have explained there the details because right now off hand I can't recall why.

1.09i has \xint_firstofthreeafterstop (now _thenstop) etc for faster expansion.

```
\def\xintifSgn {\romannumeral0\xintifsgn }%
\def\xintifsgn #1%
{%
    \ifcase \romannumeral0\xintsgn{#1}
               \expandafter\xint_secondofthree_thenstop
            \or\expandafter\xint_thirdofthree_thenstop
          \else\expandafter\xint_firstofthree_thenstop
    \fi
}%
```

32.17 \xintifZero, \xintifNotZero

Expandable two-way fork added in 1.09a. Branches expandably depending on whether the argument is zero (branch A) or not (branch B). 1.09i restyling. By the way it appears (not thoroughly tested, though) that \if tests are faster than \ifnum tests.

```
\def\xintifZero {\romannumeral0\xintifzero }%
\def\xintifzero #1%
{%
    \if0\xintSgn{#1}%
       \expandafter\xint_firstoftwo_thenstop
    \else
       \expandafter\xint_secondoftwo_thenstop
    \fi
}%
\def\xintifNotZero {\romannumeral0\xintifnotzero }%
\def\xintifnotzero #1%
{%
    \if0\xintSgn{#1}%
       \expandafter\xint_secondoftwo_thenstop
    \else
       \expandafter\xint_firstoftwo_thenstop
    \fi
}%
```

32.18 \xintifOne

added in 1.09i.

```
\def\xintifOne {\romannumeral0\xintifone }%
\def\xintifone #1%
```

```
{%
    \if1\xintIsOne{#1}%
       \expandafter\xint_firstoftwo_thenstop
    \else
       \expandafter\xint_secondoftwo_thenstop
    \fi
}%
```

32.19 \xintifTrueAelseB, \xintifFalseAelseB

1.09i. Warning, \xintifTrueFalse, \xintifTrue deprecated, to be removed

```
\let\xintifTrueAelseB\xintifNotZero
\let\xintifFalseAelseB\xintifZero
\let\xintifTrue\xintifNotZero
\let\xintifTrueFalse\xintifNotZero
```

32.20 \xintifCmp

1.09e \xintifCmp {n}{m}{if n<m}{if n=m}{if n>m}.

```
\def\xintifCmp {\romannumeral0\xintifcmp }%
\def\xintifcmp #1#2%
{%
    \ifcase\xintCmp {#1}{#2}
              \expandafter\xint_secondofthree_thenstop
           \or\expandafter\xint_thirdofthree_thenstop
         \else\expandafter\xint_firstofthree_thenstop
    \fi
}%
```

32.21 \xintifEq

1.09a \xintifEq {n}{m}{YES if n=m}{NO if n<>m}.

```
\def\xintifEq {\romannumeral0\xintifeq }%
\def\xintifeq #1#2%
{%
    \if0\xintCmp{#1}{#2}%
              \expandafter\xint_firstoftwo_thenstop
         \else\expandafter\xint_secondoftwo_thenstop
    \fi
}%
```

32.22 \xintifGt

1.09a \xintifGt {n}{m}{YES if n>m}{NO if n<=m}.

```
\def\xintifGt {\romannumeral0\xintifgt }%
\def\xintifgt #1#2%
{%
    \if1\xintCmp{#1}{#2}%
                \expandafter\xint_firstoftwo_thenstop
          \else\expandafter\xint_secondoftwo_thenstop
    \fi
}%
```

32.23 \xintifLt

1.09a \xintifLt {n}{m}{YES if n<m}{NO if n>=m}. Restyled in 1.09i

```
\def\xintifLt {\romannumeral0\xintiflt }%
\def\xintiflt #1#2%
{%
    \ifnum\xintCmp{#1}{#2}<\xint_c_
          \expandafter\xint_firstoftwo_thenstop
    \else \expandafter\xint_secondoftwo_thenstop
    \fi
}%
```

32.24 \xintifOdd

1.09e. Restyled in 1.09i.

```
\def\xintifOdd {\romannumeral0\xintifodd }%
\def\xintifodd #1%
{%
    \if\xintOdd{#1}1%
       \expandafter\xint_firstoftwo_thenstop
    \else
       \expandafter\xint_secondoftwo_thenstop
    \fi
}%
```

32.25 \xintOpp

\xintnum added in 1.09a

```
\def\xintiiOpp {\romannumeral0\xintiiopp }%
\def\xintiiopp #1%
{%
    \expandafter\XINT_opp \romannumeral-`0#1%
}%
\def\xintiOpp {\romannumeral0\xintiopp }%
\def\xintiopp #1%
{%
```

```
    \expandafter\XINT_opp \romannumeral0\xintnum{#1}%
}%
\let\xintOpp\xintiOpp \let\xintopp\xintiopp
\def\XINT_Opp #1{\romannumeral0\XINT_opp #1}%
\def\XINT_opp #1%
{%
    \xint_UDzerominusfork
      #1-{ 0}%      zero
      0#1{ }%     negative
       0-{ -#1}%  positive
    \krof
}%
```

32.26 \xintAbs

Release 1.09a has now \xintiabs which does \xintnum (contrarily to some other i-macros, but similarly as \xintiAdd etc...) and this is inherited by DecSplit, by Sqr, and macros of xintgcd.sty.

```
\def\xintiiAbs {\romannumeral0\xintiiabs }%
\def\xintiiabs #1%
{%
    \expandafter\XINT_abs \romannumeral-`0#1%
}%
\def\xintiAbs {\romannumeral0\xintiabs }%
\def\xintiabs #1%
{%
    \expandafter\XINT_abs \romannumeral0\xintnum{#1}%
}%
\let\xintAbs\xintiAbs \let\xintabs\xintiabs
\def\XINT_Abs #1{\romannumeral0\XINT_abs #1}%
\def\XINT_abs #1%
{%
    \xint_UDsignfork
      #1{ }%
       -{ #1}%
    \krof
}%
```


ARITHMETIC OPERATIONS: ADDITION, SUBTRACTION, SUMS, MULTIPLICATION, PRODUCTS, FACTORIAL, POWERS, EUCLIDEAN DIVISION.

Release 1.03 re-organizes sub-routines to facilitate future developments: the diverse variants of addition, with diverse conditions on inputs and output are first listed; they will be used in multiplication, or in the summation, or in the power routines. I am aware that the commenting is close to non-existent, sorry about that.

ADDITION I: \XINT_add_A

INPUT:
\romannumeral0\XINT_add_A 0{}<N1>\W\X\Y\Z <N2>\W\X\Y\Z
1. <N1> et <N2> renversés
2. de longueur 4n (avec des leading zéros éventuels)
3. l'un des deux ne doit pas se terminer par 0000
[Donc on peut avoir 0000 comme input si l'autre est >0 et ne se termine pas en 0000 bien sûr]. On peut avoir l'un des deux vides. Mais alors l'autre ne doit être ni vide ni 0000.

OUTPUT: la somme <N1>+<N2>, ordre normal, plus sur 4n, pas de leading zeros La procédure est plus rapide lorsque <N1> est le plus court des deux.
Nota bene: (30 avril 2013). J'ai une version qui est deux fois plus rapide sur des nombres d'environ 1000 chiffres chacun, et qui commence à être avantageuse pour des nombres d'au moins 200 chiffres. Cependant il serait vraiment compliqué d'en étendre l'utilisation aux emplois de l'addition dans les autres routines, comme celle de multiplication ou celle de division; et son implémentation ajouterait au minimum la mesure de la longueur des summands.

```
\def\XINT_add_A #1#2#3#4#5#6%
{%
    \xint_gob_til_W #3\xint_add_az\W
    \XINT_add_AB #1{#3#4#5#6}{#2}%
}%
\def\xint_add_az\W\XINT_add_AB #1#2%
{%
    \XINT_add_AC_checkcarry #1%
}%
```

ici #2 est prévu pour l'addition, mais attention il devra être renversé pour \numexpr. #3 = résultat partiel. #4 = chiffres qui restent. On vérifie si le deuxième nombre s'arrête.

```
\def\XINT_add_AB #1#2#3#4\W\X\Y\Z #5#6#7#8%
{%
    \xint_gob_til_W #5\xint_add_bz\W
    \XINT_add_ABE #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
}%
\def\XINT_add_ABE #1#2#3#4#5#6%
{%
    \expandafter\XINT_add_ABEA\the\numexpr #1+10#5#4#3#2+#6.%
}%
\def\XINT_add_ABEA #1#2#3.#4%
{%
    \XINT_add_A  #2{#3#4}%
}%
```

ici le deuxième nombre est fini #6 part à la poubelle, #2#3#4#5 est le #2 dans \XINT_add_AB on ne vérifie pas la retenue cette fois, mais les fois suivantes

```
\def\xint_add_bz\W\XINT_add_ABE #1#2#3#4#5#6%
{%
```

```
    \expandafter\XINT_add_CC\the\numexpr #1+10#5#4#3#2.%
}%
\def\XINT_add_CC #1#2#3.#4%
{%
    \XINT_add_AC_checkcarry  #2{#3#4}% on va examiner et \'eliminer #2
}%
```

retenue plus chiffres qui restent de l'un des deux nombres. #2 = résultat partiel
#3#4#5#6 = summand, avec plus significatif à droite

```
\def\XINT_add_AC_checkcarry #1%
{%
    \xint_gob_til_zero #1\xint_add_AC_nocarry 0\XINT_add_C
}%
\def\xint_add_AC_nocarry 0\XINT_add_C #1#2\W\X\Y\Z
{%
    \expandafter
    \xint_cleanupzeros_andstop
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax
    #1%
}%
\def\XINT_add_C #1#2#3#4#5%
{%
    \xint_gob_til_W #2\xint_add_cz\W
    \XINT_add_CD {#5#4#3#2}{#1}%
}%
\def\XINT_add_CD #1%
{%
    \expandafter\XINT_add_CC\the\numexpr 1+10#1.%
}%
\def\xint_add_cz\W\XINT_add_CD #1#2{ 1#2}%
```

Addition II: \XINT_addr_A.
INPUT: \romannumeral0\XINT_addr_A 0{}<N1>\W\X\Y\Z <N2>\W\X\Y\Z

Comme \XINT_add_A, la différence principale c'est qu'elle donne son résultat aussi *sur 4n*, renversé. De plus cette variante accepte que l'un ou même les deux inputs soient vides. Utilisé par la sommation et par la division (pour les quotients). Et aussi par la multiplication d'ailleurs.

INPUT: comme pour \XINT_add_A
1. <N1> et <N2> renversés
2. de longueur 4n (avec des leading zéros éventuels)
3. l'un des deux ne doit pas se terminer par 0000

OUTPUT: la somme <N1>+<N2>, *aussi renversée* et *sur 4n*

```
\def\XINT_addr_A #1#2#3#4#5#6%
```

```
{%
    \xint_gob_til_W #3\xint_addr_az\W
    \XINT_addr_B #1{#3#4#5#6}{#2}%
}%
\def\xint_addr_az\W\XINT_addr_B #1#2%
{%
    \XINT_addr_AC_checkcarry #1%
}%
\def\XINT_addr_B #1#2#3#4\W\X\Y\Z #5#6#7#8%
{%
    \xint_gob_til_W #5\xint_addr_bz\W
    \XINT_addr_E #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
}%
\def\XINT_addr_E #1#2#3#4#5#6%
{%
    \expandafter\XINT_addr_ABEA\the\numexpr #1+10#5#4#3#2+#6\relax
}%
\def\XINT_addr_ABEA #1#2#3#4#5#6#7%
{%
    \XINT_addr_A  #2{#7#6#5#4#3}%
}%
\def\xint_addr_bz\W\XINT_addr_E #1#2#3#4#5#6%
{%
    \expandafter\XINT_addr_CC\the\numexpr #1+10#5#4#3#2\relax
}%
\def\XINT_addr_CC #1#2#3#4#5#6#7%
{%
    \XINT_addr_AC_checkcarry  #2{#7#6#5#4#3}%
}%
\def\XINT_addr_AC_checkcarry #1%
{%
    \xint_gob_til_zero #1\xint_addr_AC_nocarry 0\XINT_addr_C
}%
\def\xint_addr_AC_nocarry 0\XINT_addr_C #1#2\W\X\Y\Z { #1#2}%
\def\XINT_addr_C #1#2#3#4#5%
{%
    \xint_gob_til_W #2\xint_addr_cz\W
    \XINT_addr_D {#5#4#3#2}{#1}%
}%
\def\XINT_addr_D #1%
{%
    \expandafter\XINT_addr_CC\the\numexpr 1+10#1\relax
}%
\def\xint_addr_cz\W\XINT_addr_D #1#2{ #21000}%
```

```
ADDITION III, \XINT_addm_A
INPUT:\romannumeral0\XINT_addm_A 0{}<N1>\W\X\Y\Z <N2>\W\X\Y\Z
1. <N1> et <N2> renversés
2. <N1> de longueur 4n ; <N2> non
```

3. <N2> est *garanti au moins aussi long* que <N1>
OUTPUT: la somme <N1>+<N2>, ordre normal, pas sur 4n, leading zeros retirés. Utilisé par la multiplication.

```
\def\XINT_addm_A #1#2#3#4#5#6%
{%
    \xint_gob_til_W #3\xint_addm_az\W
    \XINT_addm_AB #1{#3#4#5#6}{#2}%
}%
\def\xint_addm_az\W\XINT_addm_AB #1#2%
{%
    \XINT_addm_AC_checkcarry #1%
}%
\def\XINT_addm_AB #1#2#3#4\W\X\Y\Z #5#6#7#8%
{%
    \XINT_addm_ABE #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
}%
\def\XINT_addm_ABE #1#2#3#4#5#6%
{%
    \expandafter\XINT_addm_ABEA\the\numexpr #1+10#5#4#3#2+#6.%
}%
\def\XINT_addm_ABEA #1#2#3.#4%
{%
    \XINT_addm_A  #2{#3#4}%
}%
\def\XINT_addm_AC_checkcarry #1%
{%
    \xint_gob_til_zero #1\xint_addm_AC_nocarry 0\XINT_addm_C
}%
\def\xint_addm_AC_nocarry 0\XINT_addm_C #1#2\W\X\Y\Z
{%
    \expandafter
    \xint_cleanupzeros_andstop
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax
    #1%
}%
\def\XINT_addm_C #1#2#3#4#5%
{%
    \xint_gob_til_W
    #5\xint_addm_cw
    #4\xint_addm_cx
    #3\xint_addm_cy
    #2\xint_addm_cz
    \W\XINT_addm_CD {#5#4#3#2}{#1}%
```

```
}%
\def\XINT_addm_CD #1%
{%
    \expandafter\XINT_addm_CC\the\numexpr 1+10#1.%
}%
\def\XINT_addm_CC #1#2#3.#4%
{%
    \XINT_addm_AC_checkcarry  #2{#3#4}%
}%
\def\xint_addm_cw
    #1\xint_addm_cx
    #2\xint_addm_cy
    #3\xint_addm_cz
    \W\XINT_addm_CD
{%
    \expandafter\XINT_addm_CDw\the\numexpr 1+#1#2#3.%
}%
\def\XINT_addm_CDw #1.#2#3\X\Y\Z
{%
    \XINT_addm_end #1#3%
}%
\def\xint_addm_cx
    #1\xint_addm_cy
    #2\xint_addm_cz
    \W\XINT_addm_CD
{%
    \expandafter\XINT_addm_CDx\the\numexpr 1+#1#2.%
}%
\def\XINT_addm_CDx #1.#2#3\Y\Z
{%
    \XINT_addm_end #1#3%
}%
\def\xint_addm_cy
    #1\xint_addm_cz
    \W\XINT_addm_CD
{%
    \expandafter\XINT_addm_CDy\the\numexpr 1+#1.%
}%
\def\XINT_addm_CDy  #1.#2#3\Z
{%
    \XINT_addm_end #1#3%
}%
\def\xint_addm_cz\W\XINT_addm_CD #1#2#3{\XINT_addm_end #1#3}%
\edef\XINT_addm_end #1#2#3#4#5%
    {\noexpand\expandafter\space\noexpand\the\numexpr #1#2#3#4#5\relax}%
```

```
ADDITION IV, variante \XINT_addp_A
INPUT: \romannumeral0\XINT_addp_A 0{}<N1>\W\X\Y\Z <N2>\W\X\Y\Z
1. <N1> et <N2> renversés
```

2. <N1> de longueur 4n ; <N2> non
3. <N2> est *garanti au moins aussi long* que <N1>
OUTPUT: la somme <N1>+<N2>, dans l'ordre renversé, sur 4n, et en faisant attention de ne pas terminer en 0000. Utilisé par la multiplication servant pour le calcul des puissances.

```
\def\XINT_addp_A #1#2#3#4#5#6%
{%
    \xint_gob_til_W #3\xint_addp_az\W
    \XINT_addp_AB #1{#3#4#5#6}{#2}%
}%
\def\xint_addp_az\W\XINT_addp_AB #1#2%
{%
    \XINT_addp_AC_checkcarry #1%
}%
\def\XINT_addp_AC_checkcarry #1%
{%
    \xint_gob_til_zero #1\xint_addp_AC_nocarry 0\XINT_addp_C
}%
\def\xint_addp_AC_nocarry 0\XINT_addp_C
{%
    \XINT_addp_F
}%
\def\XINT_addp_AB #1#2#3#4\W\X\Y\Z #5#6#7#8%
{%
    \XINT_addp_ABE #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
}%
\def\XINT_addp_ABE #1#2#3#4#5#6%
{%
    \expandafter\XINT_addp_ABEA\the\numexpr #1+10#5#4#3#2+#6\relax
}%
\def\XINT_addp_ABEA #1#2#3#4#5#6#7%
{%
   \XINT_addp_A  #2{#7#6#5#4#3}%<-- attention on met donc \`a droite
}%
\def\XINT_addp_C #1#2#3#4#5%
{%
    \xint_gob_til_W
    #5\xint_addp_cw
    #4\xint_addp_cx
    #3\xint_addp_cy
    #2\xint_addp_cz
    \W\XINT_addp_CD   {#5#4#3#2}{#1}%
}%
\def\XINT_addp_CD #1%
{%
    \expandafter\XINT_addp_CC\the\numexpr 1+10#1\relax
}%
\def\XINT_addp_CC #1#2#3#4#5#6#7%
```

```
{%
    \XINT_addp_AC_checkcarry  #2{#7#6#5#4#3}%
}%
\def\xint_addp_cw
    #1\xint_addp_cx
    #2\xint_addp_cy
    #3\xint_addp_cz
    \W\XINT_addp_CD
{%
    \expandafter\XINT_addp_CDw\the\numexpr \xint_c_i+10#1#2#3\relax
}%
\def\XINT_addp_CDw #1#2#3#4#5#6%
{%
    \xint_gob_til_zeros_iv #2#3#4#5\XINT_addp_endDw_zeros
                          0000\XINT_addp_endDw #2#3#4#5%
}%
\def\XINT_addp_endDw_zeros 0000\XINT_addp_endDw 0000#1\X\Y\Z{ #1}%
\def\XINT_addp_endDw #1#2#3#4#5\X\Y\Z{ #5#4#3#2#1}%
\def\xint_addp_cx
    #1\xint_addp_cy
    #2\xint_addp_cz
    \W\XINT_addp_CD
{%
    \expandafter\XINT_addp_CDx\the\numexpr \xint_c_i+100#1#2\relax
}%
\def\XINT_addp_CDx #1#2#3#4#5#6%
{%
    \xint_gob_til_zeros_iv #2#3#4#5\XINT_addp_endDx_zeros
                          0000\XINT_addp_endDx #2#3#4#5%
}%
\def\XINT_addp_endDx_zeros 0000\XINT_addp_endDx 0000#1\Y\Z{ #1}%
\def\XINT_addp_endDx #1#2#3#4#5\Y\Z{ #5#4#3#2#1}%
\def\xint_addp_cy #1\xint_addp_cz\W\XINT_addp_CD
{%
    \expandafter\XINT_addp_CDy\the\numexpr \xint_c_i+1000#1\relax
}%
\def\XINT_addp_CDy #1#2#3#4#5#6%
{%
    \xint_gob_til_zeros_iv #2#3#4#5\XINT_addp_endDy_zeros
                          0000\XINT_addp_endDy #2#3#4#5%
}%
\def\XINT_addp_endDy_zeros 0000\XINT_addp_endDy 0000#1\Z{ #1}%
\def\XINT_addp_endDy #1#2#3#4#5\Z{ #5#4#3#2#1}%
\def\xint_addp_cz\W\XINT_addp_CD #1#2{ #21000}%
\def\XINT_addp_F #1#2#3#4#5%
{%
    \xint_gob_til_W
    #5\xint_addp_Gw
    #4\xint_addp_Gx
```

```
    #3\xint_addp_Gy
    #2\xint_addp_Gz
    \W\XINT_addp_G   {#2#3#4#5}{#1}%
}%
\def\XINT_addp_G #1#2%
{%
    \XINT_addp_F {#2#1}%
}%
\def\xint_addp_Gw
    #1\xint_addp_Gx
    #2\xint_addp_Gy
    #3\xint_addp_Gz
    \W\XINT_addp_G #4%
{%
    \xint_gob_til_zeros_iv  #3#2#10\XINT_addp_endGw_zeros
                           0000\XINT_addp_endGw #3#2#10%
}%
\def\XINT_addp_endGw_zeros 0000\XINT_addp_endGw 0000#1\X\Y\Z{ #1}%
\def\XINT_addp_endGw #1#2#3#4#5\X\Y\Z{ #5#1#2#3#4}%
\def\xint_addp_Gx
    #1\xint_addp_Gy
    #2\xint_addp_Gz
    \W\XINT_addp_G #3%
{%
    \xint_gob_til_zeros_iv  #2#100\XINT_addp_endGx_zeros
                          0000\XINT_addp_endGx #2#100%
}%
\def\XINT_addp_endGx_zeros 0000\XINT_addp_endGx 0000#1\Y\Z{ #1}%
\def\XINT_addp_endGx #1#2#3#4#5\Y\Z{ #5#1#2#3#4}%
\def\xint_addp_Gy
    #1\xint_addp_Gz
    \W\XINT_addp_G #2%
{%
    \xint_gob_til_zeros_iv   #1000\XINT_addp_endGy_zeros
                          0000\XINT_addp_endGy #1000%
}%
\def\XINT_addp_endGy_zeros 0000\XINT_addp_endGy 0000#1\Z{ #1}%
\def\XINT_addp_endGy #1#2#3#4#5\Z{ #5#1#2#3#4}%
\def\xint_addp_Gz\W\XINT_addp_G #1#2{ #2}%
```

32.27 \xintAdd

Release 1.09a has \xintnum added into \xintiAdd.

```
\def\xintiiAdd {\romannumeral0\xintiiadd }%
\def\xintiiadd #1%
{%
    \expandafter\xint_iiadd\expandafter{\romannumeral-`0#1}%
}%
```

```
\def\xint_iiadd #1#2%
{%
    \expandafter\XINT_add_fork \romannumeral-`0#2\Z #1\Z
}%
\def\xintiAdd {\romannumeral0\xintiadd }%
\def\xintiadd #1%
{%
    \expandafter\xint_add\expandafter{\romannumeral0\xintnum{#1}}%
}%
\def\xint_add #1#2%
{%
    \expandafter\XINT_add_fork \romannumeral0\xintnum{#2}\Z #1\Z
}%
\let\xintAdd\xintiAdd \let\xintadd\xintiadd
\def\XINT_Add #1#2{\romannumeral0\XINT_add_fork #2\Z #1\Z }%
\def\XINT_add #1#2{\XINT_add_fork #2\Z #1\Z }%
```

ADDITION Ici #1#2 vient du *deuxième* argument de \xintAdd et #3#4 donc du *premier* [algo plus efficace lorsque le premier est plus long que le second]

```
\def\XINT_add_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\XINT_add_secondiszero
      #3\XINT_add_firstiszero
       0
        {\xint_UDsignsfork
          #1#3\XINT_add_minusminus          % #1 = #3 = -
           #1-\XINT_add_minusplus           % #1 = -
           #3-\XINT_add_plusminus           % #3 = -
            --\XINT_add_plusplus
         \krof }%
    \krof
    {#2}{#4}#1#3%
}%
\def\XINT_add_secondiszero #1#2#3#4{ #4#2}%
\def\XINT_add_firstiszero #1#2#3#4{ #3#1}%
```

#1 vient du *deuxième* et #2 vient du *premier*

```
\def\XINT_add_minusminus #1#2#3#4%
{%
    \expandafter\xint_minus_thenstop%
    \romannumeral0\XINT_add_pre {#2}{#1}%
}%
\def\XINT_add_minusplus #1#2#3#4%
{%
    \XINT_sub_pre {#4#2}{#1}%
}%
\def\XINT_add_plusminus #1#2#3#4%
```

```
{%
    \XINT_sub_pre {#3#1}{#2}%
}%
\def\XINT_add_plusplus #1#2#3#4%
{%
    \XINT_add_pre {#4#2}{#3#1}%
}%
\def\XINT_add_pre #1%
{%
  \expandafter\XINT_add_pre_b\expandafter
  {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z }%
}%
\def\XINT_add_pre_b #1#2%
{%
    \expandafter\XINT_add_A
        \expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
        \W\X\Y\Z #1\W\X\Y\Z
}%
```

32.28 \xintSub

Release 1.09a has \xintnum added into \xintiSub.

```
\def\xintiiSub {\romannumeral0\xintiisub }%
\def\xintiisub #1%
{%
    \expandafter\xint_iisub\expandafter{\romannumeral-`0#1}%
}%
\def\xint_iisub #1#2%
{%
    \expandafter\XINT_sub_fork \romannumeral-`0#2\Z #1\Z
}%
\def\xintiSub {\romannumeral0\xintisub }%
\def\xintisub #1%
{%
    \expandafter\xint_sub\expandafter{\romannumeral0\xintnum{#1}}%
}%
\def\xint_sub #1#2%
{%
    \expandafter\XINT_sub_fork \romannumeral0\xintnum{#2}\Z #1\Z
}%
\def\XINT_Sub #1#2{\romannumeral0\XINT_sub_fork #2\Z #1\Z }%
\def\XINT_sub #1#2{\XINT_sub_fork #2\Z #1\Z }%
\let\xintSub\xintiSub \let\xintsub\xintisub
```

SOUSTRACTION #3#4-#1#2: #3#4 vient du *premier* #1#2 vient du *second*

```
\def\XINT_sub_fork #1#2\Z #3#4\Z
```

```
{%
    \xint_UDsignsfork
          #1#3\XINT_sub_minusminus
           #1-\XINT_sub_minusplus   % attention, #3=0 possible
           #3-\XINT_sub_plusminus   % attention, #1=0 possible
            --{\xint_UDzerofork
                       #1\XINT_sub_secondiszero
                       #3\XINT_sub_firstiszero
                        0\XINT_sub_plusplus
                       \krof }%
    \krof
    {#2}{#4}#1#3%
}%
\def\XINT_sub_secondiszero #1#2#3#4{ #4#2}%
\def\XINT_sub_firstiszero  #1#2#3#4{ -#3#1}%
\def\XINT_sub_plusplus #1#2#3#4%
{%
    \XINT_sub_pre {#4#2}{#3#1}%
}%
\def\XINT_sub_minusminus #1#2#3#4%
{%
    \XINT_sub_pre {#1}{#2}%
}%
\def\XINT_sub_minusplus #1#2#3#4%
{%
    \xint_gob_til_zero #4\xint_sub_mp0\XINT_add_pre {#4#2}{#1}%
}%
\def\xint_sub_mp0\XINT_add_pre #1#2{ #2}%
\def\XINT_sub_plusminus #1#2#3#4%
{%
    \xint_gob_til_zero #3\xint_sub_pm0\expandafter\xint_minus_thenstop%
    \romannumeral0\XINT_add_pre {#2}{#3#1}%
}%
\def\xint_sub_pm #1\XINT_add_pre #2#3{ -#2}%
\def\XINT_sub_pre #1%
{%
  \expandafter\XINT_sub_pre_b\expandafter
  {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z }%
}%
\def\XINT_sub_pre_b #1#2%
{%
    \expandafter\XINT_sub_A
        \expandafter1\expandafter{\expandafter}%
    \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
        \W\X\Y\Z #1 \W\X\Y\Z
}%
```

\romannumeral0\XINT_sub_A 1{}<N1>\W\X\Y\Z<N2>\W\X\Y\Z
N1 et N2 sont présentés à l'envers ET ON A RAJOUTÉ DES ZÉROS POUR QUE LEURS

LONGUEURS À CHACUN SOIENT MULTIPLES DE 4, MAIS AUCUN NE SE TERMINE EN 0000.
output: N2 - N1
Elle donne le résultat dans le **bon ordre**, avec le bon signe, et sans zéros superflus.

```
\def\XINT_sub_A #1#2#3\W\X\Y\Z #4#5#6#7%
{%
    \xint_gob_til_W
    #4\xint_sub_az
    \W\XINT_sub_B #1{#4#5#6#7}{#2}#3\W\X\Y\Z
}%
\def\XINT_sub_B #1#2#3#4#5#6#7%
{%
    \xint_gob_til_W
    #4\xint_sub_bz
    \W\XINT_sub_onestep #1#2{#7#6#5#4}{#3}%
}%
```

d'abord la branche principale #6 = 4 chiffres de N1, plus significatif en *premier*, #2#3#4#5 chiffres de N2, plus significatif en *dernier* On veut N2 - N1.

```
\def\XINT_sub_onestep #1#2#3#4#5#6%
{%
    \expandafter\XINT_sub_backtoA\the\numexpr 11#5#4#3#2-#6+#1-\xint_c_i.%
}%
```

ON PRODUIT LE RÉSULTAT DANS LE BON ORDRE

```
\def\XINT_sub_backtoA #1#2#3.#4%
{%
    \XINT_sub_A #2{#3#4}%
}%
\def\xint_sub_bz
    \W\XINT_sub_onestep #1#2#3#4#5#6#7%
{%
    \xint_UDzerofork
      #1\XINT_sub_C   % une retenue
       0\XINT_sub_D   % pas de retenue
    \krof
    {#7}#2#3#4#5%
}%
\def\XINT_sub_D #1#2\W\X\Y\Z
{%
    \expandafter
    \xint_cleanupzeros_andstop
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
```

```
      \xint_relax
    #1%
}%
\def\XINT_sub_C #1#2#3#4#5%
{%
    \xint_gob_til_W
    #2\xint_sub_cz
    \W\XINT_sub_AC_onestep {#5#4#3#2}{#1}%
}%
\def\XINT_sub_AC_onestep #1%
{%
    \expandafter\XINT_sub_backtoC\the\numexpr 11#1-\xint_c_i.%
}%
\def\XINT_sub_backtoC #1#2#3.#4%
{%
    \XINT_sub_AC_checkcarry #2{#3#4}% la retenue va \^etre examin\'ee
}%
\def\XINT_sub_AC_checkcarry #1%
{%
    \xint_gob_til_one #1\xint_sub_AC_nocarry 1\XINT_sub_C
}%
\def\xint_sub_AC_nocarry 1\XINT_sub_C #1#2\W\X\Y\Z
{%
    \expandafter
    \XINT_cuz_loop
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax
    #1\W\W\W\W\W\W\W\Z
}%
\def\xint_sub_cz\W\XINT_sub_AC_onestep #1%
{%
    \XINT_cuz
}%
\def\xint_sub_az\W\XINT_sub_B #1#2#3#4#5#6#7%
{%
    \xint_gob_til_W
    #4\xint_sub_ez
    \W\XINT_sub_Eenter #1{#3}#4#5#6#7%
}%
```

le premier nombre continue, le résultat sera < 0.

```
\def\XINT_sub_Eenter #1#2%
{%
    \expandafter
    \XINT_sub_E\expandafter1\expandafter{\expandafter}%
```

```
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax
    \W\X\Y\Z #1%
}%
\def\XINT_sub_E #1#2#3#4#5#6%
{%
    \xint_gob_til_W #3\xint_sub_F\W
    \XINT_sub_Eonestep #1{#6#5#4#3}{#2}%
}%
\def\XINT_sub_Eonestep #1#2%
{%
    \expandafter\XINT_sub_backtoE\the\numexpr 109999-#2+#1.%
}%
\def\XINT_sub_backtoE #1#2#3.#4%
{%
    \XINT_sub_E #2{#3#4}%
}%
\def\xint_sub_F\W\XINT_sub_Eonestep #1#2#3#4%
{%
    \xint_UDonezerofork
      #4#1{\XINT_sub_Fdec 0}% soustraire  1. Et faire signe -
      #1#4{\XINT_sub_Finc 1}% additionner 1. Et faire signe -
        10\XINT_sub_DD     % terminer. Mais avec signe -
    \krof
    {#3}%
}%
\def\XINT_sub_DD {\expandafter\xint_minus_thenstop\romannumeral0\XINT_sub_D }%
\def\XINT_sub_Fdec #1#2#3#4#5#6%
{%
    \xint_gob_til_W #3\xint_sub_Fdec_finish\W
    \XINT_sub_Fdec_onestep #1{#6#5#4#3}{#2}%
}%
\def\XINT_sub_Fdec_onestep #1#2%
{%
    \expandafter\XINT_sub_backtoFdec\the\numexpr 11#2+#1-\xint_c_i.%
}%
\def\XINT_sub_backtoFdec #1#2#3.#4%
{%
    \XINT_sub_Fdec #2{#3#4}%
}%
\def\xint_sub_Fdec_finish\W\XINT_sub_Fdec_onestep #1#2%
{%
    \expandafter\xint_minus_thenstop\romannumeral0\XINT_cuz
}%
\def\XINT_sub_Finc #1#2#3#4#5#6%
```

```
{%
    \xint_gob_til_W #3\xint_sub_Finc_finish\W
    \XINT_sub_Finc_onestep #1{#6#5#4#3}{#2}%
}%
\def\XINT_sub_Finc_onestep #1#2%
{%
    \expandafter\XINT_sub_backtoFinc\the\numexpr 10#2+#1.%
}%
\def\XINT_sub_backtoFinc #1#2#3.#4%
{%
    \XINT_sub_Finc #2{#3#4}%
}%
\def\xint_sub_Finc_finish\W\XINT_sub_Finc_onestep #1#2#3%
{%
    \xint_UDzerofork
     #1{\expandafter\expandafter\expandafter
        \xint_minus_thenstop\xint_cleanupzeros_nostop}%
      0{ -1}%
    \krof
    #3%
}%
\def\xint_sub_ez\W\XINT_sub_Eenter #1%
{%
    \xint_UDzerofork
      #1\XINT_sub_K % il y a une retenue
       0\XINT_sub_L % pas de retenue
    \krof
}%
\def\XINT_sub_L #1\W\X\Y\Z {\XINT_cuz_loop #1\W\W\W\W\W\W\W\Z }%
\def\XINT_sub_K #1%
{%
    \expandafter
    \XINT_sub_KK\expandafter1\expandafter{\expandafter}%
    \romannumeral0%
    \XINT_rord_main {}#1%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax
}%
\def\XINT_sub_KK #1#2#3#4#5#6%
{%
    \xint_gob_til_W #3\xint_sub_KK_finish\W
    \XINT_sub_KK_onestep #1{#6#5#4#3}{#2}%
}%
\def\XINT_sub_KK_onestep #1#2%
{%
    \expandafter\XINT_sub_backtoKK\the\numexpr 109999-#2+#1.%
}%
```

```
\def\XINT_sub_backtoKK #1#2#3.#4%
{%
    \XINT_sub_KK #2{#3#4}%
}%
\def\xint_sub_KK_finish\W\XINT_sub_KK_onestep #1#2#3%
{%
    \expandafter\xint_minus_thenstop
    \romannumeral0\XINT_cuz_loop #3\W\W\W\W\W\W\W\Z
}%
```

32.29 \xintCmp

Release 1.09a has \xintnum inserted into \xintCmp. Unnecessary \xintiCmp suppressed in 1.09f.

```
\def\xintCmp {\romannumeral0\xintcmp }%
\def\xintcmp #1%
{%
    \expandafter\xint_cmp\expandafter{\romannumeral0\xintnum{#1}}%
}%
\def\xint_cmp #1#2%
{%
    \expandafter\XINT_cmp_fork \romannumeral0\xintnum{#2}\Z #1\Z
}%
\def\XINT_Cmp #1#2{\romannumeral0\XINT_cmp_fork #2\Z #1\Z }%
```

COMPARAISON
1 si #3#4>#1#2, 0 si #3#4=#1#2, -1 si #3#4<#1#2
#3#4 vient du *premier*, #1#2 vient du *second*

```
\def\XINT_cmp_fork #1#2\Z #3#4\Z
{%
    \xint_UDsignsfork
          #1#3\XINT_cmp_minusminus
           #1-\XINT_cmp_minusplus
           #3-\XINT_cmp_plusminus
            --{\xint_UDzerosfork
                        #1#3\XINT_cmp_zerozero
                         #10\XINT_cmp_zeroplus
                         #30\XINT_cmp_pluszero
                          00\XINT_cmp_plusplus
                        \krof }%
    \krof
    {#2}{#4}#1#3%
}%
\def\XINT_cmp_minusplus #1#2#3#4{ 1}%
\def\XINT_cmp_plusminus #1#2#3#4{ -1}%
\def\XINT_cmp_zerozero  #1#2#3#4{ 0}%
\def\XINT_cmp_zeroplus  #1#2#3#4{ 1}%
```

```
\def\XINT_cmp_pluszero  #1#2#3#4{ -1}%
\def\XINT_cmp_plusplus #1#2#3#4%
{%
    \XINT_cmp_pre {#4#2}{#3#1}%
}%
\def\XINT_cmp_minusminus #1#2#3#4%
{%
    \XINT_cmp_pre {#1}{#2}%
}%
\def\XINT_cmp_pre #1%
{%
  \expandafter\XINT_cmp_pre_b\expandafter
  {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z }%
}%
\def\XINT_cmp_pre_b #1#2%
{%
    \expandafter\XINT_cmp_A
    \expandafter1\expandafter{\expandafter}%
    \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
        \W\X\Y\Z #1\W\X\Y\Z
}%
```

COMPARAISON
N1 et N2 sont présentés à l'envers ET ON A RAJOUTÉ DES ZÉROS POUR QUE LEUR LONGUEURS À CHACUN SOIENT MULTIPLES DE 4, MAIS AUCUN NE SE TERMINE EN 0000. routine appelée via
\XINT_cmp_A 1{}<N1>\W\X\Y\Z<N2>\W\X\Y\Z
ATTENTION RENVOIE 1 SI N1 < N2, 0 si N1 = N2, -1 si N1 > N2

```
\def\XINT_cmp_A #1#2#3\W\X\Y\Z #4#5#6#7%
{%
    \xint_gob_til_W #4\xint_cmp_az\W
    \XINT_cmp_B #1{#4#5#6#7}{#2}#3\W\X\Y\Z
}%
\def\XINT_cmp_B #1#2#3#4#5#6#7%
{%
    \xint_gob_til_W#4\xint_cmp_bz\W
    \XINT_cmp_onestep #1#2{#7#6#5#4}{#3}%
}%
\def\XINT_cmp_onestep #1#2#3#4#5#6%
{%
    \expandafter\XINT_cmp_backtoA\the\numexpr 11#5#4#3#2-#6+#1-\xint_c_i.%
}%
\def\XINT_cmp_backtoA #1#2#3.#4%
{%
    \XINT_cmp_A #2{#3#4}%
}%
\def\xint_cmp_bz\W\XINT_cmp_onestep #1\Z { 1}%
\def\xint_cmp_az\W\XINT_cmp_B #1#2#3#4#5#6#7%
{%
```

```
    \xint_gob_til_W #4\xint_cmp_ez\W
    \XINT_cmp_Eenter #1{#3}#4#5#6#7%
}%
\def\XINT_cmp_Eenter #1\Z { -1}%
\def\xint_cmp_ez\W\XINT_cmp_Eenter #1%
{%
    \xint_UDzerofork
      #1\XINT_cmp_K             %     il y a une retenue
       0\XINT_cmp_L             %     pas de retenue
    \krof
}%
\def\XINT_cmp_K #1\Z { -1}%
\def\XINT_cmp_L #1{\XINT_OneIfPositive_main #1}%
\def\XINT_OneIfPositive #1%
{%
    \XINT_OneIfPositive_main #1\W\X\Y\Z%
}%
\def\XINT_OneIfPositive_main #1#2#3#4%
{%
    \xint_gob_til_Z #4\xint_OneIfPositive_terminated\Z
    \XINT_OneIfPositive_onestep #1#2#3#4%
}%
\def\xint_OneIfPositive_terminated\Z\XINT_OneIfPositive_onestep\W\X\Y\Z { 0}%
\def\XINT_OneIfPositive_onestep #1#2#3#4%
{%
    \expandafter\XINT_OneIfPositive_check\the\numexpr #1#2#3#4\relax
}%
\def\XINT_OneIfPositive_check #1%
{%
    \xint_gob_til_zero #1\xint_OneIfPositive_backtomain 0%
    \XINT_OneIfPositive_finish #1%
}%
\def\XINT_OneIfPositive_finish #1\W\X\Y\Z{ 1}%
\def\xint_OneIfPositive_backtomain 0\XINT_OneIfPositive_finish 0%
                   {\XINT_OneIfPositive_main }%
```

32.30 \xintEq, \xintGt, \xintLt

1.09a.

```
\def\xintEq {\romannumeral0\xinteq }%
\def\xinteq #1#2{\xintifeq{#1}{#2}{1}{0}}%
\def\xintGt {\romannumeral0\xintgt }%
\def\xintgt #1#2{\xintifgt{#1}{#2}{1}{0}}%
\def\xintLt {\romannumeral0\xintlt }%
\def\xintlt #1#2{\xintiflt{#1}{#2}{1}{0}}%
```

32.31 \xintIsZero, \xintIsNotZero

1.09a. restyled in 1.09i.

```
\def\xintIsZero {\romannumeral0\xintiszero }%
\def\xintiszero #1{\if0\xintSgn{#1}\xint_afterfi{ 1}\else\xint_afterfi{ 0}\fi}%
\def\xintIsNotZero {\romannumeral0\xintisnotzero }%
\def\xintisnotzero
          #1{\if0\xintSgn{#1}\xint_afterfi{ 0}\else\xint_afterfi{ 1}\fi}%
```

32.32 \xintIsTrue, \xintNot, \xintIsFalse

1.09c

```
\let\xintIsTrue\xintIsNotZero
\let\xintNot\xintIsZero
\let\xintIsFalse\xintIsZero
```

32.33 \xintAND, \xintOR, \xintXOR

1.09a. Embarrasing bugs in \xintAND and \xintOR which inserted a space token corrected in 1.09i. \xintxor restyled with \if (faster) in 1.09i

```
\def\xintAND {\romannumeral0\xintand }%
\def\xintand #1#2{\if0\xintSgn{#1}\expandafter\xint_firstoftwo
                             \else\expandafter\xint_secondoftwo\fi
                  { 0}{\xintisnotzero{#2}}}%
\def\xintOR {\romannumeral0\xintor }%
\def\xintor #1#2{\if0\xintSgn{#1}\expandafter\xint_firstoftwo
                            \else\expandafter\xint_secondoftwo\fi
                 {\xintisnotzero{#2}}{ 1}}%
\def\xintXOR {\romannumeral0\xintxor }%
\def\xintxor #1#2{\if\xintIsZero{#1}\xintIsZero{#2}%
                      \xint_afterfi{ 0}\else\xint_afterfi{ 1}\fi }%
```

32.34 \xintANDof

New with 1.09a. \xintANDof works also with an empty list.

```
\def\xintANDof      {\romannumeral0\xintandof }%
\def\xintandof     #1{\expandafter\XINT_andof_a\romannumeral-`0#1\relax }%
\def\XINT_andof_a #1{\expandafter\XINT_andof_b\romannumeral-`0#1\Z }%
\def\XINT_andof_b #1%
           {\xint_gob_til_relax #1\XINT_andof_e\relax\XINT_andof_c #1}%
\def\XINT_andof_c #1\Z
           {\xintifTrueAelseB {#1}{\XINT_andof_a}{\XINT_andof_no}}%
\def\XINT_andof_no #1\relax { 0}%
\def\XINT_andof_e #1\Z { 1}%
```

32.35 \xintORof

New with 1.09a. Works also with an empty list.

```
\def\xintORof      {\romannumeral0\xintorof }%
\def\xintorof    #1{\expandafter\XINT_orof_a\romannumeral-‘0#1\relax }%
\def\XINT_orof_a #1{\expandafter\XINT_orof_b\romannumeral-‘0#1\Z }%
\def\XINT_orof_b #1%
           {\xint_gob_til_relax #1\XINT_orof_e\relax\XINT_orof_c #1}%
\def\XINT_orof_c #1\Z
           {\xintifTrueAelseB {#1}{\XINT_orof_yes}{\XINT_orof_a}}%
\def\XINT_orof_yes #1\relax { 1}%
\def\XINT_orof_e #1\Z { 0}%
```

32.36 \xintXORof

New with 1.09a. Works with an empty list, too. \XINT_xorof_c more efficient in 1.09i

```
\def\xintXORof      {\romannumeral0\xintxorof }%
\def\xintxorof    #1{\expandafter\XINT_xorof_a\expandafter
                     0\romannumeral-‘0#1\relax }%
\def\XINT_xorof_a #1#2{\expandafter\XINT_xorof_b\romannumeral-‘0#2\Z #1}%
\def\XINT_xorof_b #1%
           {\xint_gob_til_relax #1\XINT_xorof_e\relax\XINT_xorof_c #1}%
\def\XINT_xorof_c #1\Z #2%
           {\xintifTrueAelseB {#1}{\if #20\xint_afterfi{\XINT_xorof_a 1}%
                                 \else\xint_afterfi{\XINT_xorof_a 0}\fi}%
                                {\XINT_xorof_a #2}%
           }%
\def\XINT_xorof_e #1\Z #2{ #2}%
```

32.37 \xintGeq

Release 1.09a has \xintnum added into \xintGeq. Unused and useless \xintiGeq removed in 1.09e. PLUS GRAND OU ÉGAL attention compare les **valeurs absolues**

```
\def\xintGeq {\romannumeral0\xintgeq }%
\def\xintgeq #1%
{%
    \expandafter\xint_geq\expandafter {\romannumeral0\xintnum{#1}}%
}%
\def\xint_geq #1#2%
{%
    \expandafter\XINT_geq_fork \romannumeral0\xintnum{#2}\Z #1\Z
}%
\def\XINT_Geq #1#2{\romannumeral0\XINT_geq_fork #2\Z #1\Z }%
```

PLUS GRAND OU ÉGAL ATTENTION, TESTE les VALEURS ABSOLUES

```
\def\XINT_geq_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\XINT_geq_secondiszero % |#1#2|=0
      #3\XINT_geq_firstiszero % |#1#2|>0
       0{\xint_UDsignsfork
                  #1#3\XINT_geq_minusminus
                   #1-\XINT_geq_minusplus
                   #3-\XINT_geq_plusminus
                    --\XINT_geq_plusplus
                \krof }%
    \krof
    {#2}{#4}#1#3%
}%
\def\XINT_geq_secondiszero     #1#2#3#4{ 1}%
\def\XINT_geq_firstiszero      #1#2#3#4{ 0}%
\def\XINT_geq_plusplus   #1#2#3#4{\XINT_geq_pre {#4#2}{#3#1}}%
\def\XINT_geq_minusminus #1#2#3#4{\XINT_geq_pre {#2}{#1}}%
\def\XINT_geq_minusplus  #1#2#3#4{\XINT_geq_pre {#4#2}{#1}}%
\def\XINT_geq_plusminus  #1#2#3#4{\XINT_geq_pre {#2}{#3#1}}%
\def\XINT_geq_pre #1%
{%
  \expandafter\XINT_geq_pre_b\expandafter
  {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z }%
}%
\def\XINT_geq_pre_b #1#2%
{%
    \expandafter\XINT_geq_A
    \expandafter1\expandafter{\expandafter}%
    \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
        \W\X\Y\Z #1 \W\X\Y\Z
}%
```

PLUS GRAND OU ÉGAL
N1 et N2 sont présentés à l'envers ET ON A RAJOUTÉ DES ZÉROS POUR QUE LEURS LONGUEURS À CHACUN SOIENT MULTIPLES DE 4, MAIS AUCUN NE SE TERMINE EN 0000
routine appelée via
\romannumeral0\XINT_geq_A 1{}<N1>\W\X\Y\Z<N2>\W\X\Y\Z
ATTENTION RENVOIE 1 SI N1 < N2 ou N1 = N2 et 0 si N1 > N2

```
\def\XINT_geq_A #1#2#3\W\X\Y\Z #4#5#6#7%
{%
    \xint_gob_til_W #4\xint_geq_az\W
    \XINT_geq_B #1{#4#5#6#7}{#2}#3\W\X\Y\Z
}%
\def\XINT_geq_B #1#2#3#4#5#6#7%
{%
    \xint_gob_til_W #4\xint_geq_bz\W
```

```
    \XINT_geq_onestep #1#2{#7#6#5#4}{#3}%
}%
\def\XINT_geq_onestep #1#2#3#4#5#6%
{%
    \expandafter\XINT_geq_backtoA\the\numexpr 11#5#4#3#2-#6+#1-\xint_c_i.%
}%
\def\XINT_geq_backtoA #1#2#3.#4%
{%
    \XINT_geq_A #2{#3#4}%
}%
\def\xint_geq_bz\W\XINT_geq_onestep #1\W\X\Y\Z { 1}%
\def\xint_geq_az\W\XINT_geq_B #1#2#3#4#5#6#7%
{%
    \xint_gob_til_W #4\xint_geq_ez\W
    \XINT_geq_Eenter #1%
}%
\def\XINT_geq_Eenter #1\W\X\Y\Z { 0}%
\def\xint_geq_ez\W\XINT_geq_Eenter #1%
{%
    \xint_UDzerofork
      #1{ 0}            %     il y a une retenue
       0{ 1}            %     pas de retenue
    \krof
}%
```

32.38 \xintMax

The rationale is that it is more efficient than using \xintCmp. 1.03 makes the code a tiny bit slower but easier to re-use for fractions. Note: actually since 1.08a code for fractions does not all reduce to these entry points, so perhaps I should revert the changes made in 1.03. Release 1.09a has \xintnum added into \xintiMax.

```
\def\xintiMax {\romannumeral0\xintimax }%
\def\xintimax #1%
{%
    \expandafter\xint_max\expandafter {\romannumeral0\xintnum{#1}}%
}%
\let\xintMax\xintiMax \let\xintmax\xintimax
\def\xint_max #1#2%
{%
    \expandafter\XINT_max_pre\expandafter {\romannumeral0\xintnum{#2}}{#1}%
}%
\def\XINT_max_pre #1#2{\XINT_max_fork #1\Z #2\Z {#2}{#1}}%
\def\XINT_Max #1#2{\romannumeral0\XINT_max_fork #2\Z #1\Z {#1}{#2}}%
```

#3#4 vient du *premier*, #1#2 vient du *second*

```
\def\XINT_max_fork #1#2\Z #3#4\Z
```

```
{%
    \xint_UDsignsfork
          #1#3\XINT_max_minusminus  % A < 0, B < 0
           #1-\XINT_max_minusplus   % B < 0, A >= 0
           #3-\XINT_max_plusminus   % A < 0, B >= 0
            --{\xint_UDzerosfork
                      #1#3\XINT_max_zerozero % A = B = 0
                       #10\XINT_max_zeroplus % B = 0, A > 0
                       #30\XINT_max_pluszero % A = 0, B > 0
                        00\XINT_max_plusplus % A, B > 0
                      \krof }%
    \krof
    {#2}{#4}#1#3%
}%
```

A = #4#2, B = #3#1

```
\def\XINT_max_zerozero  #1#2#3#4{\xint_firstoftwo_thenstop }%
\def\XINT_max_zeroplus  #1#2#3#4{\xint_firstoftwo_thenstop }%
\def\XINT_max_pluszero  #1#2#3#4{\xint_secondoftwo_thenstop }%
\def\XINT_max_minusplus #1#2#3#4{\xint_firstoftwo_thenstop }%
\def\XINT_max_plusminus #1#2#3#4{\xint_secondoftwo_thenstop }%
\def\XINT_max_plusplus  #1#2#3#4%
{%
    \ifodd\XINT_Geq {#4#2}{#3#1}
      \expandafter\xint_firstoftwo_thenstop
    \else
      \expandafter\xint_secondoftwo_thenstop
    \fi
}%
```

#3=-, #4=-, #1 = |B| = -B, #2 = |A| = -A

```
\def\XINT_max_minusminus #1#2#3#4%
{%
    \ifodd\XINT_Geq {#1}{#2}
      \expandafter\xint_firstoftwo_thenstop
    \else
      \expandafter\xint_secondoftwo_thenstop
    \fi
}%
```

32.39 \xintMaxof

New with 1.09a.

```
\def\xintiMaxof      {\romannumeral0\xintimaxof }%
\def\xintimaxof    #1{\expandafter\XINT_imaxof_a\romannumeral-`0#1\relax }%
\def\XINT_imaxof_a #1{\expandafter\XINT_imaxof_b\romannumeral0\xintnum{#1}\Z }%
```

```
\def\XINT_imaxof_b #1\Z #2%
           {\expandafter\XINT_imaxof_c\romannumeral-`0#2\Z {#1}\Z}%
\def\XINT_imaxof_c #1%
           {\xint_gob_til_relax #1\XINT_imaxof_e\relax\XINT_imaxof_d #1}%
\def\XINT_imaxof_d #1\Z
           {\expandafter\XINT_imaxof_b\romannumeral0\xintimax {#1}}%
\def\XINT_imaxof_e #1\Z #2\Z { #2}%
\let\xintMaxof\xintiMaxof \let\xintmaxof\xintimaxof
```

32.40 \xintMin

\xintnum added New with 1.09a.

```
\def\xintiMin {\romannumeral0\xintimin }%
\def\xintimin #1%
{%
    \expandafter\xint_min\expandafter {\romannumeral0\xintnum{#1}}%
}%
\let\xintMin\xintiMin \let\xintmin\xintimin
\def\xint_min #1#2%
{%
    \expandafter\XINT_min_pre\expandafter {\romannumeral0\xintnum{#2}}{#1}%
}%
\def\XINT_min_pre #1#2{\XINT_min_fork #1\Z #2\Z {#2}{#1}}%
\def\XINT_Min #1#2{\romannumeral0\XINT_min_fork #2\Z #1\Z {#1}{#2}}%
```

#3#4 vient du *premier*, #1#2 vient du *second*

```
\def\XINT_min_fork #1#2\Z #3#4\Z
{%
    \xint_UDsignsfork
          #1#3\XINT_min_minusminus  % A < 0, B < 0
           #1-\XINT_min_minusplus   % B < 0, A >= 0
           #3-\XINT_min_plusminus   % A < 0, B >= 0
            --{\xint_UDzerosfork
                      #1#3\XINT_min_zerozero % A = B = 0
                       #10\XINT_min_zeroplus % B = 0, A > 0
                       #30\XINT_min_pluszero % A = 0, B > 0
                        00\XINT_min_plusplus % A, B > 0
                      \krof }%
    \krof
    {#2}{#4}#1#3%
}%
```

A = #4#2, B = #3#1

```
\def\XINT_min_zerozero  #1#2#3#4{\xint_firstoftwo_thenstop }%
\def\XINT_min_zeroplus  #1#2#3#4{\xint_secondoftwo_thenstop }%
\def\XINT_min_pluszero  #1#2#3#4{\xint_firstoftwo_thenstop }%
```

```
\def\XINT_min_minusplus #1#2#3#4{\xint_secondoftwo_thenstop }%
\def\XINT_min_plusminus #1#2#3#4{\xint_firstoftwo_thenstop }%
\def\XINT_min_plusplus  #1#2#3#4%
{%
    \ifodd\XINT_Geq {#4#2}{#3#1}
      \expandafter\xint_secondoftwo_thenstop
    \else
      \expandafter\xint_firstoftwo_thenstop
    \fi
}%
```

#3=-, #4=-, #1 = |B| = -B, #2 = |A| = -A

```
\def\XINT_min_minusminus #1#2#3#4%
{%
    \ifodd\XINT_Geq {#1}{#2}
      \expandafter\xint_secondoftwo_thenstop
    \else
      \expandafter\xint_firstoftwo_thenstop
    \fi
}%
```

32.41 \xintMinof

1.09a

```
\def\xintiMinof      {\romannumeral0\xintiminof }%
\def\xintiminof    #1{\expandafter\XINT_iminof_a\romannumeral-`0#1\relax }%
\def\XINT_iminof_a #1{\expandafter\XINT_iminof_b\romannumeral0\xintnum{#1}\Z }%
\def\XINT_iminof_b #1\Z #2%
           {\expandafter\XINT_iminof_c\romannumeral-`0#2\Z {#1}\Z}%
\def\XINT_iminof_c #1%
           {\xint_gob_til_relax #1\XINT_iminof_e\relax\XINT_iminof_d #1}%
\def\XINT_iminof_d #1\Z
           {\expandafter\XINT_iminof_b\romannumeral0\xintimin {#1}}%
\def\XINT_iminof_e #1\Z #2\Z { #2}%
\let\xintMinof\xintiMinof \let\xintminof\xintiminof
```

32.42 \xintSum

\xintSum {{a}{b}...{z}}
\xintSumExpr {a}{b}...{z}\relax
1.03 (drastically) simplifies and makes the routines more efficient (for big computations). Also the way \xintSum and \xintSumExpr ...\relax are related. has been modified. Now \xintSumExpr \z \relax is accepted input when \z expands to a list of braced terms (prior only \xintSum {\z} or \xintSum \z was possible).

1.09a does NOT add the \xintnum overhead. 1.09h renames \xintiSum to \xintiiSum to correctly reflect this.

```
\def\xintiiSum {\romannumeral0\xintiisum }%
\def\xintiisum #1{\xintiisumexpr #1\relax }%
\def\xintiiSumExpr {\romannumeral0\xintiisumexpr }%
\def\xintiisumexpr {\expandafter\XINT_sumexpr\romannumeral-`0}%
\let\xintSum\xintiiSum \let\xintsum\xintiisum
\let\xintSumExpr\xintiiSumExpr \let\xintsumexpr\xintiisumexpr
\def\XINT_sumexpr {\XINT_sum_loop {0000}{0000}}%
\def\XINT_sum_loop #1#2#3%
{%
    \expandafter\XINT_sum_checksign\romannumeral-`0#3\Z {#1}{#2}%
}%
\def\XINT_sum_checksign #1%
{%
    \xint_gob_til_relax #1\XINT_sum_finished\relax
    \xint_gob_til_zero #1\XINT_sum_skipzeroinput0%
    \xint_UDsignfork
      #1\XINT_sum_N
       -{\XINT_sum_P #1}%
    \krof
}%
\def\XINT_sum_finished #1\Z #2#3%
{%
    \XINT_sub_A 1{}#3\W\X\Y\Z #2\W\X\Y\Z
}%
\def\XINT_sum_skipzeroinput #1\krof #2\Z {\XINT_sum_loop }%
\def\XINT_sum_P #1\Z #2%
{%
    \expandafter\XINT_sum_loop\expandafter
    {\romannumeral0\expandafter
     \XINT_addr_A\expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z
    \W\X\Y\Z #2\W\X\Y\Z }%
}%
\def\XINT_sum_N #1\Z #2#3%
{%
    \expandafter\XINT_sum_NN\expandafter
    {\romannumeral0\expandafter
     \XINT_addr_A\expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z
    \W\X\Y\Z #3\W\X\Y\Z }{#2}%
}%
\def\XINT_sum_NN #1#2{\XINT_sum_loop {#2}{#1}}%
```

32.43 \xintMul

1.09a adds \xintnum

```
\def\xintiiMul {\romannumeral0\xintiimul }%
\def\xintiimul #1%
```

```
{%
    \expandafter\xint_iimul\expandafter {\romannumeral-`0#1}%
}%
\def\xint_iimul #1#2%
{%
    \expandafter\XINT_mul_fork \romannumeral-`0#2\Z #1\Z
}%
\def\xintiMul {\romannumeral0\xintimul }%
\def\xintimul #1%
{%
    \expandafter\xint_mul\expandafter {\romannumeral0\xintnum{#1}}%
}%
\def\xint_mul #1#2%
{%
    \expandafter\XINT_mul_fork \romannumeral0\xintnum{#2}\Z #1\Z
}%
\let\xintMul\xintiMul \let\xintmul\xintimul
\def\XINT_Mul #1#2{\romannumeral0\XINT_mul_fork #2\Z #1\Z }%
```

MULTIPLICATION

Ici #1#2 = 2e input et #3#4 = 1er input

Release 1.03 adds some overhead to first compute and compare the lengths of the two inputs. The algorithm is asymmetrical and whether the first input is the longest or the shortest sometimes has a strong impact. 50 digits times 1000 digits used to be 5 times faster than 1000 digits times 50 digits. With the new code, the user input order does not matter as it is decided by the routine what is best. This is important for the extension to fractions, as there is no way then to generally control or guess the most frequent sizes of the inputs besides actually computing their lengths.

```
\def\XINT_mul_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\XINT_mul_zero
      #3\XINT_mul_zero
       0{\xint_UDsignsfork
          #1#3\XINT_mul_minusminus          % #1 = #3 = -
           #1-{\XINT_mul_minusplus #3}%          % #1 = -
           #3-{\XINT_mul_plusminus #1}%          % #3 = -
            --{\XINT_mul_plusplus #1#3}%
         \krof }%
    \krof
    {#2}{#4}%
}%
\def\XINT_mul_zero #1#2{ 0}%
\def\XINT_mul_minusminus #1#2%
{%
      \expandafter\XINT_mul_choice_a
      \expandafter{\romannumeral0\xintlength {#2}}%
      {\romannumeral0\xintlength {#1}}{#1}{#2}%
```

```
}%
\def\XINT_mul_minusplus #1#2#3%
{%
    \expandafter\xint_minus_thenstop\romannumeral0\expandafter
    \XINT_mul_choice_a
    \expandafter{\romannumeral0\xintlength {#1#3}}%
    {\romannumeral0\xintlength {#2}}{#2}{#1#3}%
}%
\def\XINT_mul_plusminus #1#2#3%
{%
    \expandafter\xint_minus_thenstop\romannumeral0\expandafter
    \XINT_mul_choice_a
    \expandafter{\romannumeral0\xintlength {#3}}%
    {\romannumeral0\xintlength {#1#2}}{#1#2}{#3}%
}%
\def\XINT_mul_plusplus #1#2#3#4%
{%
    \expandafter\XINT_mul_choice_a
    \expandafter{\romannumeral0\xintlength {#2#4}}%
    {\romannumeral0\xintlength {#1#3}}{#1#3}{#2#4}%
}%
\def\XINT_mul_choice_a #1#2%
{%
    \expandafter\XINT_mul_choice_b\expandafter{#2}{#1}%
}%
\def\XINT_mul_choice_b #1#2%
{%
    \ifnum #1<\xint_c_v
       \expandafter\XINT_mul_choice_littlebyfirst
    \else
    \ifnum #2<\xint_c_v
      \expandafter\expandafter\expandafter\XINT_mul_choice_littlebysecond
      \else
      \expandafter\expandafter\expandafter\XINT_mul_choice_compare
      \fi
    \fi
    {#1}{#2}%
}%
\def\XINT_mul_choice_littlebyfirst #1#2#3#4%
{%
    \expandafter\XINT_mul_M
    \expandafter{\the\numexpr #3\expandafter}%
    \romannumeral0\XINT_RQ {}#4\R\R\R\R\R\R\R\R\Z \Z\Z\Z\Z
}%
\def\XINT_mul_choice_littlebysecond #1#2#3#4%
{%
    \expandafter\XINT_mul_M
    \expandafter{\the\numexpr #4\expandafter}%
    \romannumeral0\XINT_RQ {}#3\R\R\R\R\R\R\R\R\Z \Z\Z\Z\Z
```

```
}%
\def\XINT_mul_choice_compare #1#2%
{%
    \ifnum #1>#2
        \expandafter \XINT_mul_choice_i
    \else
        \expandafter \XINT_mul_choice_ii
    \fi
    {#1}{#2}%
}%
\def\XINT_mul_choice_i #1#2%
{%
   \ifnum #1<\numexpr\ifcase \numexpr (#2-\xint_c_iii)/\xint_c_iv\relax
                      \or 330\or 168\or 109\or 80\or 66\or 52\else 0\fi\relax
       \expandafter\XINT_mul_choice_same
   \else
       \expandafter\XINT_mul_choice_permute
   \fi
}%
\def\XINT_mul_choice_ii #1#2%
{%
   \ifnum #2<\numexpr\ifcase \numexpr (#1-\xint_c_iii)/\xint_c_iv\relax
                      \or 330\or 168\or 109\or 80\or 66\or 52\else 0\fi\relax
       \expandafter\XINT_mul_choice_permute
   \else
       \expandafter\XINT_mul_choice_same
   \fi
}%
\def\XINT_mul_choice_same #1#2%
{%
    \expandafter\XINT_mul_enter
    \romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z
    \Z\Z\Z\Z #2\W\W\W\W
}%
\def\XINT_mul_choice_permute #1#2%
{%
    \expandafter\XINT_mul_enter
    \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
    \Z\Z\Z\Z #1\W\W\W\W
}%
```

Cette portion de routine d'addition se branche directement sur _addr_ lorsque le premier nombre est épuisé, ce qui est garanti arriver avant le second nombre. Elle produit son résultat toujours sur 4n, renversé. Ses deux inputs sont garantis sur 4n.

```
\def\XINT_mul_Ar #1#2#3#4#5#6%
{%
    \xint_gob_til_Z #6\xint_mul_br\Z\XINT_mul_Br #1{#6#5#4#3}{#2}%
}%
```

```
\def\xint_mul_br\Z\XINT_mul_Br #1#2%
{%
     \XINT_addr_AC_checkcarry #1%
}%
\def\XINT_mul_Br #1#2#3#4\W\X\Y\Z #5#6#7#8%
{%
    \expandafter\XINT_mul_ABEAr
    \the\numexpr #1+10#2+#8#7#6#5.{#3}#4\W\X\Y\Z
}%
\def\XINT_mul_ABEAr #1#2#3#4#5#6.#7%
{%
    \XINT_mul_Ar #2{#7#6#5#4#3}%
}%
```

<< Petite >> multiplication. mul_Mr renvoie le résultat *à l'envers*, sur *4n*
\romannumeral0\XINT_mul_Mr {<n>}<N>\Z\Z\Z\Z
Fait la multiplication de <N> par <n>, qui est < 10000. <N> est présenté *à l'envers*, sur *4n*. Lorsque <n> vaut 0, donne 0000.

```
\def\XINT_mul_Mr #1%
{%
    \expandafter\XINT_mul_Mr_checkifzeroorone\expandafter{\the\numexpr #1}%
}%
\def\XINT_mul_Mr_checkifzeroorone #1%
{%
    \ifcase #1
      \expandafter\XINT_mul_Mr_zero
    \or
      \expandafter\XINT_mul_Mr_one
    \else
      \expandafter\XINT_mul_Nr
    \fi
    {0000}{}{#1}%
}%
\def\XINT_mul_Mr_zero #1\Z\Z\Z\Z { 0000}%
\def\XINT_mul_Mr_one #1#2#3#4\Z\Z\Z\Z { #4}%
\def\XINT_mul_Nr #1#2#3#4#5#6#7%
{%
    \xint_gob_til_Z #4\xint_mul_pr\Z\XINT_mul_Pr {#1}{#3}{#7#6#5#4}{#2}{#3}%
}%
\def\XINT_mul_Pr #1#2#3%
{%
    \expandafter\XINT_mul_Lr\the\numexpr \xint_c_x^viii+#1+#2*#3\relax
}%
\def\XINT_mul_Lr 1#1#2#3#4#5#6#7#8#9%
{%
    \XINT_mul_Nr {#1#2#3#4}{#9#8#7#6#5}%
}%
\def\xint_mul_pr\Z\XINT_mul_Pr #1#2#3#4#5%
{%
```

```
    \xint_gob_til_zeros_iv #1\XINT_mul_Mr_end_nocarry 0000%
    \XINT_mul_Mr_end_carry #1{#4}%
}%
\def\XINT_mul_Mr_end_nocarry 0000\XINT_mul_Mr_end_carry 0000#1{ #1}%
\def\XINT_mul_Mr_end_carry #1#2#3#4#5{ #5#4#3#2#1}%
```

<< Petite >> multiplication. renvoie le résultat *à l'endroit*, avec *nettoyage des leading zéros*.
\romannumeral0\XINT_mul_M {<n>}<N>\Z\Z\Z\Z
Fait la multiplication de <N> par <n>, qui est < 10000. <N> est présenté *à l'envers*, sur *4n*.

```
\def\XINT_mul_M #1%
{%
    \expandafter\XINT_mul_M_checkifzeroorone\expandafter{\the\numexpr #1}%
}%
\def\XINT_mul_M_checkifzeroorone #1%
{%
    \ifcase #1
      \expandafter\XINT_mul_M_zero
    \or
      \expandafter\XINT_mul_M_one
    \else
      \expandafter\XINT_mul_N
    \fi
    {0000}{}{#1}%
}%
\def\XINT_mul_M_zero #1\Z\Z\Z\Z { 0}%
\def\XINT_mul_M_one #1#2#3#4\Z\Z\Z\Z
{%
    \expandafter\xint_cleanupzeros_andstop\romannumeral0\xintreverseorder{#4}%
}%
\def\XINT_mul_N #1#2#3#4#5#6#7%
{%
    \xint_gob_til_Z #4\xint_mul_p\Z\XINT_mul_P {#1}{#3}{#7#6#5#4}{#2}{#3}%
}%
\def\XINT_mul_P #1#2#3%
{%
    \expandafter\XINT_mul_L\the\numexpr \xint_c_x^viii+#1+#2*#3\relax
}%
\def\XINT_mul_L 1#1#2#3#4#5#6#7#8#9%
{%
    \XINT_mul_N {#1#2#3#4}{#5#6#7#8#9}%
}%
\def\xint_mul_p\Z\XINT_mul_P #1#2#3#4#5%
{%
    \XINT_mul_M_end #1#4%
}%
\edef\XINT_mul_M_end #1#2#3#4#5#6#7#8%
{%
```

```
    \noexpand\expandafter\space\noexpand\the\numexpr #1#2#3#4#5#6#7#8\relax
}%
```

Routine de multiplication principale (attention délimiteurs modifiés pour 1.08)
Le résultat partiel est toujours maintenu avec significatif à droite et il a un nombre multiple de 4 de chiffres
\romannumeral0\XINT_mul_enter <N1>\Z\Z\Z\Z <N2>\W\W\W\W
avec <N1> *renversé*, *longueur 4n* (zéros éventuellement ajoutés au-delà du chiffre le plus significatif) et <N2> dans l'ordre *normal*, et pas forcément longueur 4n. pas de signes.
Pour 1.08: dans \XINT_mul_enter et les modifs de 1.03 qui filtrent les courts, on pourrait croire que le second opérande a au moins quatre chiffres; mais le problème c'est que ceci est appelé par \XINT_sqr. Et de plus \XINT_sqr est utilisé dans la nouvelle routine d'extraction de racine carrée: je ne veux pas rajouter l'overhead à \XINT_sqr de voir si a longueur est au moins 4. Dilemme donc. Il ne semble pas y avoir d'autres accès directs (celui de big fac n'est pas un problème). J'ai presque été tenté de faire du 5x4, mais si on veut maintenir les résultats intermédiaires sur 4n, il y a des complications. Par ailleurs, je modifie aussi un petit peu la façon de coder la suite, compte tenu du style que j'ai développé ultérieurement. Attention terminaison modifiée pour le deuxième opérande.

```
\def\XINT_mul_enter #1\Z\Z\Z\Z #2#3#4#5%
{%
    \xint_gob_til_W #5\XINT_mul_exit_a\W
    \XINT_mul_start {#2#3#4#5}#1\Z\Z\Z\Z
}%
\def\XINT_mul_exit_a\W\XINT_mul_start #1%
{%
    \XINT_mul_exit_b #1%
}%
\def\XINT_mul_exit_b #1#2#3#4%
{%
    \xint_gob_til_W
      #2\XINT_mul_exit_ci
      #3\XINT_mul_exit_cii
      \W\XINT_mul_exit_ciii #1#2#3#4%
}%
\def\XINT_mul_exit_ciii #1\W #2\Z\Z\Z\Z \W\W\W
{%
    \XINT_mul_M {#1}#2\Z\Z\Z\Z
}%
\def\XINT_mul_exit_cii\W\XINT_mul_exit_ciii #1\W\W #2\Z\Z\Z\Z \W\W
{%
    \XINT_mul_M {#1}#2\Z\Z\Z\Z
}%
\def\XINT_mul_exit_ci\W\XINT_mul_exit_cii
                        \W\XINT_mul_exit_ciii #1\W\W\W #2\Z\Z\Z\Z \W
{%
    \XINT_mul_M {#1}#2\Z\Z\Z\Z
}%
```

```
\def\XINT_mul_start #1#2\Z\Z\Z\Z
{%
    \expandafter\XINT_mul_main\expandafter
    {\romannumeral0\XINT_mul_Mr {#1}#2\Z\Z\Z\Z}#2\Z\Z\Z\Z
}%
\def\XINT_mul_main #1#2\Z\Z\Z\Z #3#4#5#6%
{%
    \xint_gob_til_W #6\XINT_mul_finish_a\W
    \XINT_mul_compute {#3#4#5#6}{#1}#2\Z\Z\Z\Z
}%
\def\XINT_mul_compute #1#2#3\Z\Z\Z\Z
{%
    \expandafter\XINT_mul_main\expandafter
    {\romannumeral0\expandafter
     \XINT_mul_Ar\expandafter0\expandafter{\expandafter}%
     \romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z
     \W\X\Y\Z 0000#2\W\X\Y\Z }#3\Z\Z\Z\Z
}%
```

Ici, le deuxième nombre se termine. Fin du calcul. On utilise la variante \XINT_addm_A de l'addition car on sait que le deuxième terme est au moins aussi long que le premier. Lorsque le multiplicateur avait longueur 4n, la dernière addition a fourni le résultat à l'envers, il faut donc encore le renverser.

```
\def\XINT_mul_finish_a\W\XINT_mul_compute #1%
{%
    \XINT_mul_finish_b #1%
}%
\def\XINT_mul_finish_b #1#2#3#4%
{%
    \xint_gob_til_W
      #1\XINT_mul_finish_c
      #2\XINT_mul_finish_ci
      #3\XINT_mul_finish_cii
      \W\XINT_mul_finish_ciii #1#2#3#4%
}%
\def\XINT_mul_finish_ciii #1\W #2#3\Z\Z\Z\Z \W\W\W
{%
    \expandafter\XINT_addm_A\expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z \W\X\Y\Z 000#2\W\X\Y\Z
}%
\def\XINT_mul_finish_cii
    \W\XINT_mul_finish_ciii #1\W\W #2#3\Z\Z\Z\Z \W\W
{%
    \expandafter\XINT_addm_A\expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z \W\X\Y\Z 00#2\W\X\Y\Z
}%
\def\XINT_mul_finish_ci #1\XINT_mul_finish_ciii #2\W\W\W #3#4\Z\Z\Z\Z \W
{%
    \expandafter\XINT_addm_A\expandafter0\expandafter{\expandafter}%
```

```
    \romannumeral0\XINT_mul_Mr {#2}#4\Z\Z\Z\Z \W\X\Y\Z 0#3\W\X\Y\Z
}%
\def\XINT_mul_finish_c #1\XINT_mul_finish_ciii \W\W\W\W #2#3\Z\Z\Z\Z
{%
    \expandafter\xint_cleanupzeros_andstop\romannumeral0\xintreverseorder{#2}%
}%
```

Variante de la Multiplication
\romannumeral0\XINT_mulr_enter <N1>\Z\Z\Z\Z <N2>\W\W\W\W
Ici <N1> est à l'envers sur 4n, et <N2> est à l'endroit, pas sur 4n, comme dans \XINT_mul_enter, mais le résultat est lui-même fourni *à l'envers*, sur *4n* (en faisant attention de ne pas avoir 0000 à la fin).
Utilisé par le calcul des puissances. J'ai modifié dans 1.08 sur le modèle de la nouvelle version de \XINT_mul_enter. Je pourrais économiser des macros et fusionner \XINT_mul_enter et \XINT_mulr_enter. Une autre fois.

```
\def\XINT_mulr_enter #1\Z\Z\Z\Z #2#3#4#5%
{%
    \xint_gob_til_W #5\XINT_mulr_exit_a\W
    \XINT_mulr_start {#2#3#4#5}#1\Z\Z\Z\Z
}%
\def\XINT_mulr_exit_a\W\XINT_mulr_start #1%
{%
    \XINT_mulr_exit_b #1%
}%
\def\XINT_mulr_exit_b #1#2#3#4%
{%
    \xint_gob_til_W
      #2\XINT_mulr_exit_ci
      #3\XINT_mulr_exit_cii
      \W\XINT_mulr_exit_ciii #1#2#3#4%
}%
\def\XINT_mulr_exit_ciii #1\W #2\Z\Z\Z\Z \W\W\W
{%
    \XINT_mul_Mr {#1}#2\Z\Z\Z\Z
}%
\def\XINT_mulr_exit_cii\W\XINT_mulr_exit_ciii #1\W\W #2\Z\Z\Z\Z \W\W
{%
    \XINT_mul_Mr {#1}#2\Z\Z\Z\Z
}%
\def\XINT_mulr_exit_ci\W\XINT_mulr_exit_cii
                    \W\XINT_mulr_exit_ciii #1\W\W\W #2\Z\Z\Z\Z \W
{%
    \XINT_mul_Mr {#1}#2\Z\Z\Z\Z
}%
\def\XINT_mulr_start #1#2\Z\Z\Z\Z
{%
    \expandafter\XINT_mulr_main\expandafter
    {\romannumeral0\XINT_mul_Mr {#1}#2\Z\Z\Z\Z}#2\Z\Z\Z\Z
}%
```

```
\def\XINT_mulr_main #1#2\Z\Z\Z\Z #3#4#5#6%
{%
    \xint_gob_til_W #6\XINT_mulr_finish_a\W
    \XINT_mulr_compute {#3#4#5#6}{#1}#2\Z\Z\Z\Z
}%
\def\XINT_mulr_compute #1#2#3\Z\Z\Z\Z
{%
    \expandafter\XINT_mulr_main\expandafter
    {\romannumeral0\expandafter
     \XINT_mul_Ar\expandafter0\expandafter{\expandafter}%
     \romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z
     \W\X\Y\Z 0000#2\W\X\Y\Z }#3\Z\Z\Z\Z
}%
\def\XINT_mulr_finish_a\W\XINT_mulr_compute #1%
{%
    \XINT_mulr_finish_b #1%
}%
\def\XINT_mulr_finish_b #1#2#3#4%
{%
    \xint_gob_til_W
      #1\XINT_mulr_finish_c
      #2\XINT_mulr_finish_ci
      #3\XINT_mulr_finish_cii
      \W\XINT_mulr_finish_ciii #1#2#3#4%
}%
\def\XINT_mulr_finish_ciii #1\W #2#3\Z\Z\Z\Z \W\W\W
{%
    \expandafter\XINT_addp_A\expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z \W\X\Y\Z 000#2\W\X\Y\Z
}%
\def\XINT_mulr_finish_cii
    \W\XINT_mulr_finish_ciii #1\W\W #2#3\Z\Z\Z\Z \W\W
{%
    \expandafter\XINT_addp_A\expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z \W\X\Y\Z 00#2\W\X\Y\Z
}%
\def\XINT_mulr_finish_ci #1\XINT_mulr_finish_ciii #2\W\W\W #3#4\Z\Z\Z\Z \W
{%
    \expandafter\XINT_addp_A\expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_mul_Mr {#2}#4\Z\Z\Z\Z \W\X\Y\Z 0#3\W\X\Y\Z
}%
\def\XINT_mulr_finish_c #1\XINT_mulr_finish_ciii \W\W\W\W #2#3\Z\Z\Z\Z { #2}%
```

32.44 \xintSqr

```
\def\xintiiSqr {\romannumeral0\xintiisqr }%
\def\xintiisqr #1%
{%
    \expandafter\XINT_sqr\expandafter {\romannumeral0\xintiiabs{#1}}%
```

```
}%
\def\xintiSqr {\romannumeral0\xintisqr }%
\def\xintisqr #1%
{%
    \expandafter\XINT_sqr\expandafter {\romannumeral0\xintiabs{#1}}%
}%
\let\xintSqr\xintiSqr \let\xintsqr\xintisqr
\def\XINT_sqr #1%
{%
    \expandafter\XINT_mul_enter
          \romannumeral0%
          \XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z
          \Z\Z\Z\Z #1\W\W\W\W
}%
```

32.45 \xintPrd

\xintPrd {{a}...{z}}
\xintPrdExpr {a}...{z}\relax
Release 1.02 modified the product routine. The earlier version was faster in situations where each new term is bigger than the product of all previous terms, a situation which arises in the algorithm for computing powers. The 1.02 version was changed to be more efficient on big products, where the new term is small compared to what has been computed so far (the power algorithm now has its own product routine).

Finally, the 1.03 version just simplifies everything as the multiplication now decides what is best, with the price of a little overhead. So the code has been dramatically reduced here.

In 1.03 I also modify the way \xintPrd and \xintPrdExpr ...\relax are related. Now \xintPrdExpr \z \relax is accepted input when \z expands to a list of braced terms (prior only \xintPrd {\z} or \xintPrd \z was possible).

In 1.06a I suddenly decide that \xintProductExpr was a silly name, and as the package is new and certainly not used, I decide I may just switch to \xintPrdExpr which I should have used from the beginning.

1.09a does NOT add the \xintnum overhead. 1.09h renames \xintiPrd to \xintiiPrd to correctly reflect this.

```
\def\xintiiPrd {\romannumeral0\xintiiprd }%
\def\xintiiprd #1{\xintiiprdexpr #1\relax }%
\let\xintPrd\xintiiPrd
\let\xintprd\xintiiprd
\def\xintiiPrdExpr {\romannumeral0\xintiiprdexpr }%
\def\xintiiprdexpr {\expandafter\XINT_prdexpr\romannumeral-`0}%
\let\xintPrdExpr\xintiiPrdExpr
\let\xintprdexpr\xintiiprdexpr
\def\XINT_prdexpr {\XINT_prod_loop_a 1\Z }%
\def\XINT_prod_loop_a #1\Z #2%
    {\expandafter\XINT_prod_loop_b \romannumeral-`0#2\Z #1\Z \Z}%
\def\XINT_prod_loop_b #1%
    {\xint_gob_til_relax #1\XINT_prod_finished\relax\XINT_prod_loop_c #1}%
```

```
\def\XINT_prod_loop_c
    {\expandafter\XINT_prod_loop_a\romannumeral0\XINT_mul_fork }%
\def\XINT_prod_finished #1\Z #2\Z \Z { #2}%
```

32.46 \xintFac

Modified with 1.02 and again in 1.03 for greater efficiency. I am tempted, here and elsewhere, to use \ifcase\XINT_Geq {#1}{1000000000} rather than \ifnum\xintLength {#1}>9 but for the time being I leave things as they stand. With release 1.05, rather than using \xintLength I opt finally for direct use of \numexpr (which will throw a suitable number too big message), and to raise the \xintError: FactorialOfTooBigNumber for argument larger than 1000000 (rather than 1000000000). With 1.09a, \xintFac uses \xintnum.

1.09j for no special reason, I lower the maximal number from 999999 to 100000. Any how this computation would need more memory than TL2013 standard allows to TeX. And I don't even mention time...

```
\def\xintiFac {\romannumeral0\xintifac }%
\def\xintifac #1%
{%
    \expandafter\XINT_fac_fork\expandafter{\the\numexpr #1}%
}%
\let\xintFac\xintiFac \let\xintfac\xintifac
\def\XINT_fac_fork #1%
{%
    \ifcase\XINT_cntSgn #1\Z
       \xint_afterfi{\expandafter\space\expandafter 1\xint_gobble_i }%
    \or
       \expandafter\XINT_fac_checklength
    \else
       \xint_afterfi{\expandafter\xintError:FactorialOfNegativeNumber
                 \expandafter\space\expandafter 1\xint_gobble_i }%
    \fi
    {#1}%
}%
\def\XINT_fac_checklength #1%
{%
    \ifnum #1>100000
         \xint_afterfi{\expandafter\xintError:FactorialOfTooBigNumber
                        \expandafter\space\expandafter 1\xint_gobble_i }%
    \else
         \xint_afterfi{\ifnum #1>\xint_c_ixixixix
                           \expandafter\XINT_fac_big_loop
                        \else
                           \expandafter\XINT_fac_loop
                        \fi }%
    \fi
    {#1}%
}%
```

```
\def\XINT_fac_big_loop #1{\XINT_fac_big_loop_main {10000}{#1}{}}%
\def\XINT_fac_big_loop_main #1#2#3%
{%
    \ifnum #1<#2
        \expandafter
            \XINT_fac_big_loop_main
        \expandafter
           {\the\numexpr #1+1\expandafter }%
    \else
        \expandafter\XINT_fac_big_docomputation
    \fi
    {#2}{#3{#1}}%
}%
\def\XINT_fac_big_docomputation #1#2%
{%
    \expandafter \XINT_fac_bigcompute_loop \expandafter
    {\romannumeral0\XINT_fac_loop {9999}}#2\relax
}%
\def\XINT_fac_bigcompute_loop #1#2%
{%
    \xint_gob_til_relax #2\XINT_fac_bigcompute_end\relax
    \expandafter\XINT_fac_bigcompute_loop\expandafter
    {\expandafter\XINT_mul_enter
     \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
     \Z\Z\Z\Z #1\W\W\W\W }%
}%
\def\XINT_fac_bigcompute_end #1#2#3#4#5%
{%
    \XINT_fac_bigcompute_end_ #5%
}%
\def\XINT_fac_bigcompute_end_ #1\R #2\Z \W\X\Y\Z #3\W\X\Y\Z { #3}%
\def\XINT_fac_loop #1{\XINT_fac_loop_main 1{1000}{#1}}%
\def\XINT_fac_loop_main #1#2#3%
{%
    \ifnum #3>#1
    \else
        \expandafter\XINT_fac_loop_exit
    \fi
    \expandafter\XINT_fac_loop_main\expandafter
    {\the\numexpr #1+1\expandafter }\expandafter
    {\romannumeral0\XINT_mul_Mr {#1}#2\Z\Z\Z\Z }%
    {#3}%
}%
\def\XINT_fac_loop_exit #1#2#3#4#5#6#7%
{%
    \XINT_fac_loop_exit_ #6%
}%
\def\XINT_fac_loop_exit_ #1#2#3%
{%
```

```
    \XINT_mul_M
}%
```

32.47 \xintPow

1.02 modified the \XINT_posprod routine, the was renamed \XINT_pow_posprod and moved here, as it was well adapted for computing powers. Then 1.03 moved the special variants of multiplication (hence of addition) which were needed to earlier in this style file.

Modified in 1.06, the exponent is given to a \numexpr rather than twice expanded. \xintnum added in 1.09a.

\XINT_pow_posprod: Routine de produit servant pour le calcul des puissances. Chaque nouveau terme est plus grand que ce qui a déjà été calculé. Par conséquent on a intérêt à le conserver en second dans la routine de multiplication, donc le précédent calcul a intérêt à avoir été donné sur 4n, à l'envers. Il faut donc modifier la multiplication pour qu'elle fasse cela. Ce qui oblige à utiliser une version spéciale de l'addition également.

1.09j has reorganized the main loop, the described above \XINT_pow_posprod routine has been removed, intermediate multiplications are done immediately. Also, the maximal accepted exponent is now 100000 (no such restriction in \xintFloatPow, which accepts any exponent less than 2^31, and in \xintFloatPower which accepts long integers as exponent).

2^100000=9.990020930143845e30102 and multiplication of two numbers with 30000 digits would take hours on my laptop (seconds for 1000 digits).

```
\def\xintiiPow {\romannumeral0\xintiipow }%
\def\xintiipow #1%
{%
    \expandafter\xint_pow\romannumeral-`0#1\Z%
}%
\def\xintiPow {\romannumeral0\xintipow }%
\def\xintipow #1%
{%
    \expandafter\xint_pow\romannumeral0\xintnum{#1}\Z%
}%
\let\xintPow\xintiPow \let\xintpow\xintipow
\def\xint_pow #1#2\Z
{%
    \xint_UDsignfork
      #1\XINT_pow_Aneg
       -\XINT_pow_Anonneg
    \krof
       #1{#2}%
}%
\def\XINT_pow_Aneg #1#2#3%
{%
   \expandafter\XINT_pow_Aneg_\expandafter{\the\numexpr #3}{#2}%
}%
\def\XINT_pow_Aneg_ #1%
```

```
{%
  \ifodd #1
      \expandafter\XINT_pow_Aneg_Bodd
  \fi
  \XINT_pow_Anonneg_ {#1}%
}%
\def\XINT_pow_Aneg_Bodd #1%
{%
   \expandafter\XINT_opp\romannumeral0\XINT_pow_Anonneg_
}%
```

B = #3, faire le xpxp. Modified with 1.06: use of \numexpr.

```
\def\XINT_pow_Anonneg #1#2#3%
{%
  \expandafter\XINT_pow_Anonneg_\expandafter {\the\numexpr #3}{#1#2}%
}%
```

#1 = B, #2 = |A|

```
\def\XINT_pow_Anonneg_ #1#2%
{%
   \ifcase\XINT_Cmp {#2}{1}
       \expandafter\XINT_pow_AisOne
   \or
       \expandafter\XINT_pow_AatleastTwo
   \else
       \expandafter\XINT_pow_AisZero
   \fi
   {#1}{#2}%
}%
\def\XINT_pow_AisOne #1#2{ 1}%
```

#1 = B

```
\def\XINT_pow_AisZero #1#2%
{%
    \ifcase\XINT_cntSgn #1\Z
        \xint_afterfi { 1}%
    \or
        \xint_afterfi { 0}%
    \else
        \xint_afterfi {\xintError:DivisionByZero\space 0}%
    \fi
}%
\def\XINT_pow_AatleastTwo #1%
{%
   \ifcase\XINT_cntSgn #1\Z
       \expandafter\XINT_pow_BisZero
   \or
```

```
        \expandafter\XINT_pow_checkBsize
    \else
        \expandafter\XINT_pow_BisNegative
    \fi
    {#1}%
}%
\edef\XINT_pow_BisNegative #1#2%
    {\noexpand\xintError:FractionRoundedToZero\space 0}%
\def\XINT_pow_BisZero #1#2{ 1}%
```

B = #1 > 0, A = #2 > 1. With 1.05, I replace \xintiLen{#1}>9 by direct use of \numexpr [to generate an error message if the exponent is too large] 1.06: \numexpr was already used above.

```
\def\XINT_pow_checkBsize #1%
{%
    \ifnum #1>100000
        \expandafter\XINT_pow_BtooBig
    \else
        \expandafter\XINT_pow_loopI
    \fi
    {#1}%
}%
\edef\XINT_pow_BtooBig #1#2{\noexpand\xintError:ExponentTooBig\space 0}%
\def\XINT_pow_loopI #1%
{%
    \ifnum #1=\xint_c_i\XINT_pow_Iend\fi
    \ifodd #1
       \expandafter\XINT_pow_loopI_odd
    \else
       \expandafter\XINT_pow_loopI_even
    \fi
    {#1}%
}%
\edef\XINT_pow_Iend\fi #1\fi #2#3{\noexpand\fi\space #3}%
\def\XINT_pow_loopI_even #1#2%
{%
    \expandafter\XINT_pow_loopI\expandafter
    {\the\numexpr #1/\xint_c_ii\expandafter}\expandafter
    {\romannumeral0\xintiisqr {#2}}%
}%
\def\XINT_pow_loopI_odd #1#2%
{%
    \expandafter\XINT_pow_loopI_odda\expandafter
    {\romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z }{#1}{#2}%
}%
\def\XINT_pow_loopI_odda #1#2#3%
{%
    \expandafter\XINT_pow_loopII\expandafter
    {\the\numexpr #2/\xint_c_ii-\xint_c_i\expandafter}\expandafter
```

```
    {\romannumeral0\xintiisqr {#3}}{#1}%
}%
\def\XINT_pow_loopII #1%
{%
    \ifnum #1 = \xint_c_i\XINT_pow_IIend\fi
    \ifodd #1
       \expandafter\XINT_pow_loopII_odd
    \else
       \expandafter\XINT_pow_loopII_even
    \fi
    {#1}%
}%
\def\XINT_pow_loopII_even #1#2%
{%
    \expandafter\XINT_pow_loopII\expandafter
    {\the\numexpr #1/\xint_c_ii\expandafter}\expandafter
    {\romannumeral0\xintiisqr {#2}}%
}%
\def\XINT_pow_loopII_odd #1#2#3%
{%
    \expandafter\XINT_pow_loopII_odda\expandafter
    {\romannumeral0\XINT_mulr_enter #3\Z\Z\Z\Z #2\W\W\W\W}{#1}{#2}%
}%
\def\XINT_pow_loopII_odda #1#2#3%
{%
    \expandafter\XINT_pow_loopII\expandafter
    {\the\numexpr #2/\xint_c_ii-\xint_c_i\expandafter}\expandafter
    {\romannumeral0\xintiisqr {#3}}{#1}%
}%
\def\XINT_pow_IIend\fi #1\fi #2#3#4%
{%
    \fi\XINT_mul_enter #4\Z\Z\Z\Z #3\W\W\W\W
}%
```

32.48 \xintDivision, \xintQuo, \xintRem

The 1.09a release inserted the use of \xintnum. The \xintiiDivision etc... are the ones which do only \romannumeral-‘0.

January 5, 2014: Naturally, addition, subtraction, multiplication and division are the first things I did and since then I had left the division untouched. So in preparation of release 1.09j, I started revisiting the division, I did various minor improvements obtaining roughly 10% efficiency gain. Then I decided I should deliberately impact the input save stack, with the hope to gain more speed from removing tokens and leaving them upstream.

For this however I had to modify the underlying mathematical algorithm. The initial one is a bit unusual I guess, and, I trust, rather efficient, but it does not produce the quotient digits (in base 10000) one by one; at any given time it is possible that some correction will be made, which means it is not an appropriate algorithm for a TeX implementation which will abandon the quotient upstream. Thus

I now have with 1.09j a new underlying mathematical algorithm, presumably much more standard. It is a bit complicated to implement expandably these things, but in the end I had regained the already mentioned 10% efficiency and even more for small to medium sized inputs (up to 30% perhaps). And in passing I did a special routine for divisors < 10000, which is 5 to 10 times faster still.

But, I then tested a variant of my new implementation which again did not impact the input save stack and, for sizes of up to 200 digits, it is not much worse, indeed it is perhaps actually better than the one abandoning the quotient digits upstream (and in the end putting them in the correct order). So, finally, I re-incorporated the produced quotient digits within a tail recursion. Hence \xintDivision, like all other routines in xint (except \xintSeq without optional parameter) still does not impact the input save stack. One can have a produced quotient longer than 4x5000=20000 digits, and no need to worry about \xintTrunc, \xintRound, \xintFloat, \xintFloatSqrt, etc... and all other places using the division.

However outputting to a file (which is basically the only thing one can do, multiplying out two 20000 digits numbers already takes hours, for 100000 it would be days if not weeks) 100000 digits is slow... the truncation routine will add 100000 zeros (circa) and then trim them four by four. Definitely I should do a routine XTrunc which will work by blocks of say 64, and furthermore, being destined to be used in and \edef or a \write, it could be much more efficient as it could simply be based on tail loop, which so far nothing in xint does because I want things to expand fully under \romannumeral-‘0 (and don’t imagine inserting chains of thousands of \expandafter’s...) in order to be nestable. Inside \xintexpr such style of tail recursion leaving downstream things should definitely be implemented for the routines for which it is possible as things get expanded inside \csname..\endcsname. I don’t do yet anything like this for 1.09j.

```
\def\xintiiQuo {\romannumeral0\xintiiquo }%
\def\xintiiRem {\romannumeral0\xintiirem }%
\def\xintiiquo {\expandafter\xint_firstoftwo_thenstop
                                       \romannumeral0\xintiidivision }%
\def\xintiirem {\expandafter\xint_secondoftwo_thenstop
                                       \romannumeral0\xintiidivision }%
\def\xintQuo {\romannumeral0\xintquo }%
\def\xintRem {\romannumeral0\xintrem }%
\def\xintquo {\expandafter\xint_firstoftwo_thenstop
                                       \romannumeral0\xintdivision }%
\def\xintrem {\expandafter\xint_secondoftwo_thenstop
                                       \romannumeral0\xintdivision }%
```

#1 = A, #2 = B. On calcule le quotient et le reste dans la division euclidienne de A par B.

```
\def\xintiiDivision {\romannumeral0\xintiidivision }%
\def\xintiidivision #1%
{%
    \expandafter\xint_iidivision\expandafter {\romannumeral-‘0#1}%
}%
\def\xint_iidivision #1#2%
```

```
{%
    \expandafter\XINT_div_fork \romannumeral-`0#2\Z #1\Z
}%
\def\xintDivision {\romannumeral0\xintdivision }%
\def\xintdivision #1%
{%
    \expandafter\xint_division\expandafter {\romannumeral0\xintnum{#1}}%
}%
\def\xint_division #1#2%
{%
    \expandafter\XINT_div_fork \romannumeral0\xintnum{#2}\Z #1\Z
}%
```

#1#2 = 2e input = diviseur = B. #3#4 = 1er input = divisé = A.

```
\def\XINT_div_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\XINT_div_BisZero
      #3\XINT_div_AisZero
       0{\xint_UDsignfork
           #1\XINT_div_BisNegative  % B < 0
           #3\XINT_div_AisNegative  % A < 0, B > 0
            -\XINT_div_plusplus     % B > 0, A > 0
         \krof }%
    \krof
    {#2}{#4}#1#3% #1#2=B, #3#4=A
}%
\edef\XINT_div_BisZero #1#2#3#4{\noexpand\xintError:DivisionByZero\space {0}{0}}%
\def\XINT_div_AisZero #1#2#3#4{ {0}{0}}%
```

jusqu'à présent c'est facile.
minusplus signifie B < 0, A > 0
plusminus signifie B > 0, A < 0
Ici #3#1 correspond au diviseur B et #4#2 au divisé A.

Cases with B<0 or especially A<0 are treated sub-optimally in terms of post-processing, things get reversed which could have been produced directly in the wanted order, but A,B>0 is given priority for optimization. I should revise the next few macros, definitely.

```
\def\XINT_div_plusplus #1#2#3#4{\XINT_div_prepare {#3#1}{#4#2}}%
```

B = #3#1 < 0, A non nul positif ou négatif

```
\def\XINT_div_BisNegative #1#2#3#4%
{%
    \expandafter\XINT_div_BisNegative_b
    \romannumeral0\XINT_div_fork #1\Z #4#2\Z
}%
\edef\XINT_div_BisNegative_b #1%
```

```
{%
    \noexpand\expandafter\space\noexpand\expandafter
     {\noexpand\romannumeral0\noexpand\XINT_opp #1}%
}%
```

B = #3#1 > 0, A =-#2< 0

```
\def\XINT_div_AisNegative #1#2#3#4%
{%
    \expandafter\XINT_div_AisNegative_b
    \romannumeral0\XINT_div_prepare {#3#1}{#2}{#3#1}%
}%
\def\XINT_div_AisNegative_b #1#2%
{%
    \if0\XINT_Sgn #2\Z
       \expandafter \XINT_div_AisNegative_Rzero
    \else
       \expandafter \XINT_div_AisNegative_Rpositive
    \fi
    {#1}{#2}%
}%
```

en #3 on a une copie de B (à l'endroit)

```
\edef\XINT_div_AisNegative_Rzero #1#2#3%
{%
    \noexpand\expandafter\space\noexpand\expandafter
    {\noexpand\romannumeral0\noexpand\XINT_opp #1}{0}%
}%
```

#1 = quotient, #2 = reste, #3 = diviseur initial (à l'endroit) remplace Reste par B - Reste, après avoir remplacé Q par -(Q+1) de sorte que la formule a = qb + r, 0<= r < |b| est valable

```
\def\XINT_div_AisNegative_Rpositive #1%
{%
    \expandafter \XINT_div_AisNegative_Rpositive_b \expandafter
       {\romannumeral0\xintiiopp{\xintInc {#1}}}%
}%
\def\XINT_div_AisNegative_Rpositive_b #1#2#3%
{%
    \expandafter \xint_exchangetwo_keepbraces_thenstop \expandafter
        {\romannumeral0\XINT_sub {#3}{#2}}{#1}%
}%
```

Pour la suite A et B sont > 0. #1 = B. Pour le moment à l'endroit. Calcul du plus petit K = 4n >= longueur de B

```
\def\XINT_div_prepare #1%
{%
```

```
    \expandafter \XINT_div_prepareB_aa \expandafter
        {\romannumeral0\xintlength {#1}}{#1}% B > 0 ici
}%
\def\XINT_div_prepareB_aa #1%
{%
    \ifnum #1=\xint_c_i
      \expandafter\XINT_div_prepareB_onedigit
    \else
      \expandafter\XINT_div_prepareB_a
    \fi
    {#1}%
}%
\def\XINT_div_prepareB_a #1%
{%
  \expandafter\XINT_div_prepareB_c\expandafter
  {\the\numexpr \xint_c_iv*((#1+\xint_c_i)/\xint_c_iv)}{#1}%
}%
```

B=1 and B=2 treated specially.

```
\def\XINT_div_prepareB_onedigit #1#2%
{%
    \ifcase#2
    \or\expandafter\XINT_div_BisOne
    \or\expandafter\XINT_div_BisTwo
    \else\expandafter\XINT_div_prepareB_e
    \fi {000}{0}{4}{#2}%
}%
\def\XINT_div_BisOne #1#2#3#4#5{ {#5}{0}}%
\def\XINT_div_BisTwo #1#2#3#4#5%
{%
    \expandafter\expandafter\expandafter\XINT_div_BisTwo_a
    \ifodd\xintiiLDg{#5} \expandafter1\else \expandafter0\fi {#5}%
}%
\edef\XINT_div_BisTwo_a #1#2%
{%
    \noexpand\expandafter\space\noexpand\expandafter
    {\noexpand\romannumeral0\noexpand\xinthalf {#2}}{#1}%
}%
```

#1 = K. 1.09j uses \csname, earlier versions did it with \ifcase.

```
\def\XINT_div_prepareB_c #1#2%
{%
    \csname XINT_div_prepareB_d\romannumeral\numexpr#1-#2\endcsname
    {#1}%
}%
\def\XINT_div_prepareB_d    {\XINT_div_prepareB_e {}{0000}}%
\def\XINT_div_prepareB_di   {\XINT_div_prepareB_e {0}{000}}%
\def\XINT_div_prepareB_dii  {\XINT_div_prepareB_e {00}{00}}%
```

```
\def\XINT_div_prepareB_diii {\XINT_div_prepareB_e {000}{0}}%
\def\XINT_div_cleanR         #10000.{{#1}}%
```

#1 = zéros à rajouter à B, #2=c [modifié dans 1.09j, ce sont maintenant des zéros explicites en nombre 4 - ancien c, et on utilisera \XINT_div_cleanR et non plus \XINT_dsh_checksignx pour nettoyer à la fin des zéros en excès dans le Reste; in all comments next, «c» stands now {0} or {00} or {000} or {0000} rather than a digit as in earlier versions], #3=K, #4 = B

```
\def\XINT_div_prepareB_e #1#2#3#4%
{%
    \ifnum#3=\xint_c_iv\expandafter\XINT_div_prepareLittleB_f
                       \else\expandafter\XINT_div_prepareB_f
    \fi
    #4#1{#3}{#2}{#1}%
}%
```

x = #1#2#3#4 = 4 premiers chiffres de B. #1 est non nul. B is reversed. With 1.09j or latter x+1 and (x+1)/2 are pre-computed. Si K=4 on ne renverse pas B, et donc B=x dans la suite. De plus pour K=4 on ne travaille pas avec x+1 et (x+1)/2 mais avec x et x/2.

```
\def\XINT_div_prepareB_f #1#2#3#4#5#{%
    \expandafter\XINT_div_prepareB_g
     \the\numexpr  #1#2#3#4+\xint_c_i\expandafter
    .\the\numexpr (#1#2#3#4+\xint_c_i)/\xint_c_ii\expandafter
    .\romannumeral0\xintreverseorder {#1#2#3#4#5}.{#1#2#3#4}%
}%
\def\XINT_div_prepareLittleB_f #1#{%
    \expandafter\XINT_div_prepareB_g \the\numexpr  #1/\xint_c_ii.{}.{}.{#1}%
}%
```

#1 = x' = x+1= 1+quatre premiers chiffres de B, #2 = y = (x+1)/2 précalculé #3 = B préparé et maintenant renversé, #4=x, #5 = K, #6 = «c», #7= {} ou {0} ou {00} ou {000}, #8 = A initial On multiplie aussi A par 10^c. -> AK{x'yx}B«c». Par contre dans le cas little on a #1=y=(x/2), #2={}, #3={}, #4=x, donc cela donne ->AK{y{}x}{}«c», il n'y a pas de B.

```
\def\XINT_div_prepareB_g #1.#2.#3.#4#5#6#7#8%
{%
    \XINT_div_prepareA_a {#8#7}{#5}{{#1}{#2}{#4}}{#3}{#6}%
}%
```

A, K, {x'yx}, B«c»

```
\def\XINT_div_prepareA_a #1%
{%
    \expandafter\XINT_div_prepareA_b\expandafter
       {\romannumeral0\xintlength {#1}}{#1}%
}%
```

L0, A, K, {x'yx}, B«c»

```
\def\XINT_div_prepareA_b #1%
{%
  \expandafter\XINT_div_prepareA_c\expandafter
    {\the\numexpr \xint_c_iv*((#1+\xint_c_i)/\xint_c_iv)}{#1}%
}%
```

L, L0, A, K, {x'yx}, B, «c»

```
\def\XINT_div_prepareA_c #1#2%
{%
    \csname XINT_div_prepareA_d\romannumeral\numexpr #1-#2\endcsname
    {#1}%
}%
\def\XINT_div_prepareA_d     {\XINT_div_prepareA_e {}}%
\def\XINT_div_prepareA_di    {\XINT_div_prepareA_e {0}}%
\def\XINT_div_prepareA_dii   {\XINT_div_prepareA_e {00}}%
\def\XINT_div_prepareA_diii  {\XINT_div_prepareA_e {000}}%
```

#1#3 = A préparé, #2 = longueur de ce A préparé, #4=K, #5={x'yx}-> LKAx'yxB«c»

```
\def\XINT_div_prepareA_e #1#2#3#4#5%
{%
    \XINT_div_start_a {#2}{#4}{#1#3}#5%
}%
```

L, K, A, x',y,x, B, «c» (avec y{}x{} au lieu de x'yxB dans la variante little)

```
\def\XINT_div_start_a #1#2%
{%
    \ifnum #2=\xint_c_iv \expandafter\XINT_div_little_b
    \else
      \ifnum #1 < #2
        \expandafter\expandafter\expandafter\XINT_div_III_aa
      \else
        \expandafter\expandafter\expandafter\XINT_div_start_b
      \fi
    \fi
    {#1}{#2}%
}%
```

L, K, A, x',y,x, B, «c».

```
\def\XINT_div_III_aa #1#2#3#4#5#6#7%
{%
    \expandafter\expandafter\expandafter
    \XINT_div_III_b\xint_cleanupzeros_nostop #3.{0000}%
}%
```

R.Q«c».

```
\def\XINT_div_III_b #1%
{%
    \if0#1%
       \expandafter\XINT_div_III_bRzero
    \else
       \expandafter\XINT_div_III_bRpos
    \fi
    #1%
}%
\def\XINT_div_III_bRzero 0.#1#2%
{%
    \expandafter\space\expandafter
   {\romannumeral0\XINT_cuz_loop #1\W\W\W\W\W\W\W\Z}{0}%
}%
\def\XINT_div_III_bRpos #1.#2#3%
{%
    \expandafter\XINT_div_III_c \XINT_div_cleanR #1#3.{#2}%
}%
\def\XINT_div_III_c #1#2%
{%
    \expandafter\space\expandafter
   {\romannumeral0\XINT_cuz_loop #2\W\W\W\W\W\W\W\Z}{#1}%
}%
```

L, K, A, x',y,x, B, «c»->K.A.x{LK{x'y}x}B«c»

```
\def\XINT_div_start_b #1#2#3#4#5#6%
{%
    \XINT_div_start_c {#2}.#3.{#6}{{#1}{#2}{{#4}{#5}}{#6}}%
}%
```

Kalpha.A.x{LK{x'y}x}, B, «c», au début #2=alpha est vide

```
\def\XINT_div_start_c #1#2.#3#4#5#6%
{%
    \ifnum #1=\xint_c_iv\XINT_div_start_ca\fi
    \expandafter\XINT_div_start_c\expandafter
         {\the\numexpr #1-\xint_c_iv}#2#3#4#5#6.%
}%
\def\XINT_div_start_ca\fi\expandafter\XINT_div_start_c\expandafter
    #1#2#3#4#5{\fi\XINT_div_start_d {#2#3#4#5}#2#3#4#5}%
```

#1=a, #2=alpha (de longueur K, à l'endroit).#3=reste de A.#4=x, #5={LK{x'y}x},#6=B,«c» -> a, x, alpha, B, {0000}, L, K, {x'y},x, alpha'=reste de A, B{}«c». Pour K=4 on a en fait B=x, faudra revoir après.

```
\def\XINT_div_start_d #1#2.#3.#4#5#6%
{%
```

```
    \XINT_div_I_a {#1}{#4}{#2}{#6}{0000}#5{#3}{#6}{}%
}%
```

Ceci est le point de retour de la boucle principale. a, x, alpha, B, q0, L, K, {x'y}, x, alpha', BQ«c»

```
\def\XINT_div_I_a #1#2%
{%
    \expandafter\XINT_div_I_b\the\numexpr #1/#2.{#1}{#2}%
}%
\def\XINT_div_I_b #1%
{%
    \xint_gob_til_zero #1\XINT_div_I_czero 0\XINT_div_I_c #1%
}%
```

On intercepte quotient nul: #1=a, x, alpha, B, #5=q0, L, K, {x'y}, x, alpha', BQ«c» -> q{alpha} L, K, {x'y}, x, alpha', BQ«c»

```
\def\XINT_div_I_czero 0%
    \XINT_div_I_c 0.#1#2#3#4#5{\XINT_div_I_g {#5}{#3}}%
\def\XINT_div_I_c #1.#2#3%
{%
    \expandafter\XINT_div_I_da\the\numexpr #2-#1*#3.#1.%
}%
```

r.q.alpha, B, q0, L, K, {x'y}, x, alpha', BQ«c»

```
\def\XINT_div_I_da #1.%
{%
    \ifnum #1>\xint_c_ix
       \expandafter\XINT_div_I_dP
    \else
       \ifnum #1<\xint_c_
        \expandafter\expandafter\expandafter\XINT_div_I_dN
       \else
        \expandafter\expandafter\expandafter\XINT_div_I_db
       \fi
    \fi
}%
\def\XINT_div_I_dN #1.%
{%
    \expandafter\XINT_div_I_dP\the\numexpr #1-\xint_c_i.%
}%
\def\XINT_div_I_db #1.#2#3% #1=q=un chiffre, #2=alpha, #3=B
{%
    \expandafter\XINT_div_I_dc\expandafter
    {\romannumeral0\expandafter\XINT_div_sub_xpxp\expandafter
       {\romannumeral0\xintreverseorder{#2}}%
       {\romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z }}%
    #1{#2}{#3}%
```

```
}%
\def\XINT_div_I_dc #1#2%
{%
    \if-#1% s'arranger pour que si négatif on ait renvoyé alpha=-.
        \expandafter\xint_firstoftwo
    \else\expandafter\xint_secondoftwo\fi
    {\expandafter\XINT_div_I_dP\the\numexpr #2-\xint_c_i.}%
    {\XINT_div_I_e {#1}#2}%
}%
```

alpha,q,ancien alpha,B, q0->1nouveauq.alpha, L, K, {x'y},x, alpha', BQ«c»

```
\def\XINT_div_I_e #1#2#3#4#5%
{%
    \expandafter\XINT_div_I_f \the\numexpr \xint_c_x^iv+#2+#5{#1}%
}%
```

q.alpha, B, q0, L, K, {x'y},x, alpha'BQ«c» (intercepter q=0?) -> 1nouveauq.nouvel alpha, L, K, {x'y}, x, alpha',BQ«c»

```
\def\XINT_div_I_dP #1.#2#3#4%
{%
    \expandafter \XINT_div_I_f \the\numexpr \xint_c_x^iv+#1+#4\expandafter
    {\romannumeral0\expandafter\XINT_div_sub_xpxp\expandafter
      {\romannumeral0\xintreverseorder{#2}}%
      {\romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z }}%
}%
```

1#1#2#3#4=nouveau q, nouvel alpha, L, K, {x'y},x,alpha', BQ«c»

```
\def\XINT_div_I_f 1#1#2#3#4{\XINT_div_I_g {#1#2#3#4}}%
```

#1=q,#2=nouvel alpha,#3=L, #4=K, #5={x'y}, #6=x, #7= alpha',#8=B, #9=Q«c» -> {x'y}alpha.alpha'.{{x'y}xKL}B{Qq}«c»

```
\def\XINT_div_I_g #1#2#3#4#5#6#7#8#9%
{%
     \ifnum#3=#4
          \expandafter\XINT_div_III_ab
     \else
          \expandafter\XINT_div_I_h
     \fi
     {#5}#2.#7.{{#5}{#6}{#4}{#3}}{#8}{#9#1}%
}%
```

{x'y}alpha.alpha'.{{x'y}xKL}B{Qq}«c» -> R sans leading zeros.{Qq}«c»

```
\def\XINT_div_III_ab #1#2.#3.#4#5%
{%
    \expandafter\XINT_div_III_b
    \romannumeral0\XINT_cuz_loop #2#3\W\W\W\W\W\W\W\Z.%
}%
```

#1={x'y}alpha.#2#3#4#5#6=reste de A. #7={{x'y},x,K,L},#8=B,nouveauQ«c» devient {x'y},alpha sur K+4 chiffres.B, {{x'y},x,K,L}, #6= nouvel alpha',B,nouveauQ«c»

```
\def\XINT_div_I_h #1.#2#3#4#5#6.#7#8%
{%
    \XINT_div_II_b #1#2#3#4#5.{#8}{#7}{#6}{#8}%
}%
```

{x'y}alpha.B, {{x'y},x,K,L}, nouveau alpha',B, Q«c» On intercepte la situation avec alpha débutant par 0000 qui est la seule qui pourrait donner un q1 nul. Donc q1 est non nul et la soustraction spéciale recevra un q1*B de longueur K ou K+4 et jamais 0000. Ensuite un q2 éventuel s'il est calculé est nécessairement non nul lui aussi. Comme dans la phase I on a aussi intercepté un q nul, la soustraction spéciale ne reçoit donc jamais un qB nul. Note: j'ai testé plusieurs fois que ma technique de gob_til_zeros est plus rapide que d'utiliser un \ifnum

```
\def\XINT_div_II_b #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_zeros_iv #2#3#4#5\XINT_div_II_skipc 0000%
    \XINT_div_II_c #1{#2#3#4#5}{#6#7#8#9}%
}%
```

x'y{0000}{4chiffres}reste de alpha.#6=B,#7={{x'y},x,K,L}, alpha',B, Q«c» -> {x'y}x,K,L (à diminuer de 4), {alpha sur K}B{q1=0000}{alpha'}B,Q«c»

```
\def\XINT_div_II_skipc 0000\XINT_div_II_c #1#2#3#4#5.#6#7%
{%
    \XINT_div_II_k #7{#4#5}{#6}{0000}%
}%
```

x'ya->1qx'yalpha.B, {{x'y},x,K,L}, nouveau alpha',B, Q«c»

```
\def\XINT_div_II_c #1#2#3#4%
{%
     \expandafter\XINT_div_II_d\the\numexpr (#3#4+#2)/#1+\xint_c_ixixixix\relax
     {#1}{#2}#3#4%
}%
```

1 suivi de q1 sur quatre chiffres, #5=x', #6=y, #7=alpha.#8=B, {{x'y},x,K,L}, alpha', B, Q«c» --> nouvel alpha.x',y,B,q1,{{x'y},x,K,L}, alpha', B, Q«c»

```
\def\XINT_div_II_d 1#1#2#3#4#5#6#7.#8%
{%
    \expandafter\XINT_div_II_e
    \romannumeral0\expandafter\XINT_div_sub_xpxp\expandafter
      {\romannumeral0\xintreverseorder{#7}}%
      {\romannumeral0\XINT_mul_Mr {#1#2#3#4}#8\Z\Z\Z\Z }.%
    {#5}{#6}{#8}{#1#2#3#4}%
}%
```

alpha.x',y,B,q1, {{x'y},x,K,L}, alpha', B, Q«c»

```
\def\XINT_div_II_e #1#2#3#4%
{%
    \xint_gob_til_zeros_iv #1#2#3#4\XINT_div_II_skipf 0000%
    \XINT_div_II_f #1#2#3#4%
}%
```

0000alpha sur K chiffres.#2=x',#3=y,#4=B,#5=q1, #6={{x'y},x,K,L}, #7=alpha',BQ«c» -> {x'y}x,K,L (à diminuer de 4), {alpha sur K}B{q1}{alpha'}BQ«c»

```
\def\XINT_div_II_skipf 0000\XINT_div_II_f 0000#1.#2#3#4#5#6%
{%
    \XINT_div_II_k #6{#1}{#4}{#5}%
}%
```

a1 (huit chiffres), alpha (sur K+4), x', y, B, q1, {{x'y},x,K,L}, alpha', B,Q«c»

```
\def\XINT_div_II_f #1#2#3#4#5#6#7#8#9.%
{%
    \XINT_div_II_fa {#1#2#3#4#5#6#7#8}{#1#2#3#4#5#6#7#8#9}%
}%
\def\XINT_div_II_fa #1#2#3#4%
{%
    \expandafter\XINT_div_II_g\expandafter
                                {\the\numexpr (#1+#4)/#3-\xint_c_i}{#2}%
}%
```

#1=q, #2=alpha (K+4), #3=B, #4=q1, {{x'y},x,K,L}, alpha', BQ«c» -> 1 puis nouveau q sur 4 chiffres, nouvel alpha sur K chiffres, B, {{x'y},x,K,L}, alpha',BQ«c»

```
\def\XINT_div_II_g #1#2#3#4%
{%
    \expandafter \XINT_div_II_h
    \the\numexpr #4+#1+\xint_c_x^iv\expandafter\expandafter\expandafter
    {\expandafter\xint_gobble_iv
    \romannumeral0\expandafter\XINT_div_sub_xpxp\expandafter
       {\romannumeral0\xintreverseorder{#2}}%
       {\romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z }}{#3}%
}%
```

1 puis nouveau q sur 4 chiffres, #5=nouvel alpha sur K chiffres, #6=B, #7={{x'y},x,K,L} avec L à ajuster, alpha', BQ«c» -> {x'y}x,K,L à diminuer de 4, {alpha}B{q}, alpha', BQ«c»

```
\def\XINT_div_II_h 1#1#2#3#4#5#6#7%
{%
    \XINT_div_II_k #7{#5}{#6}{#1#2#3#4}%
}%
```

{x'y}x,K,L à diminuer de 4, alpha, B{q}alpha',BQ«c» ->nouveau L.K,x',y,x,alpha.B,q,alpha',B,Q«c» ->{LK{x'y}x},x,a,alpha.B,q,alpha',B,Q«c»

```
\def\XINT_div_II_k #1#2#3#4#5%
{%
    \expandafter\XINT_div_II_l \the\numexpr #4-\xint_c_iv.{#3}#1{#2}#5.%
}%
\def\XINT_div_II_l #1.#2#3#4#5#6#7#8#9%
{%
    \XINT_div_II_m {{#1}{#2}{{#3}{#4}}{#5}}{#5}{#6#7#8#9}#6#7#8#9%
}%
```

{LK{x'y}x},x,a,alpha.B{q}alpha'BQ -> a, x, alpha, B, q, L, K, {x'y}, x, alpha', BQ«c»

```
\def\XINT_div_II_m #1#2#3#4.#5#6%
{%
     \XINT_div_I_a {#3}{#2}{#4}{#5}{#6}#1%
}%
```

L, K, A, y,{},x, {},«c»->A.{yx}L{}«c» Comme ici K=4, dans la phase I on n'a pas besoin de alpha, car a = alpha. De plus on a maintenu B dans l'ordre qui est donc la même chose que x. Par ailleurs la phase I est simplifiée, il s'agit simplement de la division euclidienne de a par x, et de plus on n'a à la faire qu'une unique fois et ensuite la phase II peut boucler sur elle-même au lieu de revenir en phase I, par conséquent il n'y a pas non plus de q0 ici. Enfin, le y est (x/2) pas ((x+1)/2) il n'y a pas de x'=x+1

```
\def\XINT_div_little_b #1#2#3#4#5#6#7%
{%
    \XINT_div_little_c #3.{{#4}{#6}}{#1}%
}%
```

#1#2#3#4=a, #5=alpha'=reste de A.#6={yx}, #7=L, «c» -> a, y, x, L, alpha'=reste de A, «c».

```
\def\XINT_div_little_c #1#2#3#4#5.#6#7%
{%
    \XINT_div_littleI_a {#1#2#3#4}#6{#7}{#5}%
}%
```

a, y, x, L, alpha',«c» On calcule ici (contrairement à la phase I générale) le vrai quotient euclidien de a par x=B, c'est donc un chiffre de 0 à 9. De plus on n'a à faire cela qu'une unique fois.

```
\def\XINT_div_littleI_a #1#2#3%
{%
    \expandafter\XINT_div_littleI_b
    \the\numexpr (#1+#2)/#3-\xint_c_i{#1}{#2}{#3}%
}%
```

On intercepte quotient nul: [est-ce vraiment utile? ou n'est-ce pas plutôt une perte de temps en moyenne? il faudrait tester] q=0#1=a, #2=y, x, L, alpha', «c» -> II_a avec L{alpha}alpha'.{yx}{0000}«c». Et en cas de quotient non nul on procède avec littleI_c avec #1=q, #2=a, #3=y, #4=x -> {nouvel alpha sur 4 chiffres}q{yx},L,alpha',«c».

```
\def\XINT_div_littleI_b #1%
{%
    \xint_gob_til_zero #1\XINT_div_littleI_skip 0\XINT_div_littleI_c #1%
}%
\def\XINT_div_littleI_skip 0\XINT_div_littleI_c 0#1#2#3#4#5%
   {\XINT_div_littleII_a {#4}{#1}#5.{{#2}{#3}}{0000}}%
\def\XINT_div_littleI_c #1#2#3#4%
{%
    \expandafter\expandafter\expandafter\XINT_div_littleI_e
    \expandafter\expandafter\expandafter
    {\expandafter\xint_gobble_i\the\numexpr \xint_c_x^iv+#2-#1*#4}#1{{#3}{#4}}%
}%
```

#1=nouvel alpha sur 4 chiffres#2=q,#3={yx}, #4=L, #5=alpha',«c» -> L{alpha}alpha'.{yx}{000q}«c» point d'entrée de la boucle principale

```
\def\XINT_div_littleI_e #1#2#3#4#5%
   {\XINT_div_littleII_a {#4}{#1}#5.{#3}{000#2}}%
```

L{alpha}alpha'.{yx}Q«c» et c'est là qu'on boucle

```
\def\XINT_div_littleII_a #1%
{%
     \ifnum#1=\xint_c_iv
          \expandafter\XINT_div_littleIII_ab
     \else
          \expandafter\XINT_div_littleII_b
     \fi {#1}%
}%
```

L{alpha}alpha'.{yx}Q«c» -> (en fait #3 est vide normalement ici) R sans leading zeros.Q«c»

```
\def\XINT_div_littleIII_ab #1#2#3.#4%
{%
    \expandafter\XINT_div_III_b\the\numexpr #2#3.%
}%
```

L{alpha}alpha'.{yx}Q«c». On diminue L de quatre, comme cela c'est fait.

```
\def\XINT_div_littleII_b #1%
{%
   \expandafter\XINT_div_littleII_c\expandafter {\the\numexpr #1-\xint_c_iv}%
}%
```

{nouveauL}{alpha}alpha'.{yx}Q«c». On prélève 4 chiffres de alpha' -> {nouvel alpha sur huit chiffres}yx{nouveau L}{nouvel alpha'}Q«c». Regarder si l'ancien alpha était 0000 n'avancerait à rien car obligerait à refaire une chose comme la phase I, donc on ne perd pas de temps avec ça, on reste en permanence en phase II.

```
\def\XINT_div_littleII_c #1#2#3#4#5#6#7.#8%
{%
    \XINT_div_littleII_d {#2#3#4#5#6}#8{#1}{#7}%
}%
\def\XINT_div_littleII_d #1#2#3%
{%
    \expandafter\XINT_div_littleII_e\the\numexpr (#1+#2)/#3+\xint_c_ixixixix.%
    {#1}{#2}{#3}%
}%
```

1 suivi de #1=q1 sur quatre chiffres.#2=alpha, #3=y, #4=x, L, alpha', Q«c» --> nouvel alpha sur 4.{q1}{yx},L,alpha', Q«c»

```
\def\XINT_div_littleII_e 1#1.#2#3#4%
{%
    \expandafter\expandafter\expandafter\XINT_div_littleII_f
    \expandafter\xint_gobble_i\the\numexpr \xint_c_x^iv+#2-#1*#4.%
    {#1}{{#3}{#4}}%
}%
```

alpha.q,{yx},L,alpha',Q«c»->L{alpha}alpha'.{yx}{Qq}«c»

```
\def\XINT_div_littleII_f #1.#2#3#4#5#6%
{%
    \XINT_div_littleII_a {#4}{#1}#5.{#3}{#6#2}%
}%
```

La soustraction spéciale. Dans 1.09j, elle fait A-qB, pour A (en fait alpha dans mes dénominations des commentaires du code) et qB chacun de longueur K ou K+4, avec K au moins huit multiple de quatre, qB a ses quatre chiffres significatifs (qui sont à droite) non nuls. Si A-qB<0 il suffit de renvoyer -, le résultat n'importe pas. On est sûr que qB est non nul. On le met dans cette version en premier pour tester plus facilement le cas avec qB de longueur K+4 et A de longueur seulement K. Lorsque la longueur de qB est inférieure ou égale à celle de A, on va jusqu'à la fin de A et donc c'est la retenue finale qui décide du cas négatif éventuel. Le résultat non négatif est toujours donc renvoyé avec la même longueur que A, et il est dans l'ordre. J'ai fait une implémentation des phases I et II en maintenant alpha toujours à l'envers afin d'éviter le reverse order systématique fait sur A (ou plutôt alpha), mais alors il fallait que la soustraction ici s'arrange pour repérer les huit chiffres les plus significatifs, au final ce n'était pas plus rapide, et même pénalisant pour de gros inputs. Dans les versions 1.09i et antérieures (en fait je pense qu'ici rien quasiment n'avait bougé depuis la première implémentation), la soustraction spéciale n'était pratiquée que dans des cas avec certainement A-qB positif ou nul. De plus on n'excluait pas q=0, donc il fallait aussi faire un éventuel reverseorder sur ce qui était encore non traité.

Les cas avec q=0 sont maintenant interceptés en amont et comme A et qB ont toujours quasiment la même longueur on ne s'embarrasse pas de complications pour la fin.

```
\def\XINT_div_sub_xpxp #1#2% #1=alpha déjà renversé, #2 se développe en qB
{%
    \expandafter\XINT_div_sub_xpxp_b #2\W\X\Y\Z #1\W\X\Y\Z
}%
\def\XINT_div_sub_xpxp_b
{%
    \XINT_div_sub_A 1{}%
}%
\def\XINT_div_sub_A #1#2#3#4#5#6%
{%
    \xint_gob_til_W #3\xint_div_sub_az\W
    \XINT_div_sub_B #1{#3#4#5#6}{#2}%
}%
\def\XINT_div_sub_B #1#2#3#4\W\X\Y\Z #5#6#7#8%
{%
    \xint_gob_til_W #5\xint_div_sub_bz\W
    \XINT_div_sub_onestep #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
}%
\def\XINT_div_sub_onestep #1#2#3#4#5#6%
{%
    \expandafter\XINT_div_sub_backtoA
    \the\numexpr 11#6-#5#4#3#2+#1-\xint_c_i.%
}%
\def\XINT_div_sub_backtoA #1#2#3.#4%
{%
    \XINT_div_sub_A #2{#3#4}%
}%
```

si on arrive en sub_bz c'est que qB était de longueur K+4 et A seulement de longueur K, le résultat est donc < 0, renvoyer juste -

```
\def\xint_div_sub_bz\W\XINT_div_sub_onestep #1\Z { -}%
```

si on arrive en sub_az c'est que qB était de longueur inférieure ou égale à celle de A, donc on continue jusqu'à la fin de A, et on vérifiera la retenue à la fin.

```
\def\xint_div_sub_az\W\XINT_div_sub_B #1#2{\XINT_div_sub_C #1}%
\def\XINT_div_sub_C #1#2#3#4#5#6%
{%
    \xint_gob_til_W #3\xint_div_sub_cz\W
    \XINT_div_sub_C_onestep #1{#6#5#4#3}{#2}%
}%
\def\XINT_div_sub_C_onestep #1#2%
{%
    \expandafter\XINT_div_sub_backtoC \the\numexpr 11#2+#1-\xint_c_i.%
}%
\def\XINT_div_sub_backtoC #1#2#3.#4%
```

```
{%
    \XINT_div_sub_C #2{#3#4}%
}%
```

une fois arrivé en sub_cz on teste la retenue pour voir si le résultat final est en fait négatif, dans ce cas on renvoie seulement -

```
\def\xint_div_sub_cz\W\XINT_div_sub_C_onestep #1#2%
{%
    \if#10% retenue
         \expandafter\xint_div_sub_neg
    \else\expandafter\xint_div_sub_ok
    \fi
}%
\def\xint_div_sub_neg #1{ -}%
\def\xint_div_sub_ok  #1{ #1}%
```


DECIMAL OPERATIONS: FIRST DIGIT, LASTDIGIT, ODDNESS, MULTIPLICATION BY TEN, QUOTIENT BY TEN, QUOTIENT OR MULTIPLICATION BY POWER OF TEN, SPLIT OPERATION.

32.49 \xintFDg

FIRST DIGIT. Code simplified in 1.05. And prepared for redefinition by xintfrac to parse through \xintNum. Version 1.09a inserts the \xintnum already here.

```
\def\xintiiFDg {\romannumeral0\xintiifdg }%
\def\xintiifdg #1%
{%
    \expandafter\XINT_fdg \romannumeral-`0#1\W\Z
}%
\def\xintFDg {\romannumeral0\xintfdg }%
\def\xintfdg #1%
{%
    \expandafter\XINT_fdg \romannumeral0\xintnum{#1}\W\Z
}%
\def\XINT_FDg #1{\romannumeral0\XINT_fdg #1\W\Z }%
\def\XINT_fdg #1#2#3\Z
{%
    \xint_UDzerominusfork
      #1-{ 0}%   zero
      0#1{ #2}%  negative
       0-{ #1}%  positive
    \krof
}%
```

32.50 \xintLDg

LAST DIGIT. Simplified in 1.05. And prepared for extension by xintfrac to parse through \xintNum. Release 1.09a adds the \xintnum already here, and this propagates to \xintOdd, etc... 1.09e The \xintiiLDg is for defining \xintiiOdd which is used once (currently) elsewhere .

```
\def\xintiiLDg {\romannumeral0\xintiildg }%
\def\xintiildg #1%
{%
    \expandafter\XINT_ldg\expandafter {\romannumeral-`0#1}%
}%
\def\xintLDg {\romannumeral0\xintldg }%
\def\xintldg #1%
{%
    \expandafter\XINT_ldg\expandafter {\romannumeral0\xintnum{#1}}%
}%
\def\XINT_LDg #1{\romannumeral0\XINT_ldg {#1}}%
\def\XINT_ldg #1%
{%
    \expandafter\XINT_ldg_\romannumeral0\xintreverseorder {#1}\Z
}%
\def\XINT_ldg_ #1#2\Z{ #1}%
```

32.51 \xintMON, \xintMMON

MINUS ONE TO THE POWER N and (-1)^{N-1}

```
\def\xintiiMON {\romannumeral0\xintiimon }%
\def\xintiimon #1%
{%
    \ifodd\xintiiLDg {#1}
        \xint_afterfi{ -1}%
    \else
        \xint_afterfi{ 1}%
    \fi
}%
\def\xintiiMMON {\romannumeral0\xintiimmon }%
\def\xintiimmon #1%
{%
    \ifodd\xintiiLDg {#1}
        \xint_afterfi{ 1}%
    \else
        \xint_afterfi{ -1}%
    \fi
}%
\def\xintMON {\romannumeral0\xintmon }%
\def\xintmon #1%
{%
```

```
    \ifodd\xintLDg {#1}
        \xint_afterfi{ -1}%
    \else
        \xint_afterfi{ 1}%
    \fi
}%
\def\xintMMON {\romannumeral0\xintmmon }%
\def\xintmmon #1%
{%
    \ifodd\xintLDg {#1}
        \xint_afterfi{ 1}%
    \else
        \xint_afterfi{ -1}%
    \fi
}%
```

32.52 \xintOdd

1.05 has \xintiOdd, whereas \xintOdd parses through \xintNum. Inadvertently, 1.09a redefined \xintiLDg so \xintiOdd also parsed through \xintNum. Anyway, having a \xintOdd and a \xintiOdd was silly. Removed in 1.09f

```
\def\xintiiOdd {\romannumeral0\xintiiodd }%
\def\xintiiodd #1%
{%
    \ifodd\xintiiLDg{#1}
        \xint_afterfi{ 1}%
    \else
        \xint_afterfi{ 0}%
    \fi
}%
\def\xintOdd {\romannumeral0\xintodd }%
\def\xintodd #1%
{%
    \ifodd\xintLDg{#1}
        \xint_afterfi{ 1}%
    \else
        \xint_afterfi{ 0}%
    \fi
}%
```

32.53 \xintDSL

DECIMAL SHIFT LEFT (=MULTIPLICATION PAR 10)

```
\def\xintDSL {\romannumeral0\xintdsl }%
\def\xintdsl #1%
{%
```

```
    \expandafter\XINT_dsl \romannumeral-`0#1\Z
}%
\def\XINT_DSL #1{\romannumeral0\XINT_dsl #1\Z }%
\def\XINT_dsl #1%
{%
    \xint_gob_til_zero #1\xint_dsl_zero 0\XINT_dsl_ #1%
}%
\def\xint_dsl_zero 0\XINT_dsl_ 0#1\Z { 0}%
\def\XINT_dsl_ #1\Z { #10}%
```

32.54 \xintDSR

DECIMAL SHIFT RIGHT (=DIVISION PAR 10). Release 1.06b which replaced all @'s by underscores left undefined the \xint_minus used in \XINT_dsr_b, and this bug was fixed only later in release 1.09b

```
\def\xintDSR {\romannumeral0\xintdsr }%
\def\xintdsr #1%
{%
    \expandafter\XINT_dsr_a\expandafter {\romannumeral-`0#1}\W\Z
}%
\def\XINT_DSR #1{\romannumeral0\XINT_dsr_a {#1}\W\Z }%
\def\XINT_dsr_a
{%
    \expandafter\XINT_dsr_b\romannumeral0\xintreverseorder
}%
\def\XINT_dsr_b #1#2#3\Z
{%
    \xint_gob_til_W #2\xint_dsr_onedigit\W
    \xint_gob_til_minus #2\xint_dsr_onedigit-%
    \expandafter\XINT_dsr_removew
    \romannumeral0\xintreverseorder {#2#3}%
}%
\def\xint_dsr_onedigit #1\xintreverseorder #2{ 0}%
\def\XINT_dsr_removew #1\W { }%
```

32.55 \xintDSH, \xintDSHr

DECIMAL SHIFTS \xintDSH {x}{A}
si x <= 0, fait A -> A.10^(|x|). v1.03 corrige l'oversight pour A=0.
si x > 0, et A >=0, fait A -> quo(A,10^(x))
si x > 0, et A < 0, fait A -> -quo(-A,10^(x))
(donc pour x > 0 c'est comme DSR itéré x fois)
\xintDSHr donne le 'reste' (si x<=0 donne zéro).

Release 1.06 now feeds x to a \numexpr first. I will have to revise this code at some point.

```
\def\xintDSHr {\romannumeral0\xintdshr }%
```

```
\def\xintdshr #1%
{%
    \expandafter\XINT_dshr_checkxpositive \the\numexpr #1\relax\Z
}%
\def\XINT_dshr_checkxpositive #1%
{%
    \xint_UDzerominusfork
      0#1\XINT_dshr_xzeroorneg
      #1-\XINT_dshr_xzeroorneg
       0-\XINT_dshr_xpositive
    \krof #1%
}%
\def\XINT_dshr_xzeroorneg #1\Z #2{ 0}%
\def\XINT_dshr_xpositive #1\Z
{%
    \expandafter\xint_secondoftwo_thenstop\romannumeral0\xintdsx {#1}%
}%
\def\xintDSH {\romannumeral0\xintdsh }%
\def\xintdsh #1#2%
{%
    \expandafter\xint_dsh\expandafter {\romannumeral-`0#2}{#1}%
}%
\def\xint_dsh #1#2%
{%
    \expandafter\XINT_dsh_checksignx \the\numexpr #2\relax\Z {#1}%
}%
\def\XINT_dsh_checksignx #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_dsh_xiszero
      0#1\XINT_dsx_xisNeg_checkA     % on passe direct dans DSx
       0-{\XINT_dsh_xisPos #1}%
    \krof
}%
\def\XINT_dsh_xiszero #1\Z #2{ #2}%
\def\XINT_dsh_xisPos #1\Z #2%
{%
    \expandafter\xint_firstoftwo_thenstop
    \romannumeral0\XINT_dsx_checksignA #2\Z {#1}% via DSx
}%
```

32.56 \xintDSx

Je fais cette routine pour la version 1.01, après modification de \xintDecSplit. Dorénavant \xintDSx fera appel à \xintDecSplit et de même \xintDSH fera appel à \xintDSx. J'ai donc supprimé entièrement l'ancien code de \xintDSH et re-écrit entièrement celui de \xintDecSplit pour x positif.

--> Attention le cas x=0 est traité dans la même catégorie que x > 0 <--
si x < 0, fait A -> A.10^(|x|)

si x >= 0, et A >=0, fait A -> {quo(A,10^(x))}{rem(A,10^(x))}
si x >= 0, et A < 0, d'abord on calcule {quo(-A,10^(x))}{rem(-A,10^(x))}
puis, si le premier n'est pas nul on lui donne le signe -
si le premier est nul on donne le signe - au second.

On peut donc toujours reconstituer l'original A par 10^x Q \pm R où il faut prendre le signe plus si Q est positif ou nul et le signe moins si Q est strictement négatif.

Release 1.06 has a faster and more compactly coded \XINT_dsx_zeroloop. Also, x is now given to a \numexpr. The earlier code should be then simplified, but I leave as is for the time being.

Release 1.07 modified the coding of \XINT_dsx_zeroloop, to avoid impacting the input stack. Indeed the truncating, rounding, and conversion to float routines all use internally \XINT_dsx_zeroloop (via \XINT_dsx_addzerosnofuss), and they were thus roughly limited to generating N = 8 times the input save stack size digits. On TL2012 and TL2013, this means 40000 = 8x5000 digits. Although generating more than 40000 digits is more like a one shot thing, I wanted to open the possibility of outputting tens of thousands of digits to faile, thus I re-organized \XINT_dsx_zeroloop.

January 5, 2014: but it is only with the new division implementation of 1.09j and also with its special \xintXTrunc routine that the possibility mentioned in the last paragraph has become a concrete one in terms of computation time.

```
\def\xintDSx {\romannumeral0\xintdsx }%
\def\xintdsx #1#2%
{%
    \expandafter\xint_dsx\expandafter {\romannumeral-‘0#2}{#1}%
}%
\def\xint_dsx #1#2%
{%
    \expandafter\XINT_dsx_checksignx \the\numexpr #2\relax\Z {#1}%
}%
\def\XINT_DSx #1#2{\romannumeral0\XINT_dsx_checksignx #1\Z {#2}}%
\def\XINT_dsx #1#2{\XINT_dsx_checksignx #1\Z {#2}}%
\def\XINT_dsx_checksignx #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_dsx_xisZero
      0#1\XINT_dsx_xisNeg_checkA
       0-{\XINT_dsx_xisPos #1}%
    \krof
}%
\def\XINT_dsx_xisZero #1\Z #2{ {#2}{0}}% attention comme x > 0
\def\XINT_dsx_xisNeg_checkA #1\Z #2%
{%
    \XINT_dsx_xisNeg_checkA_ #2\Z {#1}%
}%
\def\XINT_dsx_xisNeg_checkA_ #1#2\Z #3%
{%
    \xint_gob_til_zero #1\XINT_dsx_xisNeg_Azero 0%
```

```
    \XINT_dsx_xisNeg_checkx {#3}{#3}{}\Z {#1#2}%
}%
\def\XINT_dsx_xisNeg_Azero #1\Z #2{ 0}%
\def\XINT_dsx_xisNeg_checkx #1%
{%
    \ifnum #1>1000000
       \xint_afterfi
       {\xintError:TooBigDecimalShift
        \expandafter\space\expandafter 0\xint_gobble_iv }%
    \else
       \expandafter \XINT_dsx_zeroloop
    \fi
}%
\def\XINT_dsx_addzerosnofuss #1{\XINT_dsx_zeroloop {#1}{}\Z }%
\def\XINT_dsx_zeroloop #1#2%
{%
    \ifnum #1<\xint_c_ix \XINT_dsx_exita\fi
    \expandafter\XINT_dsx_zeroloop\expandafter
        {\the\numexpr #1-\xint_c_viii}{#200000000}%
}%
\def\XINT_dsx_exita\fi\expandafter\XINT_dsx_zeroloop
{%
    \fi\expandafter\XINT_dsx_exitb
}%
\def\XINT_dsx_exitb #1#2%
{%
    \expandafter\expandafter\expandafter
    \XINT_dsx_addzeros\csname xint_gobble_\romannumeral -#1\endcsname #2%
}%
\def\XINT_dsx_addzeros #1\Z #2{ #2#1}%
\def\XINT_dsx_xisPos #1\Z #2%
{%
    \XINT_dsx_checksignA #2\Z {#1}%
}%
\def\XINT_dsx_checksignA #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_dsx_AisZero
      0#1\XINT_dsx_AisNeg
       0-{\XINT_dsx_AisPos #1}%
    \krof
}%
\def\XINT_dsx_AisZero #1\Z #2{ {0}{0}}%
\def\XINT_dsx_AisNeg #1\Z #2%
{%
    \expandafter\XINT_dsx_AisNeg_dosplit_andcheckfirst
    \romannumeral0\XINT_split_checksizex {#2}{#1}%
}%
\def\XINT_dsx_AisNeg_dosplit_andcheckfirst #1%
```

```
{%
    \XINT_dsx_AisNeg_checkiffirstempty #1\Z
}%
\def\XINT_dsx_AisNeg_checkiffirstempty #1%
{%
    \xint_gob_til_Z #1\XINT_dsx_AisNeg_finish_zero\Z
    \XINT_dsx_AisNeg_finish_notzero #1%
}%
\def\XINT_dsx_AisNeg_finish_zero\Z
    \XINT_dsx_AisNeg_finish_notzero\Z #1%
{%
    \expandafter\XINT_dsx_end
    \expandafter {\romannumeral0\XINT_num {-#1}}{0}%
}%
\def\XINT_dsx_AisNeg_finish_notzero #1\Z #2%
{%
    \expandafter\XINT_dsx_end
    \expandafter {\romannumeral0\XINT_num {#2}}{-#1}%
}%
\def\XINT_dsx_AisPos #1\Z #2%
{%
    \expandafter\XINT_dsx_AisPos_finish
    \romannumeral0\XINT_split_checksizex {#2}{#1}%
}%
\def\XINT_dsx_AisPos_finish #1#2%
{%
    \expandafter\XINT_dsx_end
    \expandafter {\romannumeral0\XINT_num {#2}}%
                 {\romannumeral0\XINT_num {#1}}%
}%
\edef\XINT_dsx_end #1#2%
{%
    \noexpand\expandafter\space\noexpand\expandafter{#2}{#1}%
}%
```

32.57 \xintDecSplit, \xintDecSplitL, \xintDecSplitR

DECIMAL SPLIT

The macro \xintDecSplit {x}{A} first replaces A with |A| (*) This macro cuts the number into two pieces L and R. The concatenation LR always reproduces |A|, and R may be empty or have leading zeros. The position of the cut is specified by the first argument x. If x is zero or positive the cut location is x slots to the left of the right end of the number. If x becomes equal to or larger than the length of the number then L becomes empty. If x is negative the location of the cut is |x| slots to the right of the left end of the number.

(*) warning: this may change in a future version. Only the behavior for A non-negative is guaranteed to remain the same.

v1.05a: \XINT_split_checksizex does not compute the length anymore, rather the error will be from a \numexpr; but the limit of 999999999 does not make much sense.

v1.06: Improvements in \XINT_split_fromleft_loop, \XINT_split_fromright_loop and related macros. More readable coding, speed gains. Also, I now feed immediately a \numexpr with x. Some simplifications should probably be made to the code, which is kept as is for the time being.

1.09e pays attention to the use of xintiabs which acquired in 1.09a the xintnum overhead. So xintiiabs rather without that overhead.

```
\def\xintDecSplitL {\romannumeral0\xintdecsplitl }%
\def\xintDecSplitR {\romannumeral0\xintdecsplitr }%
\def\xintdecsplitl
{%
    \expandafter\xint_firstoftwo_thenstop
    \romannumeral0\xintdecsplit
}%
\def\xintdecsplitr
{%
    \expandafter\xint_secondoftwo_thenstop
    \romannumeral0\xintdecsplit
}%
\def\xintDecSplit {\romannumeral0\xintdecsplit }%
\def\xintdecsplit #1#2%
{%
    \expandafter \xint_split \expandafter
    {\romannumeral0\xintiiabs {#2}}{#1}%  fait expansion de A
}%
\def\xint_split #1#2%
{%
    \expandafter\XINT_split_checksizex\expandafter{\the\numexpr #2}{#1}%
}%
\def\XINT_split_checksizex #1% 999999999 is anyhow very big, could be reduced
{%
    \ifnum\numexpr\XINT_Abs{#1}>999999999
       \xint_afterfi {\xintError:TooBigDecimalSplit\XINT_split_bigx }%
    \else
       \expandafter\XINT_split_xfork
    \fi
    #1\Z
}%
\def\XINT_split_bigx  #1\Z #2%
{%
    \ifcase\XINT_cntSgn #1\Z
    \or \xint_afterfi { }{#2}}% positive big x
    \else
        \xint_afterfi { {#2}{}}% negative big x
    \fi
}%
\def\XINT_split_xfork #1%
{%
    \xint_UDzerominusfork
```

```
      #1-\XINT_split_zerosplit
      0#1\XINT_split_fromleft
       0-{\XINT_split_fromright #1}%
    \krof
}%
\def\XINT_split_zerosplit #1\Z #2{ {#2}{}}%
\def\XINT_split_fromleft  #1\Z #2%
{%
    \XINT_split_fromleft_loop {#1}{}#2\W\W\W\W\W\W\W\W\Z
}%
\def\XINT_split_fromleft_loop #1%
{%
    \ifnum #1<\xint_c_viii\XINT_split_fromleft_exita\fi
    \expandafter\XINT_split_fromleft_loop_perhaps\expandafter
    {\the\numexpr #1-\xint_c_viii\expandafter}\XINT_split_fromleft_eight
}%
\def\XINT_split_fromleft_eight #1#2#3#4#5#6#7#8#9{#9{#1#2#3#4#5#6#7#8#9}}%
\def\XINT_split_fromleft_loop_perhaps #1#2%
{%
    \xint_gob_til_W #2\XINT_split_fromleft_toofar\W
    \XINT_split_fromleft_loop {#1}%
}%
\def\XINT_split_fromleft_toofar\W\XINT_split_fromleft_loop #1#2#3\Z
{%
    \XINT_split_fromleft_toofar_b #2\Z
}%
\def\XINT_split_fromleft_toofar_b #1\W #2\Z { {#1}{}}%
\def\XINT_split_fromleft_exita\fi
    \expandafter\XINT_split_fromleft_loop_perhaps\expandafter #1#2%
   {\fi \XINT_split_fromleft_exitb #1}%
\def\XINT_split_fromleft_exitb\the\numexpr #1-\xint_c_viii\expandafter
{%
    \csname XINT_split_fromleft_endsplit_\romannumeral #1\endcsname
}%
\def\XINT_split_fromleft_endsplit_ #1#2\W #3\Z { {#1}{#2}}%
\def\XINT_split_fromleft_endsplit_i #1#2%
                {\XINT_split_fromleft_checkiftoofar #2{#1#2}}%
\def\XINT_split_fromleft_endsplit_ii #1#2#3%
                {\XINT_split_fromleft_checkiftoofar #3{#1#2#3}}%
\def\XINT_split_fromleft_endsplit_iii #1#2#3#4%
                {\XINT_split_fromleft_checkiftoofar #4{#1#2#3#4}}%
\def\XINT_split_fromleft_endsplit_iv #1#2#3#4#5%
                {\XINT_split_fromleft_checkiftoofar #5{#1#2#3#4#5}}%
\def\XINT_split_fromleft_endsplit_v #1#2#3#4#5#6%
                {\XINT_split_fromleft_checkiftoofar #6{#1#2#3#4#5#6}}%
\def\XINT_split_fromleft_endsplit_vi #1#2#3#4#5#6#7%
                {\XINT_split_fromleft_checkiftoofar #7{#1#2#3#4#5#6#7}}%
\def\XINT_split_fromleft_endsplit_vii #1#2#3#4#5#6#7#8%
                {\XINT_split_fromleft_checkiftoofar #8{#1#2#3#4#5#6#7#8}}%
```

```
\def\XINT_split_fromleft_checkiftoofar #1#2#3\W #4\Z
{%
    \xint_gob_til_W #1\XINT_split_fromleft_wenttoofar\W
    \space {#2}{#3}%
}%
\def\XINT_split_fromleft_wenttoofar\W\space #1%
{%
    \XINT_split_fromleft_wenttoofar_b #1\Z
}%
\def\XINT_split_fromleft_wenttoofar_b #1\W #2\Z { {#1}}%
\def\XINT_split_fromright #1\Z #2%
{%
    \expandafter \XINT_split_fromright_a \expandafter
    {\romannumeral0\xintreverseorder {#2}}{#1}{#2}%
}%
\def\XINT_split_fromright_a #1#2%
{%
    \XINT_split_fromright_loop {#2}{}#1\W\W\W\W\W\W\W\W\Z
}%
\def\XINT_split_fromright_loop #1%
{%
    \ifnum #1<\xint_c_viii\XINT_split_fromright_exita\fi
    \expandafter\XINT_split_fromright_loop_perhaps\expandafter
    {\the\numexpr #1-\xint_c_viii\expandafter }\XINT_split_fromright_eight
}%
\def\XINT_split_fromright_eight #1#2#3#4#5#6#7#8#9{#9{#9#8#7#6#5#4#3#2#1}}%
\def\XINT_split_fromright_loop_perhaps #1#2%
{%
    \xint_gob_til_W #2\XINT_split_fromright_toofar\W
    \XINT_split_fromright_loop {#1}%
}%
\def\XINT_split_fromright_toofar\W\XINT_split_fromright_loop #1#2#3\Z { {}}%
\def\XINT_split_fromright_exita\fi
    \expandafter\XINT_split_fromright_loop_perhaps\expandafter #1#2%
    {\fi \XINT_split_fromright_exitb #1}%
\def\XINT_split_fromright_exitb\the\numexpr #1-\xint_c_viii\expandafter
{%
    \csname XINT_split_fromright_endsplit_\romannumeral #1\endcsname
}%
\edef\XINT_split_fromright_endsplit_ #1#2\W #3\Z #4%
{%
    \noexpand\expandafter\space\noexpand\expandafter
    {\noexpand\romannumeral0\noexpand\xintreverseorder {#2}}{#1}%
}%
\def\XINT_split_fromright_endsplit_i   #1#2%
              {\XINT_split_fromright_checkiftoofar #2{#2#1}}%
\def\XINT_split_fromright_endsplit_ii  #1#2#3%
              {\XINT_split_fromright_checkiftoofar #3{#3#2#1}}%
\def\XINT_split_fromright_endsplit_iii #1#2#3#4%
```

```
             {\XINT_split_fromright_checkiftoofar #4{#4#3#2#1}}%
\def\XINT_split_fromright_endsplit_iv  #1#2#3#4#5%
             {\XINT_split_fromright_checkiftoofar #5{#5#4#3#2#1}}%
\def\XINT_split_fromright_endsplit_v   #1#2#3#4#5#6%
             {\XINT_split_fromright_checkiftoofar #6{#6#5#4#3#2#1}}%
\def\XINT_split_fromright_endsplit_vi  #1#2#3#4#5#6#7%
             {\XINT_split_fromright_checkiftoofar #7{#7#6#5#4#3#2#1}}%
\def\XINT_split_fromright_endsplit_vii #1#2#3#4#5#6#7#8%
             {\XINT_split_fromright_checkiftoofar #8{#8#7#6#5#4#3#2#1}}%
\def\XINT_split_fromright_checkiftoofar #1%
{%
    \xint_gob_til_W #1\XINT_split_fromright_wenttoofar\W
    \XINT_split_fromright_endsplit_
}%
\def\XINT_split_fromright_wenttoofar\W\XINT_split_fromright_endsplit_ #1\Z #2%
    { {}{#2}}%
```

32.58 \xintDouble

v1.08

```
\def\xintDouble {\romannumeral0\xintdouble }%
\def\xintdouble #1%
{%
     \expandafter\XINT_dbl\romannumeral-`0#1%
     \R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W
}%
\def\XINT_dbl #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_dbl_zero
      0#1\XINT_dbl_neg
       0-{\XINT_dbl_pos #1}%
    \krof
}%
\def\XINT_dbl_zero #1\Z \W\W\W\W\W\W\W { 0}%
\def\XINT_dbl_neg
   {\expandafter\xint_minus_thenstop\romannumeral0\XINT_dbl_pos }%
\def\XINT_dbl_pos
{%
    \expandafter\XINT_dbl_a \expandafter{\expandafter}\expandafter 0%
    \romannumeral0\XINT_SQ {}%
}%
\def\XINT_dbl_a #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_W #9\XINT_dbl_end_a\W
    \expandafter\XINT_dbl_b
    \the\numexpr \xint_c_x^viii+#2+\xint_c_ii*#9#8#7#6#5#4#3\relax {#1}%
}%
```

```
\def\XINT_dbl_b 1#1#2#3#4#5#6#7#8#9%
{%
    \XINT_dbl_a {#2#3#4#5#6#7#8#9}{#1}%
}%
\def\XINT_dbl_end_a #1+#2+#3\relax #4%
{%
    \expandafter\XINT_dbl_end_b #2#4%
}%
\edef\XINT_dbl_end_b #1#2#3#4#5#6#7#8%
{%
    \noexpand\expandafter\space\noexpand\the\numexpr #1#2#3#4#5#6#7#8\relax
}%
```

32.59 \xintHalf

v1.08

```
\def\xintHalf {\romannumeral0\xinthalf }%
\def\xinthalf #1%
{%
     \expandafter\XINT_half\romannumeral-`0#1%
     \R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W
}%
\def\XINT_half #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_half_zero
      0#1\XINT_half_neg
       0-{\XINT_half_pos #1}%
    \krof
}%
\def\XINT_half_zero #1\Z \W\W\W\W\W\W\W { 0}%
\def\XINT_half_neg {\expandafter\XINT_opp\romannumeral0\XINT_half_pos }%
\def\XINT_half_pos {\expandafter\XINT_half_a\romannumeral0\XINT_SQ {}}%
\def\XINT_half_a #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_W #8\XINT_half_dont\W
    \expandafter\XINT_half_b
    \the\numexpr \xint_c_x^viii+\xint_c_v*#7#6#5#4#3#2#1\relax #8%
}%
\edef\XINT_half_dont\W\expandafter\XINT_half_b
    \the\numexpr \xint_c_x^viii+\xint_c_v*#1#2#3#4#5#6#7\relax \W\W\W\W\W\W\W
{%
    \noexpand\expandafter\space
    \noexpand\the\numexpr (#1#2#3#4#5#6#7+\xint_c_i)/\xint_c_ii-\xint_c_i \relax
}%
\def\XINT_half_b 1#1#2#3#4#5#6#7#8%
{%
    \XINT_half_c {#2#3#4#5#6#7}{#1}%
```

```
}%
\def\XINT_half_c #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_W #3\XINT_half_end_a #2\W
    \expandafter\XINT_half_d
    \the\numexpr \xint_c_x^viii+\xint_c_v*#9#8#7#6#5#4#3+#2\relax {#1}%
}%
\def\XINT_half_d 1#1#2#3#4#5#6#7#8#9%
{%
    \XINT_half_c {#2#3#4#5#6#7#8#9}{#1}%
}%
\def\XINT_half_end_a #1\W #2\relax #3%
{%
    \xint_gob_til_zero #1\XINT_half_end_b 0\space #1#3%
}%
\edef\XINT_half_end_b 0\space 0#1#2#3#4#5#6#7%
{%
    \noexpand\expandafter\space\noexpand\the\numexpr #1#2#3#4#5#6#7\relax
}%
```

32.60 \xintDec

v1.08

```
\def\xintDec {\romannumeral0\xintdec }%
\def\xintdec #1%
{%
     \expandafter\XINT_dec\romannumeral-`0#1%
     \R\R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W\W
}%
\def\XINT_dec #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_dec_zero
      0#1\XINT_dec_neg
       0-{\XINT_dec_pos #1}%
    \krof
}%
\def\XINT_dec_zero #1\W\W\W\W\W\W\W\W { -1}%
\def\XINT_dec_neg
   {\expandafter\xint_minus_thenstop\romannumeral0\XINT_inc_pos }%
\def\XINT_dec_pos
{%
    \expandafter\XINT_dec_a \expandafter{\expandafter}%
    \romannumeral0\XINT_OQ {}%
}%
\def\XINT_dec_a #1#2#3#4#5#6#7#8#9%
{%
    \expandafter\XINT_dec_b
```

```
    \the\numexpr 11#9#8#7#6#5#4#3#2-\xint_c_i\relax {#1}%
}%
\def\XINT_dec_b 1#1%
{%
    \xint_gob_til_one #1\XINT_dec_A 1\XINT_dec_c
}%
\def\XINT_dec_c #1#2#3#4#5#6#7#8#9{\XINT_dec_a {#1#2#3#4#5#6#7#8#9}}%
\def\XINT_dec_A 1\XINT_dec_c #1#2#3#4#5#6#7#8#9%
   {\XINT_dec_B {#1#2#3#4#5#6#7#8#9}}%
\def\XINT_dec_B #1#2\W\W\W\W\W\W\W\W
{%
    \expandafter\XINT_dec_cleanup
    \romannumeral0\XINT_rord_main {}#2%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax
    #1%
}%
\edef\XINT_dec_cleanup #1#2#3#4#5#6#7#8%
    {\noexpand\expandafter\space\noexpand\the\numexpr #1#2#3#4#5#6#7#8\relax }%
```

32.61 \xintInc

v1.08

```
\def\xintInc {\romannumeral0\xintinc }%
\def\xintinc #1%
{%
     \expandafter\XINT_inc\romannumeral-`0#1%
     \R\R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W\W
}%
\def\XINT_inc #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_inc_zero
      0#1\XINT_inc_neg
       0-{\XINT_inc_pos #1}%
    \krof
}%
\def\XINT_inc_zero #1\W\W\W\W\W\W\W\W { 1}%
\def\XINT_inc_neg {\expandafter\XINT_opp\romannumeral0\XINT_dec_pos }%
\def\XINT_inc_pos
{%
    \expandafter\XINT_inc_a \expandafter{\expandafter}%
    \romannumeral0\XINT_OQ {}%
}%
\def\XINT_inc_a #1#2#3#4#5#6#7#8#9%
{%
```

```
    \xint_gob_til_W #9\XINT_inc_end\W
    \expandafter\XINT_inc_b
    \the\numexpr 10#9#8#7#6#5#4#3#2+\xint_c_i\relax {#1}%
}%
\def\XINT_inc_b 1#1%
{%
    \xint_gob_til_zero #1\XINT_inc_A 0\XINT_inc_c
}%
\def\XINT_inc_c #1#2#3#4#5#6#7#8#9{\XINT_inc_a {#1#2#3#4#5#6#7#8#9}}%
\def\XINT_inc_A 0\XINT_inc_c #1#2#3#4#5#6#7#8#9%
                 {\XINT_dec_B {#1#2#3#4#5#6#7#8#9}}%
\def\XINT_inc_end\W #1\relax #2{ 1#2}%
```

32.62 \xintiSqrt, \xintiSquareRoot

v1.08. 1.09a uses \xintnum.

Some overhead was added inadvertently in 1.09a to inner routines when \xintiquo and \xintidivision were also promoted to use \xintnum; release 1.09f thus uses \xintiiquo and \xintiidivision xhich avoid this \xintnum overhead.

1.09j replaced the previous long \ifcase from \XINT_sqrt_c by some nested \ifnum's.

```
\def\xintiSqrt {\romannumeral0\xintisqrt }%
\def\xintisqrt
    {\expandafter\XINT_sqrt_post\romannumeral0\xintisquareroot }%
\def\XINT_sqrt_post #1#2{\XINT_dec_pos #1\R\R\R\R\R\R\R\R\Z
                                        \W\W\W\W\W\W\W\W }%
\def\xintiSquareRoot {\romannumeral0\xintisquareroot }%
\def\xintisquareroot #1%
    {\expandafter\XINT_sqrt_checkin\romannumeral0\xintnum{#1}\Z}%
\def\XINT_sqrt_checkin #1%
{%
    \xint_UDzerominusfork
     #1-\XINT_sqrt_iszero
     0#1\XINT_sqrt_isneg
      0-{\XINT_sqrt #1}%
    \krof
}%
\def\XINT_sqrt_iszero #1\Z { 1.}%
\edef\XINT_sqrt_isneg #1\Z {\noexpand\xintError:RootOfNegative\space 1.}%
\def\XINT_sqrt #1\Z
{%
    \expandafter\XINT_sqrt_start\expandafter
            {\romannumeral0\xintlength {#1}}{#1}%
}%
\def\XINT_sqrt_start #1%
{%
    \ifnum #1<\xint_c_x
       \expandafter\XINT_sqrt_small_a
```

```
    \else
       \expandafter\XINT_sqrt_big_a
    \fi
    {#1}%
}%
\def\XINT_sqrt_small_a #1{\XINT_sqrt_a {#1}\XINT_sqrt_small_d }%
\def\XINT_sqrt_big_a   #1{\XINT_sqrt_a {#1}\XINT_sqrt_big_d   }%
\def\XINT_sqrt_a #1%
{%
   \ifodd #1
     \expandafter\XINT_sqrt_bB
   \else
     \expandafter\XINT_sqrt_bA
   \fi
   {#1}%
}%
\def\XINT_sqrt_bA #1#2#3%
{%
    \XINT_sqrt_bA_b #3\Z #2{#1}{#3}%
}%
\def\XINT_sqrt_bA_b #1#2#3\Z
{%
    \XINT_sqrt_c {#1#2}%
}%
\def\XINT_sqrt_bB #1#2#3%
{%
    \XINT_sqrt_bB_b #3\Z #2{#1}{#3}%
}%
\def\XINT_sqrt_bB_b #1#2\Z
{%
    \XINT_sqrt_c #1%
}%
\def\XINT_sqrt_c #1#2%
{%
    \expandafter #2\expandafter
    {\the\numexpr\ifnum #1>\xint_c_iii
                 \ifnum #1>\xint_c_viii
                 \ifnum #1>15 \ifnum #1>24 \ifnum #1>35
                 \ifnum #1>48 \ifnum #1>63 \ifnum #1>80
      10\else 9\fi \else 8\fi \else 7\fi \else 6\fi
        \else 5\fi \else 4\fi \else 3\fi \else 2\fi \relax }%
}%
\def\XINT_sqrt_small_d #1#2%
{%
   \expandafter\XINT_sqrt_small_e\expandafter
   {\the\numexpr #1\ifcase \numexpr #2/\xint_c_ii-\xint_c_i\relax
                     \or 0\or 00\or 000\or 0000\fi }%
}%
\def\XINT_sqrt_small_e #1#2%
```

```
{%
  \expandafter\XINT_sqrt_small_f\expandafter {\the\numexpr #1*#1-#2}{#1}%
}%
\def\XINT_sqrt_small_f #1#2%
{%
  \expandafter\XINT_sqrt_small_g\expandafter
  {\the\numexpr ((#1+#2)/(\xint_c_ii*#2))-\xint_c_i}{#1}{#2}%
}%
\def\XINT_sqrt_small_g #1%
{%
   \ifnum #1>\xint_c_
      \expandafter\XINT_sqrt_small_h
   \else
      \expandafter\XINT_sqrt_small_end
   \fi
   {#1}%
}%
\def\XINT_sqrt_small_h #1#2#3%
{%
   \expandafter\XINT_sqrt_small_f\expandafter
   {\the\numexpr #2-\xint_c_ii*#1*#3+#1*#1\expandafter}\expandafter
   {\the\numexpr #3-#1}%
}%
\def\XINT_sqrt_small_end #1#2#3{ {#3}{#2}}%
\def\XINT_sqrt_big_d #1#2%
{%
  \ifodd #2
    \expandafter\expandafter\expandafter\XINT_sqrt_big_eB
  \else
    \expandafter\expandafter\expandafter\XINT_sqrt_big_eA
  \fi
  \expandafter {\the\numexpr #2/\xint_c_ii }{#1}%
}%
\def\XINT_sqrt_big_eA  #1#2#3%
{%
   \XINT_sqrt_big_eA_a #3\Z {#2}{#1}{#3}%
}%
\def\XINT_sqrt_big_eA_a #1#2#3#4#5#6#7#8#9\Z
{%
   \XINT_sqrt_big_eA_b {#1#2#3#4#5#6#7#8}%
}%
\def\XINT_sqrt_big_eA_b #1#2%
{%
   \expandafter\XINT_sqrt_big_f
   \romannumeral0\XINT_sqrt_small_e {#2000}{#1}{#1}%
}%
\def\XINT_sqrt_big_eB #1#2#3%
{%
   \XINT_sqrt_big_eB_a #3\Z {#2}{#1}{#3}%
}%
```

```
}%
\def\XINT_sqrt_big_eB_a #1#2#3#4#5#6#7#8#9%
{%
    \XINT_sqrt_big_eB_b {#1#2#3#4#5#6#7#8#9}%
}%
\def\XINT_sqrt_big_eB_b #1#2\Z #3%
{%
    \expandafter\XINT_sqrt_big_f
    \romannumeral0\XINT_sqrt_small_e {#30000}{#1}{#1}%
}%
\def\XINT_sqrt_big_f #1#2#3#4%
{%
   \expandafter\XINT_sqrt_big_f_a\expandafter
   {\the\numexpr #2+#3\expandafter}\expandafter
   {\romannumeral0\XINT_dsx_addzerosnofuss
                    {\numexpr #4-\xint_c_iv\relax}{#1}}{#4}%
}%
\def\XINT_sqrt_big_f_a #1#2#3#4%
{%
   \expandafter\XINT_sqrt_big_g\expandafter
   {\romannumeral0\xintiisub
       {\XINT_dsx_addzerosnofuss
        {\numexpr \xint_c_ii*#3-\xint_c_viii\relax}{#1}}{#4}}%
   {#2}{#3}%
}%
\def\XINT_sqrt_big_g #1#2%
{%
    \expandafter\XINT_sqrt_big_j
    \romannumeral0\xintiidivision{#1}%
        {\romannumeral0\XINT_dbl_pos #2\R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W }{#2}%
}%
\def\XINT_sqrt_big_j #1%
{%
    \if0\XINT_Sgn #1\Z
          \expandafter \XINT_sqrt_big_end
    \else \expandafter \XINT_sqrt_big_k
    \fi {#1}%
}%
\def\XINT_sqrt_big_k #1#2#3%
{%
    \expandafter\XINT_sqrt_big_l\expandafter
    {\romannumeral0\xintiisub {#3}{#1}}%
    {\romannumeral0\xintiiadd {#2}{\xintiiSqr {#1}}}%
}%
\def\XINT_sqrt_big_l #1#2%
{%
   \expandafter\XINT_sqrt_big_g\expandafter
   {#2}{#1}%
}%
```

```
\def\XINT_sqrt_big_end #1#2#3#4{ {#3}{#2}}%
```

32.63 \xintIsTrue:csv

1.09c. For use by \xinttheboolexpr.(inside \csname, no need for a \romannumeral here). The macros may well be defined already here. I make no advertisement because I have inserted no space parsing in the :csv macros, as they will be used only with privately created comma separated lists, having no space naturally. Nevertheless they exist and can be used.

```
\def\xintIsTrue:csv #1{\expandafter\XINT_istrue:_a\romannumeral-`0#1,,^}%
\def\XINT_istrue:_a {\XINT_istrue:_b {}}%
\def\XINT_istrue:_b #1#2,%
             {\expandafter\XINT_istrue:_c\romannumeral-`0#2,{#1}}%
\def\XINT_istrue:_c #1{\if #1,\expandafter\XINT_:_f
                      \else\expandafter\XINT_istrue:_d\fi #1}%
\def\XINT_istrue:_d #1,%
         {\expandafter\XINT_istrue:_e\romannumeral0\xintisnotzero {#1},}%
\def\XINT_istrue:_e #1,#2{\XINT_istrue:_b {#2,#1}}%
\def\XINT_:_f ,#1#2^{\xint_gobble_i #1}%
```

32.64 \xintANDof:csv

1.09a. For use by \xintexpr (inside \csname, no need for a \romannumeral here).

```
\def\xintANDof:csv #1{\expandafter\XINT_andof:_a\romannumeral-`0#1,,^}%
\def\XINT_andof:_a {\expandafter\XINT_andof:_b\romannumeral-`0}%
\def\XINT_andof:_b #1{\if #1,\expandafter\XINT_andof:_e
                      \else\expandafter\XINT_andof:_c\fi #1}%
\def\XINT_andof:_c #1,{\xintifTrueAelseB {#1}{\XINT_andof:_a}{\XINT_andof:_no}}%
\def\XINT_andof:_no #1^{0}%
\def\XINT_andof:_e  #1^{1}% works with empty list
```

32.65 \xintORof:csv

1.09a. For use by \xintexpr.

```
\def\xintORof:csv #1{\expandafter\XINT_orof:_a\romannumeral-`0#1,,^}%
\def\XINT_orof:_a {\expandafter\XINT_orof:_b\romannumeral-`0}%
\def\XINT_orof:_b #1{\if #1,\expandafter\XINT_orof:_e
                      \else\expandafter\XINT_orof:_c\fi #1}%
\def\XINT_orof:_c #1,{\xintifTrueAelseB{#1}{\XINT_orof:_yes}{\XINT_orof:_a}}%
\def\XINT_orof:_yes #1^{1}%
\def\XINT_orof:_e   #1^{0}% works with empty list
```

32.66 \xintXORof:csv

1.09a. For use by \xintexpr (inside a \csname..\endcsname).

```
\def\xintXORof:csv #1{\expandafter\XINT_xorof:_a\expandafter
                         0\romannumeral-`0#1,,^}%
\def\XINT_xorof:_a #1#2,{\expandafter\XINT_xorof:_b\romannumeral-`0#2,#1}%
\def\XINT_xorof:_b #1{\if #1,\expandafter\XINT_:_e
                       \else\expandafter\XINT_xorof:_c\fi #1}%
\def\XINT_xorof:_c #1,#2%
           {\xintifTrueAelseB {#1}{\if #20\xint_afterfi{\XINT_xorof:_a 1}%
                                   \else\xint_afterfi{\XINT_xorof:_a 0}\fi}%
                                  {\XINT_xorof:_a #2}%
           }%
\def\XINT_:_e ,#1#2^{#1}% allows empty list
```

32.67 \xintiMaxof:csv

1.09i. For use by \xintiiexpr.

```
\def\xintiMaxof:csv #1{\expandafter\XINT_imaxof:_b\romannumeral-`0#1,,}%
\def\XINT_imaxof:_b #1,#2,{\expandafter\XINT_imaxof:_c\romannumeral-`0#2,{#1},}%
\def\XINT_imaxof:_c #1{\if #1,\expandafter\XINT_of:_e
                        \else\expandafter\XINT_imaxof:_d\fi #1}%
\def\XINT_imaxof:_d #1,{\expandafter\XINT_imaxof:_b\romannumeral0\xintimax {#1}}%
\def\XINT_of:_e ,#1,{#1}%
\let\xintMaxof:csv\xintiMaxof:csv
```

32.68 \xintiMinof:csv

1.09i. For use by \xintiiexpr.

```
\def\xintiMinof:csv #1{\expandafter\XINT_iminof:_b\romannumeral-`0#1,,}%
\def\XINT_iminof:_b #1,#2,{\expandafter\XINT_iminof:_c\romannumeral-`0#2,{#1},}%
\def\XINT_iminof:_c #1{\if #1,\expandafter\XINT_of:_e
                        \else\expandafter\XINT_iminof:_d\fi #1}%
\def\XINT_iminof:_d #1,{\expandafter\XINT_iminof:_b\romannumeral0\xintimin {#1}}%
\let\xintMinof:csv\xintiMinof:csv
```

32.69 \xintiiSum:csv

1.09i. For use by \xintiiexpr.

```
\def\xintiiSum:csv #1{\expandafter\XINT_iisum:_a\romannumeral-`0#1,,^}%
\def\XINT_iisum:_a {\XINT_iisum:_b {0}}%
\def\XINT_iisum:_b #1#2,{\expandafter\XINT_iisum:_c\romannumeral-`0#2,{#1}}%
\def\XINT_iisum:_c #1{\if #1,\expandafter\XINT_:_e
                       \else\expandafter\XINT_iisum:_d\fi #1}%
```

```
\def\XINT_iisum:_d #1,#2{\expandafter\XINT_iisum:_b\expandafter
                         {\romannumeral0\xintiiadd {#2}{#1}}}%
\let\xintSum:csv\xintiiSum:csv
```

32.70 \xintiiPrd:csv

1.09i. For use by \xintiiexpr.

```
\def\xintiiPrd:csv #1{\expandafter\XINT_iiprd:_a\romannumeral-`0#1,,^}%
\def\XINT_iiprd:_a {\XINT_iiprd:_b {1}}%
\def\XINT_iiprd:_b #1#2,{\expandafter\XINT_iiprd:_c\romannumeral-`0#2,{#1}}%
\def\XINT_iiprd:_c #1{\if #1,\expandafter\XINT_:_e
                       \else\expandafter\XINT_iiprd:_d\fi #1}%
\def\XINT_iiprd:_d #1,#2{\expandafter\XINT_iiprd:_b\expandafter
                         {\romannumeral0\xintiimul {#2}{#1}}}%
\let\xintPrd:csv\xintiiPrd:csv
\XINT_restorecatcodes_endinput%
```

33 Package xintbinhex implementation

The commenting is currently (2014/02/13) very sparse.

Contents

33.1 Catcodes, ε-TEX and reload detection

The code for reload detection is copied from Heiko Oberdiek's packages, and adapted here to check for previous loading of the master xint package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
```

```
  \catcode46=12   % .
  \catcode58=12   % :
  \def\space { }%
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintbinhex.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintbinhex}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintbinhex.sty
      \ifx\w\relax % but xint.sty not yet loaded.
         \y{xintbinhex}{now issuing \string\input\space xint.sty}%
         \def\z{\endgroup\input xint.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xint.sty not yet loaded.
            \y{xintbinhex}{now issuing \string\RequirePackage{xint}}%
            \def\z{\endgroup\RequirePackage{xint}}%
          \fi
      \else
        \y{xintbinhex}{I was already loaded, aborting input}%
        \aftergroup\endinput
      \fi
    \fi
  \fi
\z%
```

33.2 Confirmation of xint loading

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
```

```
  \catcode46=12   % .
  \catcode58=12   % :
  \ifdefined\PackageInfo
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \else
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
  \fi
  \def\empty {}%
  \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
  \ifx\w\relax % Plain TeX, user gave a file name at the prompt
      \y{xintbinhex}{Loading of package xint failed, aborting input}%
      \aftergroup\endinput
  \fi
  \ifx\w\empty % LaTeX, user gave a file name at the prompt
      \y{xintbinhex}{Loading of package xint failed, aborting input}%
      \aftergroup\endinput
  \fi
\endgroup%
```

33.3 Catcodes

```
\XINTsetupcatcodes%
```

33.4 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintbinhex}%
  [2014/02/13 v1.09kb Expandable binary and hexadecimal conversions (jfB)]%
```

33.5 Constants, etc...

v1.08

```
\chardef\xint_c_xvi           16
% \chardef\xint_c_ii^v          32 % already done in xint.sty
% \chardef\xint_c_ii^vi         64 % already done in xint.sty
\chardef\xint_c_ii^vii       128
\mathchardef\xint_c_ii^viii  256
\mathchardef\xint_c_ii^xii  4096
\newcount\xint_c_ii^xv  \xint_c_ii^xv  32768
\newcount\xint_c_ii^xvi \xint_c_ii^xvi 65536
\newcount\xint_c_x^v    \xint_c_x^v    100000
\newcount\xint_c_x^ix   \xint_c_x^ix   1000000000
\def\XINT_tmpa #1{%
  \expandafter\edef\csname XINT_sdth_#1\endcsname
  {\ifcase #1 0\or 1\or 2\or 3\or 4\or 5\or 6\or 7\or
              8\or 9\or A\or B\or C\or D\or E\or F\fi}}%
\xintApplyInline\XINT_tmpa
    {{0}{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}{12}{13}{14}{15}}%
\def\XINT_tmpa #1{%
  \expandafter\edef\csname XINT_sdtb_#1\endcsname
```

```
  {\ifcase #1
   0000\or 0001\or 0010\or 0011\or 0100\or 0101\or 0110\or 0111\or
   1000\or 1001\or 1010\or 1011\or 1100\or 1101\or 1110\or 1111\fi}}%
\xintApplyInline\XINT_tmpa
    {{0}{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}{12}{13}{14}{15}}%
\let\XINT_tmpa\relax
\expandafter\def\csname XINT_sbtd_0000\endcsname {0}%
\expandafter\def\csname XINT_sbtd_0001\endcsname {1}%
\expandafter\def\csname XINT_sbtd_0010\endcsname {2}%
\expandafter\def\csname XINT_sbtd_0011\endcsname {3}%
\expandafter\def\csname XINT_sbtd_0100\endcsname {4}%
\expandafter\def\csname XINT_sbtd_0101\endcsname {5}%
\expandafter\def\csname XINT_sbtd_0110\endcsname {6}%
\expandafter\def\csname XINT_sbtd_0111\endcsname {7}%
\expandafter\def\csname XINT_sbtd_1000\endcsname {8}%
\expandafter\def\csname XINT_sbtd_1001\endcsname {9}%
\expandafter\def\csname XINT_sbtd_1010\endcsname {10}%
\expandafter\def\csname XINT_sbtd_1011\endcsname {11}%
\expandafter\def\csname XINT_sbtd_1100\endcsname {12}%
\expandafter\def\csname XINT_sbtd_1101\endcsname {13}%
\expandafter\def\csname XINT_sbtd_1110\endcsname {14}%
\expandafter\def\csname XINT_sbtd_1111\endcsname {15}%
\expandafter\let\csname XINT_sbth_0000\expandafter\endcsname
                \csname XINT_sbtd_0000\endcsname
\expandafter\let\csname XINT_sbth_0001\expandafter\endcsname
                \csname XINT_sbtd_0001\endcsname
\expandafter\let\csname XINT_sbth_0010\expandafter\endcsname
                \csname XINT_sbtd_0010\endcsname
\expandafter\let\csname XINT_sbth_0011\expandafter\endcsname
                \csname XINT_sbtd_0011\endcsname
\expandafter\let\csname XINT_sbth_0100\expandafter\endcsname
                \csname XINT_sbtd_0100\endcsname
\expandafter\let\csname XINT_sbth_0101\expandafter\endcsname
                \csname XINT_sbtd_0101\endcsname
\expandafter\let\csname XINT_sbth_0110\expandafter\endcsname
                \csname XINT_sbtd_0110\endcsname
\expandafter\let\csname XINT_sbth_0111\expandafter\endcsname
                \csname XINT_sbtd_0111\endcsname
\expandafter\let\csname XINT_sbth_1000\expandafter\endcsname
                \csname XINT_sbtd_1000\endcsname
\expandafter\let\csname XINT_sbth_1001\expandafter\endcsname
                \csname XINT_sbtd_1001\endcsname
\expandafter\def\csname XINT_sbth_1010\endcsname {A}%
\expandafter\def\csname XINT_sbth_1011\endcsname {B}%
\expandafter\def\csname XINT_sbth_1100\endcsname {C}%
\expandafter\def\csname XINT_sbth_1101\endcsname {D}%
\expandafter\def\csname XINT_sbth_1110\endcsname {E}%
\expandafter\def\csname XINT_sbth_1111\endcsname {F}%
\expandafter\def\csname XINT_shtb_0\endcsname {0000}%
```

```
\expandafter\def\csname XINT_shtb_1\endcsname {0001}%
\expandafter\def\csname XINT_shtb_2\endcsname {0010}%
\expandafter\def\csname XINT_shtb_3\endcsname {0011}%
\expandafter\def\csname XINT_shtb_4\endcsname {0100}%
\expandafter\def\csname XINT_shtb_5\endcsname {0101}%
\expandafter\def\csname XINT_shtb_6\endcsname {0110}%
\expandafter\def\csname XINT_shtb_7\endcsname {0111}%
\expandafter\def\csname XINT_shtb_8\endcsname {1000}%
\expandafter\def\csname XINT_shtb_9\endcsname {1001}%
\def\XINT_shtb_A {1010}%
\def\XINT_shtb_B {1011}%
\def\XINT_shtb_C {1100}%
\def\XINT_shtb_D {1101}%
\def\XINT_shtb_E {1110}%
\def\XINT_shtb_F {1111}%
\def\XINT_shtb_G {}%
\def\XINT_smallhex #1%
{%
    \expandafter\XINT_smallhex_a\expandafter
    {\the\numexpr (#1+\xint_c_viii)/\xint_c_xvi-\xint_c_i}{#1}%
}%
\def\XINT_smallhex_a #1#2%
{%
    \csname XINT_sdth_#1\expandafter\expandafter\expandafter\endcsname
    \csname XINT_sdth_\the\numexpr #2-\xint_c_xvi*#1\endcsname
}%
\def\XINT_smallbin #1%
{%
    \expandafter\XINT_smallbin_a\expandafter
    {\the\numexpr (#1+\xint_c_viii)/\xint_c_xvi-\xint_c_i}{#1}%
}%
\def\XINT_smallbin_a #1#2%
{%
    \csname XINT_sdtb_#1\expandafter\expandafter\expandafter\endcsname
    \csname XINT_sdtb_\the\numexpr #2-\xint_c_xvi*#1\endcsname
}%
```

33.6 \xintDecToHex, \xintDecToBin

v1.08

```
\def\xintDecToHex {\romannumeral0\xintdectohex }%
\def\xintdectohex #1%
        {\expandafter\XINT_dth_checkin\romannumeral-`0#1\W\W\W\W \T}%
\def\XINT_dth_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_dth_N
        -{\XINT_dth_P #1}%
```

```
      \krof
}%
\def\XINT_dth_N {\expandafter\xint_minus_thenstop\romannumeral0\XINT_dth_P }%
\def\XINT_dth_P {\expandafter\XINT_dth_III\romannumeral-`0\XINT_dtbh_I {0.}}%
\def\xintDecToBin {\romannumeral0\xintdectobin }%
\def\xintdectobin #1%
        {\expandafter\XINT_dtb_checkin\romannumeral-`0#1\W\W\W\W \T }%
\def\XINT_dtb_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_dtb_N
        -{\XINT_dtb_P #1}%
     \krof
}%
\def\XINT_dtb_N {\expandafter\xint_minus_thenstop\romannumeral0\XINT_dtb_P }%
\def\XINT_dtb_P {\expandafter\XINT_dtb_III\romannumeral-`0\XINT_dtbh_I {0.}}%
\def\XINT_dtbh_I #1#2#3#4#5%
{%
    \xint_gob_til_W #5\XINT_dtbh_II_a\W\XINT_dtbh_I_a  {}{#2#3#4#5}#1\Z.%
}%
\def\XINT_dtbh_II_a\W\XINT_dtbh_I_a #1#2{\XINT_dtbh_II_b #2}%
\def\XINT_dtbh_II_b #1#2#3#4%
{%
    \xint_gob_til_W
      #1\XINT_dtbh_II_c
      #2\XINT_dtbh_II_ci
      #3\XINT_dtbh_II_cii
      \W\XINT_dtbh_II_ciii #1#2#3#4%
}%
\def\XINT_dtbh_II_c \W\XINT_dtbh_II_ci
                    \W\XINT_dtbh_II_cii
                    \W\XINT_dtbh_II_ciii \W\W\W\W {{}}%
\def\XINT_dtbh_II_ci #1\XINT_dtbh_II_ciii #2\W\W\W
   {\XINT_dtbh_II_d {}{#2}{0}}%
\def\XINT_dtbh_II_cii\W\XINT_dtbh_II_ciii #1#2\W\W
   {\XINT_dtbh_II_d {}{#1#2}{00}}%
\def\XINT_dtbh_II_ciii #1#2#3\W
   {\XINT_dtbh_II_d {}{#1#2#3}{000}}%
\def\XINT_dtbh_I_a #1#2#3.%
{%
    \xint_gob_til_Z #3\XINT_dtbh_I_z\Z
    \expandafter\XINT_dtbh_I_b\the\numexpr #2+#30000.{#1}%
}%
\def\XINT_dtbh_I_b #1.%
{%
    \expandafter\XINT_dtbh_I_c\the\numexpr
    (#1+\xint_c_ii^xv)/\xint_c_ii^xvi-\xint_c_i.#1.%
}%
\def\XINT_dtbh_I_c #1.#2.%
```

```
{%
    \expandafter\XINT_dtbh_I_d\expandafter
    {\the\numexpr #2-\xint_c_ii^xvi*#1}{#1}%
}%
\def\XINT_dtbh_I_d #1#2#3{\XINT_dtbh_I_a {#3#1.}{#2}}%
\def\XINT_dtbh_I_z\Z\expandafter\XINT_dtbh_I_b\the\numexpr #1+#2.%
{%
    \ifnum #1=\xint_c_ \expandafter\XINT_dtbh_I_end_zb\fi
    \XINT_dtbh_I_end_za {#1}%
}%
\def\XINT_dtbh_I_end_za #1#2{\XINT_dtbh_I {#2#1.}}%
\def\XINT_dtbh_I_end_zb\XINT_dtbh_I_end_za #1#2{\XINT_dtbh_I {#2}}%
\def\XINT_dtbh_II_d #1#2#3#4.%
{%
    \xint_gob_til_Z #4\XINT_dtbh_II_z\Z
    \expandafter\XINT_dtbh_II_e\the\numexpr #2+#4#3.{#1}{#3}%
}%
\def\XINT_dtbh_II_e #1.%
{%
    \expandafter\XINT_dtbh_II_f\the\numexpr
        (#1+\xint_c_ii^xv)/\xint_c_ii^xvi-\xint_c_i.#1.%
}%
\def\XINT_dtbh_II_f #1.#2.%
{%
    \expandafter\XINT_dtbh_II_g\expandafter
    {\the\numexpr #2-\xint_c_ii^xvi*#1}{#1}%
}%
\def\XINT_dtbh_II_g #1#2#3{\XINT_dtbh_II_d {#3#1.}{#2}}%
\def\XINT_dtbh_II_z\Z\expandafter\XINT_dtbh_II_e\the\numexpr #1+#2.%
{%
    \ifnum #1=\xint_c_ \expandafter\XINT_dtbh_II_end_zb\fi
    \XINT_dtbh_II_end_za {#1}%
}%
\def\XINT_dtbh_II_end_za #1#2#3{{}#2#1.\Z.}%
\def\XINT_dtbh_II_end_zb\XINT_dtbh_II_end_za #1#2#3{{}#2\Z.}%
\def\XINT_dth_III #1#2.%
{%
    \xint_gob_til_Z #2\XINT_dth_end\Z
    \expandafter\XINT_dth_III\expandafter
    {\romannumeral-`0\XINT_dth_small #2.#1}%
}%
\def\XINT_dth_small #1.%
{%
    \expandafter\XINT_smallhex\expandafter
    {\the\numexpr (#1+\xint_c_ii^vii)/\xint_c_ii^viii-\xint_c_i\expandafter}%
    \romannumeral-`0\expandafter\XINT_smallhex\expandafter
    {\the\numexpr
    #1-((#1+\xint_c_ii^vii)/\xint_c_ii^viii-\xint_c_i)*\xint_c_ii^viii}%
}%
```

```
\def\XINT_dth_end\Z\expandafter\XINT_dth_III\expandafter #1#2\T
{%
    \XINT_dth_end_b #1%
}%
\def\XINT_dth_end_b #1.{\XINT_dth_end_c }%
\def\XINT_dth_end_c #1{\xint_gob_til_zero #1\XINT_dth_end_d 0\space #1}%
\def\XINT_dth_end_d 0\space 0#1%
{%
    \xint_gob_til_zero #1\XINT_dth_end_e 0\space #1%
}%
\def\XINT_dth_end_e 0\space 0#1%
{%
    \xint_gob_til_zero #1\XINT_dth_end_f 0\space #1%
}%
\def\XINT_dth_end_f 0\space 0{ }%
\def\XINT_dtb_III #1#2.%
{%
    \xint_gob_til_Z #2\XINT_dtb_end\Z
    \expandafter\XINT_dtb_III\expandafter
    {\romannumeral-`0\XINT_dtb_small #2.#1}%
}%
\def\XINT_dtb_small #1.%
{%
    \expandafter\XINT_smallbin\expandafter
    {\the\numexpr (#1+\xint_c_ii^vii)/\xint_c_ii^viii-\xint_c_i\expandafter}%
    \romannumeral-`0\expandafter\XINT_smallbin\expandafter
    {\the\numexpr
    #1-((#1+\xint_c_ii^vii)/\xint_c_ii^viii-\xint_c_i)*\xint_c_ii^viii}%
}%
\def\XINT_dtb_end\Z\expandafter\XINT_dtb_III\expandafter #1#2\T
{%
    \XINT_dtb_end_b #1%
}%
\def\XINT_dtb_end_b #1.{\XINT_dtb_end_c }%
\def\XINT_dtb_end_c #1#2#3#4#5#6#7#8%
{%
    \expandafter\XINT_dtb_end_d\the\numexpr #1#2#3#4#5#6#7#8\relax
}%
\edef\XINT_dtb_end_d #1#2#3#4#5#6#7#8#9%
{%
    \noexpand\expandafter\space\noexpand\the\numexpr #1#2#3#4#5#6#7#8#9\relax
}%
```

33.7 \xintHexToDec

v1.08

```
\def\xintHexToDec {\romannumeral0\xinthextodec }%
\def\xinthextodec #1%
```

```
        {\expandafter\XINT_htd_checkin\romannumeral-`0#1\W\W\W\W \T }%
\def\XINT_htd_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_htd_neg
        -{\XINT_htd_I {0000}#1}%
     \krof
}%
\def\XINT_htd_neg {\expandafter\xint_minus_thenstop
                   \romannumeral0\XINT_htd_I {0000}}%
\def\XINT_htd_I #1#2#3#4#5%
{%
    \xint_gob_til_W #5\XINT_htd_II_a\W
    \XINT_htd_I_a  {}{"#2#3#4#5}#1\Z\Z\Z\Z
}%
\def\XINT_htd_II_a \W\XINT_htd_I_a #1#2{\XINT_htd_II_b #2}%
\def\XINT_htd_II_b "#1#2#3#4%
{%
    \xint_gob_til_W
      #1\XINT_htd_II_c
      #2\XINT_htd_II_ci
      #3\XINT_htd_II_cii
      \W\XINT_htd_II_ciii #1#2#3#4%
}%
\def\XINT_htd_II_c \W\XINT_htd_II_ci
                   \W\XINT_htd_II_cii
                   \W\XINT_htd_II_ciii \W\W\W\W #1\Z\Z\Z\Z\T
{%
    \expandafter\xint_cleanupzeros_andstop
    \romannumeral0\XINT_rord_main {}#1%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax
}%
\def\XINT_htd_II_ci #1\XINT_htd_II_ciii
                      #2\W\W\W {\XINT_htd_II_d {}{"#2}{\xint_c_xvi}}%
\def\XINT_htd_II_cii\W\XINT_htd_II_ciii
                      #1#2\W\W {\XINT_htd_II_d {}{"#1#2}{\xint_c_ii^viii}}%
\def\XINT_htd_II_ciii #1#2#3\W {\XINT_htd_II_d {}{"#1#2#3}{\xint_c_ii^xii}}%
\def\XINT_htd_I_a #1#2#3#4#5#6%
{%
    \xint_gob_til_Z #3\XINT_htd_I_end_a\Z
    \expandafter\XINT_htd_I_b\the\numexpr
    #2+\xint_c_ii^xvi*#6#5#4#3+\xint_c_x^ix\relax {#1}%
}%
\def\XINT_htd_I_b 1#1#2#3#4#5#6#7#8#9{\XINT_htd_I_c {#1#2#3#4#5}{#9#8#7#6}}%
\def\XINT_htd_I_c #1#2#3{\XINT_htd_I_a {#3#2}{#1}}%
\def\XINT_htd_I_end_a\Z\expandafter\XINT_htd_I_b\the\numexpr #1+#2\relax
```

```
{%
    \expandafter\XINT_htd_I_end_b\the\numexpr \xint_c_x^v+#1\relax
}%
\def\XINT_htd_I_end_b 1#1#2#3#4#5%
{%
    \xint_gob_til_zero #1\XINT_htd_I_end_bz0%
    \XINT_htd_I_end_c #1#2#3#4#5%
}%
\def\XINT_htd_I_end_c #1#2#3#4#5#6{\XINT_htd_I {#6#5#4#3#2#1000}}%
\def\XINT_htd_I_end_bz0\XINT_htd_I_end_c 0#1#2#3#4%
{%
    \xint_gob_til_zeros_iv #1#2#3#4\XINT_htd_I_end_bzz 0000%
    \XINT_htd_I_end_D {#4#3#2#1}%
}%
\def\XINT_htd_I_end_D  #1#2{\XINT_htd_I {#2#1}}%
\def\XINT_htd_I_end_bzz 0000\XINT_htd_I_end_D #1{\XINT_htd_I }%
\def\XINT_htd_II_d #1#2#3#4#5#6#7%
{%
    \xint_gob_til_Z #4\XINT_htd_II_end_a\Z
    \expandafter\XINT_htd_II_e\the\numexpr
    #2+#3*#7#6#5#4+\xint_c_x^viii\relax {#1}{#3}%
}%
\def\XINT_htd_II_e 1#1#2#3#4#5#6#7#8{\XINT_htd_II_f {#1#2#3#4}{#5#6#7#8}}%
\def\XINT_htd_II_f #1#2#3{\XINT_htd_II_d {#2#3}{#1}}%
\def\XINT_htd_II_end_a\Z\expandafter\XINT_htd_II_e
    \the\numexpr #1+#2\relax #3#4\T
{%
    \XINT_htd_II_end_b #1#3%
}%
\edef\XINT_htd_II_end_b #1#2#3#4#5#6#7#8%
{%
    \noexpand\expandafter\space\noexpand\the\numexpr #1#2#3#4#5#6#7#8\relax
}%
```

33.8 \xintBinToDec

v1.08

```
\def\xintBinToDec {\romannumeral0\xintbintodec }%
\def\xintbintodec #1{\expandafter\XINT_btd_checkin
                     \romannumeral-`0#1\W\W\W\W\W\W\W\W \T }%
\def\XINT_btd_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_btd_neg
        -{\XINT_btd_I {000000}#1}%
     \krof
}%
\def\XINT_btd_neg {\expandafter\xint_minus_thenstop
```

```
                               \romannumeral0\XINT_btd_I {000000}}%
\def\XINT_btd_I #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_W #9\XINT_btd_II_a {#2#3#4#5#6#7#8#9}\W
    \XINT_btd_I_a {}{\csname XINT_sbtd_#2#3#4#5\endcsname*\xint_c_xvi+%
                     \csname XINT_sbtd_#6#7#8#9\endcsname}%
    #1\Z\Z\Z\Z\Z\Z
}%
\def\XINT_btd_II_a #1\W\XINT_btd_I_a #2#3{\XINT_btd_II_b #1}%
\def\XINT_btd_II_b #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_W
      #1\XINT_btd_II_c
      #2\XINT_btd_II_ci
      #3\XINT_btd_II_cii
      #4\XINT_btd_II_ciii
      #5\XINT_btd_II_civ
      #6\XINT_btd_II_cv
      #7\XINT_btd_II_cvi
      \W\XINT_btd_II_cvii #1#2#3#4#5#6#7#8%
}%
\def\XINT_btd_II_c #1\XINT_btd_II_cvii \W\W\W\W\W\W\W\W #2\Z\Z\Z\Z\Z\Z\T
{%
    \expandafter\XINT_btd_II_c_end
    \romannumeral0\XINT_rord_main {}#2%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax
}%
\edef\XINT_btd_II_c_end #1#2#3#4#5#6%
{%
    \noexpand\expandafter\space\noexpand\the\numexpr #1#2#3#4#5#6\relax
}%
\def\XINT_btd_II_ci  #1\XINT_btd_II_cvii #2\W\W\W\W\W\W\W
   {\XINT_btd_II_d {}{#2}{\xint_c_ii }}%
\def\XINT_btd_II_cii #1\XINT_btd_II_cvii #2\W\W\W\W\W\W
   {\XINT_btd_II_d {}{\csname XINT_sbtd_00#2\endcsname }{\xint_c_iv }}%
\def\XINT_btd_II_ciii #1\XINT_btd_II_cvii #2\W\W\W\W\W
   {\XINT_btd_II_d {}{\csname XINT_sbtd_0#2\endcsname }{\xint_c_viii }}%
\def\XINT_btd_II_civ #1\XINT_btd_II_cvii #2\W\W\W\W
   {\XINT_btd_II_d {}{\csname XINT_sbtd_#2\endcsname}{\xint_c_xvi }}%
\def\XINT_btd_II_cv #1\XINT_btd_II_cvii #2#3#4#5#6\W\W\W
{%
    \XINT_btd_II_d {}{\csname XINT_sbtd_#2#3#4#5\endcsname*\xint_c_ii+%
                          #6}{\xint_c_ii^v }%
}%
\def\XINT_btd_II_cvi #1\XINT_btd_II_cvii #2#3#4#5#6#7\W\W
{%
```

```
    \XINT_btd_II_d {}{\csname XINT_sbtd_#2#3#4#5\endcsname*\xint_c_iv+%
                      \csname XINT_sbtd_00#6#7\endcsname}{\xint_c_ii^vi }%
}%
\def\XINT_btd_II_cvii #1#2#3#4#5#6#7\W
{%
    \XINT_btd_II_d {}{\csname XINT_sbtd_#1#2#3#4\endcsname*\xint_c_viii+%
                      \csname XINT_sbtd_0#5#6#7\endcsname}{\xint_c_ii^vii }%
}%
\def\XINT_btd_II_d #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_Z #4\XINT_btd_II_end_a\Z
    \expandafter\XINT_btd_II_e\the\numexpr
    #2+(\xint_c_x^ix+#3*#9#8#7#6#5#4)\relax {#1}{#3}%
}%
\def\XINT_btd_II_e 1#1#2#3#4#5#6#7#8#9{\XINT_btd_II_f {#1#2#3}{#4#5#6#7#8#9}}%
\def\XINT_btd_II_f #1#2#3{\XINT_btd_II_d {#2#3}{#1}}%
\def\XINT_btd_II_end_a\Z\expandafter\XINT_btd_II_e
    \the\numexpr #1+(#2\relax #3#4\T
{%
    \XINT_btd_II_end_b #1#3%
}%
\edef\XINT_btd_II_end_b #1#2#3#4#5#6#7#8#9%
{%
    \noexpand\expandafter\space\noexpand\the\numexpr #1#2#3#4#5#6#7#8#9\relax
}%
\def\XINT_btd_I_a #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_Z #3\XINT_btd_I_end_a\Z
    \expandafter\XINT_btd_I_b\the\numexpr
    #2+\xint_c_ii^viii*#8#7#6#5#4#3+\xint_c_x^ix\relax {#1}%
}%
\def\XINT_btd_I_b 1#1#2#3#4#5#6#7#8#9{\XINT_btd_I_c {#1#2#3}{#9#8#7#6#5#4}}%
\def\XINT_btd_I_c #1#2#3{\XINT_btd_I_a {#3#2}{#1}}%
\def\XINT_btd_I_end_a\Z\expandafter\XINT_btd_I_b
    \the\numexpr #1+\xint_c_ii^viii #2\relax
{%
    \expandafter\XINT_btd_I_end_b\the\numexpr 1000+#1\relax
}%
\def\XINT_btd_I_end_b 1#1#2#3%
{%
    \xint_gob_til_zeros_iii #1#2#3\XINT_btd_I_end_bz 000%
    \XINT_btd_I_end_c #1#2#3%
}%
\def\XINT_btd_I_end_c #1#2#3#4{\XINT_btd_I {#4#3#2#1000}}%
\def\XINT_btd_I_end_bz 000\XINT_btd_I_end_c 000{\XINT_btd_I }%
```

33.9 \xintBinToHex

v1.08

```
\def\xintBinToHex {\romannumeral0\xintbintohex }%
\def\xintbintohex #1%
{%
    \expandafter\XINT_bth_checkin
                     \romannumeral0\expandafter\XINT_num_loop
                     \romannumeral-`0#1\xint_relax\xint_relax
                                       \xint_relax\xint_relax
                     \xint_relax\xint_relax\xint_relax\xint_relax\Z
    \R\R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W\W
}%
\def\XINT_bth_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_bth_N
        -{\XINT_bth_P #1}%
     \krof
}%
\def\XINT_bth_N {\expandafter\xint_minus_thenstop\romannumeral0\XINT_bth_P }%
\def\XINT_bth_P {\expandafter\XINT_bth_I\expandafter{\expandafter}%
                 \romannumeral0\XINT_OQ {}}%
\def\XINT_bth_I #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_W #9\XINT_bth_end_a\W
    \expandafter\expandafter\expandafter
    \XINT_bth_I
    \expandafter\expandafter\expandafter
    {\csname XINT_sbth_#9#8#7#6\expandafter\expandafter\expandafter\endcsname
     \csname XINT_sbth_#5#4#3#2\endcsname #1}%
}%
\def\XINT_bth_end_a\W \expandafter\expandafter\expandafter
    \XINT_bth_I       \expandafter\expandafter\expandafter #1%
{%
    \XINT_bth_end_b #1%
}%
\def\XINT_bth_end_b  #1\endcsname #2\endcsname #3%
{%
    \xint_gob_til_zero #3\XINT_bth_end_z 0\space #3%
}%
\def\XINT_bth_end_z0\space 0{ }%
```

33.10 \xintHexToBin

v1.08

```
\def\xintHexToBin {\romannumeral0\xinthextobin }%
\def\xinthextobin #1%
{%
    \expandafter\XINT_htb_checkin\romannumeral-`0#1GGGGGGGG\T
}%
```

```
\def\XINT_htb_checkin #1%
{%
    \xint_UDsignfork
       #1\XINT_htb_N
        -{\XINT_htb_P #1}%
     \krof
}%
\def\XINT_htb_N {\expandafter\xint_minus_thenstop\romannumeral0\XINT_htb_P }%
\def\XINT_htb_P {\XINT_htb_I_a {}}%
\def\XINT_htb_I_a #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_G #9\XINT_htb_II_a G%
    \expandafter\expandafter\expandafter
    \XINT_htb_I_b
    \expandafter\expandafter\expandafter
    {\csname XINT_shtb_#2\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#3\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#4\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#5\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#6\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#7\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#8\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#9\endcsname }{#1}%
}%
\def\XINT_htb_I_b #1#2{\XINT_htb_I_a {#2#1}}%
\def\XINT_htb_II_a G\expandafter\expandafter\expandafter\XINT_htb_I_b
{%
    \expandafter\expandafter\expandafter \XINT_htb_II_b
}%
\def\XINT_htb_II_b #1#2#3\T
{%
    \XINT_num_loop #2#1%
    \xint_relax\xint_relax\xint_relax\xint_relax
    \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
```

33.11 \xintCHexToBin

v1.08

```
\def\xintCHexToBin {\romannumeral0\xintchextobin }%
\def\xintchextobin #1%
{%
    \expandafter\XINT_chtb_checkin\romannumeral-`0#1%
    \R\R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W\W
}%
\def\XINT_chtb_checkin #1%
{%
    \xint_UDsignfork
```

```
      #1\XINT_chtb_N
        -{\XINT_chtb_P #1}%
      \krof
}%
\def\XINT_chtb_N {\expandafter\xint_minus_thenstop\romannumeral0\XINT_chtb_P }%
\def\XINT_chtb_P {\expandafter\XINT_chtb_I\expandafter{\expandafter}%
                  \romannumeral0\XINT_OQ {}}%
\def\XINT_chtb_I #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_W #9\XINT_chtb_end_a\W
    \expandafter\expandafter\expandafter
    \XINT_chtb_I
    \expandafter\expandafter\expandafter
    {\csname XINT_shtb_#9\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#8\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#7\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#6\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#5\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#4\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#3\expandafter\expandafter\expandafter\endcsname
     \csname XINT_shtb_#2\endcsname
     #1}%
}%
\def\XINT_chtb_end_a\W\expandafter\expandafter\expandafter
    \XINT_chtb_I\expandafter\expandafter\expandafter #1%
{%
    \XINT_chtb_end_b #1%
    \xint_relax\xint_relax\xint_relax\xint_relax
    \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\def\XINT_chtb_end_b #1\W#2\W#3\W#4\W#5\W#6\W#7\W#8\W\endcsname
{%
    \XINT_num_loop
}%
\XINT_restorecatcodes_endinput%
```

34 Package xintgcd implementation

The commenting is currently (2014/02/13) very sparse. Release 1.09h has modified a bit the \xintTypesetEuclideAlgorithm and \xintTypesetBezoutAlgorithm layout with respect to line indentation in particular. And they use the **xinttools** \xintloop rather than the Plain TeX or LaTeX's \loop.

Contents

34.1 Catcodes, ε-TeX and reload detection

The code for reload detection is copied from Heiko Oberdiek's packages, and adapted here to check for previous loading of the master **xint** package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \def\space { }%
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintgcd.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintgcd}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintgcd.sty
      \ifx\w\relax % but xint.sty not yet loaded.
         \y{xintgcd}{now issuing \string\input\space xint.sty}%
         \def\z{\endgroup\input xint.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
```

```
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xint.sty not yet loaded.
            \y{xintgcd}{now issuing \string\RequirePackage{xint}}%
            \def\z{\endgroup\RequirePackage{xint}}%
          \fi
      \else
        \y{xintgcd}{I was already loaded, aborting input}%
        \aftergroup\endinput
      \fi
    \fi
  \fi
\z%
```

34.2 Confirmation of xint loading

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \ifdefined\PackageInfo
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \else
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
  \fi
  \def\empty {}%
  \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
  \ifx\w\relax % Plain TeX, user gave a file name at the prompt
      \y{xintgcd}{Loading of package xint failed, aborting input}%
      \aftergroup\endinput
  \fi
  \ifx\w\empty % LaTeX, user gave a file name at the prompt
      \y{xintgcd}{Loading of package xint failed, aborting input}%
      \aftergroup\endinput
  \fi
\endgroup%
```

34.3 Catcodes

```
\XINTsetupcatcodes%
```

34.4 Package identification

```
\XINT_providespackage
```

```
\ProvidesPackage{xintgcd}%
  [2014/02/13 v1.09kb Euclide algorithm with xint package (jfB)]%
```

34.5 \xintGCD

The macros of 1.09a benefits from the \xintnum which has been inserted inside \xintiabs in xint; this is a little overhead but is more convenient for the user and also makes it easier to use into \xint-expressions.

```
\def\xintGCD {\romannumeral0\xintgcd }%
\def\xintgcd #1%
{%
    \expandafter\XINT_gcd\expandafter{\romannumeral0\xintiabs {#1}}%
}%
\def\XINT_gcd #1#2%
{%
    \expandafter\XINT_gcd_fork\romannumeral0\xintiabs {#2}\Z #1\Z
}%
```

Ici #3#4=A, #1#2=B

```
\def\XINT_gcd_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\XINT_gcd_BisZero
      #3\XINT_gcd_AisZero
       0\XINT_gcd_loop
    \krof
    {#1#2}{#3#4}%
}%
\def\XINT_gcd_AisZero #1#2{ #1}%
\def\XINT_gcd_BisZero #1#2{ #2}%
\def\XINT_gcd_CheckRem #1#2\Z
{%
    \xint_gob_til_zero #1\xint_gcd_end0\XINT_gcd_loop {#1#2}%
}%
\def\xint_gcd_end0\XINT_gcd_loop #1#2{ #2}%
```

#1=B, #2=A

```
\def\XINT_gcd_loop #1#2%
{%
    \expandafter\expandafter\expandafter
        \XINT_gcd_CheckRem
    \expandafter\xint_secondoftwo
    \romannumeral0\XINT_div_prepare {#1}{#2}\Z
    {#1}%
}%
```

34.6 \xintGCDof

New with 1.09a. I also tried an optimization (not working two by two) which I thought was clever but it seemed to be less efficient ...

```
\def\xintGCDof      {\romannumeral0\xintgcdof }%
\def\xintgcdof     #1{\expandafter\XINT_gcdof_a\romannumeral-`0#1\relax }%
\def\XINT_gcdof_a #1{\expandafter\XINT_gcdof_b\romannumeral-`0#1\Z }%
\def\XINT_gcdof_b #1\Z #2{\expandafter\XINT_gcdof_c\romannumeral-`0#2\Z {#1}\Z}%
\def\XINT_gcdof_c #1{\xint_gob_til_relax #1\XINT_gcdof_e\relax\XINT_gcdof_d #1}%
\def\XINT_gcdof_d #1\Z {\expandafter\XINT_gcdof_b\romannumeral0\xintgcd {#1}}%
\def\XINT_gcdof_e #1\Z #2\Z { #2}%
```

34.7 \xintLCM

New with 1.09a. Inadvertent use of \xintiQuo which was promoted at the same time to add the \xintnum overhead. So with 1.09f \xintiiQuo without the overhead.

```
\def\xintLCM {\romannumeral0\xintlcm}%
\def\xintlcm #1%
{%
    \expandafter\XINT_lcm\expandafter{\romannumeral0\xintiabs {#1}}%
}%
\def\XINT_lcm #1#2%
{%
    \expandafter\XINT_lcm_fork\romannumeral0\xintiabs {#2}\Z #1\Z
}%
\def\XINT_lcm_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\XINT_lcm_BisZero
      #3\XINT_lcm_AisZero
       0\expandafter
    \krof
    \XINT_lcm_notzero\expandafter{\romannumeral0\XINT_gcd_loop {#1#2}{#3#4}}%
    {#1#2}{#3#4}%
}%
\def\XINT_lcm_AisZero #1#2#3#4#5{ 0}%
\def\XINT_lcm_BisZero #1#2#3#4#5{ 0}%
\def\XINT_lcm_notzero #1#2#3{\xintiimul {#2}{\xintiiQuo{#3}{#1}}}%
```

34.8 \xintLCMof

New with 1.09a

```
\def\xintLCMof      {\romannumeral0\xintlcmof }%
\def\xintlcmof     #1{\expandafter\XINT_lcmof_a\romannumeral-`0#1\relax }%
\def\XINT_lcmof_a #1{\expandafter\XINT_lcmof_b\romannumeral-`0#1\Z }%
\def\XINT_lcmof_b #1\Z #2{\expandafter\XINT_lcmof_c\romannumeral-`0#2\Z {#1}\Z}%
```

```
\def\XINT_lcmof_c #1{\xint_gob_til_relax #1\XINT_lcmof_e\relax\XINT_lcmof_d #1}%
\def\XINT_lcmof_d #1\Z {\expandafter\XINT_lcmof_b\romannumeral0\xintlcm {#1}}%
\def\XINT_lcmof_e #1\Z #2\Z { #2}%
```

34.9 \xintBezout

1.09a inserts use of \xintnum

```
\def\xintBezout {\romannumeral0\xintbezout }%
\def\xintbezout #1%
{%
    \expandafter\xint_bezout\expandafter {\romannumeral0\xintnum{#1}}%
}%
\def\xint_bezout #1#2%
{%
    \expandafter\XINT_bezout_fork \romannumeral0\xintnum{#2}\Z #1\Z
}%
```

#3#4 = A, #1#2=B

```
\def\XINT_bezout_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerosfork
     #1#3\XINT_bezout_botharezero
      #10\XINT_bezout_secondiszero
      #30\XINT_bezout_firstiszero
       00{\xint_UDsignsfork
          #1#3\XINT_bezout_minusminus % A < 0, B < 0
           #1-\XINT_bezout_minusplus  % A > 0, B < 0
           #3-\XINT_bezout_plusminus  % A < 0, B > 0
            --\XINT_bezout_plusplus   % A > 0, B > 0
         \krof }%
    \krof
    {#2}{#4}#1#3{#3#4}{#1#2}% #1#2=B, #3#4=A
}%
\edef\XINT_bezout_botharezero #1#2#3#4#5#6%
{%
    \noexpand\xintError:NoBezoutForZeros
    \space {0}{0}{0}{0}{0}%
}%
```

attention première entrée doit être ici (-1)^n donc 1
#4#2 = 0 = A, B = #3#1

```
\def\XINT_bezout_firstiszero #1#2#3#4#5#6%
{%
    \xint_UDsignfork
      #3{ {0}{#3#1}{0}{1}{#1}}%
       -{ {0}{#3#1}{0}{-1}{#1}}%
```

```
    \krof
}%
```

#4#2 = A, B = #3#1 = 0

```
\def\XINT_bezout_secondiszero #1#2#3#4#5#6%
{%
    \xint_UDsignfork
       #4{ {#4#2}{0}{-1}{0}{#2}}%
        -{ {#4#2}{0}{1}{0}{#2}}%
    \krof
}%
```

#4#2= A < 0, #3#1 = B < 0

```
\def\XINT_bezout_minusminus #1#2#3#4%
{%
    \expandafter\XINT_bezout_mm_post
    \romannumeral0\XINT_bezout_loop_a 1{#1}{#2}1001%
}%
\def\XINT_bezout_mm_post #1#2%
{%
    \expandafter\XINT_bezout_mm_postb\expandafter
    {\romannumeral0\xintiiopp{#2}}{\romannumeral0\xintiiopp{#1}}%
}%
\def\XINT_bezout_mm_postb #1#2%
{%
    \expandafter\XINT_bezout_mm_postc\expandafter {#2}{#1}%
}%
\edef\XINT_bezout_mm_postc #1#2#3#4#5%
{%
    \space {#4}{#5}{#1}{#2}{#3}%
}%
```

minusplus #4#2= A > 0, B < 0

```
\def\XINT_bezout_minusplus #1#2#3#4%
{%
    \expandafter\XINT_bezout_mp_post
    \romannumeral0\XINT_bezout_loop_a 1{#1}{#4#2}1001%
}%
\def\XINT_bezout_mp_post #1#2%
{%
    \expandafter\XINT_bezout_mp_postb\expandafter
      {\romannumeral0\xintiiopp {#2}}{#1}%
}%
\edef\XINT_bezout_mp_postb #1#2#3#4#5%
{%
    \space {#4}{#5}{#2}{#1}{#3}%
}%
```

plusminus A < 0, B > 0

```
\def\XINT_bezout_plusminus #1#2#3#4%
{%
    \expandafter\XINT_bezout_pm_post
    \romannumeral0\XINT_bezout_loop_a 1{#3#1}{#2}1001%
}%
\def\XINT_bezout_pm_post #1%
{%
    \expandafter \XINT_bezout_pm_postb \expandafter
        {\romannumeral0\xintiiopp{#1}}%
}%
\edef\XINT_bezout_pm_postb #1#2#3#4#5%
{%
    \space {#4}{#5}{#1}{#2}{#3}%
}%
```

plusplus

```
\def\XINT_bezout_plusplus #1#2#3#4%
{%
    \expandafter\XINT_bezout_pp_post
    \romannumeral0\XINT_bezout_loop_a 1{#3#1}{#4#2}1001%
}%
```

la parité (-1)^N est en #1, et on la jette ici.

```
\edef\XINT_bezout_pp_post #1#2#3#4#5%
{%
    \space {#4}{#5}{#1}{#2}{#3}%
}%
```

n = 0: 1BAalpha(0)beta(0)alpha(-1)beta(-1)
n général: {(-1)^n}{r(n-1)}{r(n-2)}{alpha(n-1)}{beta(n-1)}{alpha(n-2)}{beta(n-2)}
#2 = B, #3 = A

```
\def\XINT_bezout_loop_a #1#2#3%
{%
    \expandafter\XINT_bezout_loop_b
    \expandafter{\the\numexpr -#1\expandafter }%
    \romannumeral0\XINT_div_prepare {#2}{#3}{#2}%
}%
```

Le q(n) a ici une existence éphémère, dans le version Bezout Algorithm il faudra le conserver. On voudra à la fin {{q(n)}{r(n)}{alpha(n)}{beta(n)}}. De plus ce n'est plus (-1)^n que l'on veut mais n. (ou dans un autre ordre) {-(-1)^n}{q(n)}{r(n)}{r(n-1)}{alpha(n-1)}{beta(n-1)}{alpha(n-2)}{beta(n-2)}

```
\def\XINT_bezout_loop_b #1#2#3#4#5#6#7#8%
```

```
{%
    \expandafter \XINT_bezout_loop_c \expandafter
        {\romannumeral0\xintiiadd{\XINT_Mul{#5}{#2}}{#7}}%
        {\romannumeral0\xintiiadd{\XINT_Mul{#6}{#2}}{#8}}%
    {#1}{#3}{#4}{#5}{#6}%
}%
```

{alpha(n)}{->beta(n)}{-(-1)^n}{r(n)}{r(n-1)}{alpha(n-1)}{beta(n-1)}

```
\def\XINT_bezout_loop_c #1#2%
{%
    \expandafter \XINT_bezout_loop_d \expandafter
        {#2}{#1}%
}%
```

{beta(n)}{alpha(n)}{(-1)^(n+1)}{r(n)}{r(n-1)}{alpha(n-1)}{beta(n-1)}

```
\def\XINT_bezout_loop_d #1#2#3#4#5%
{%
    \XINT_bezout_loop_e #4\Z {#3}{#5}{#2}{#1}%
}%
```

r(n)\Z {(-1)^(n+1)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}

```
\def\XINT_bezout_loop_e #1#2\Z
{%
    \xint_gob_til_zero #1\xint_bezout_loop_exit0\XINT_bezout_loop_f
    {#1#2}%
}%
```

{r(n)}{(-1)^(n+1)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}

```
\def\XINT_bezout_loop_f #1#2%
{%
    \XINT_bezout_loop_a {#2}{#1}%
}%
```

{(-1)^(n+1)}{r(n)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)} et itération

```
\def\xint_bezout_loop_exit0\XINT_bezout_loop_f #1#2%
{%
    \ifcase #2
    \or  \expandafter\XINT_bezout_exiteven
    \else\expandafter\XINT_bezout_exitodd
    \fi
}%
\edef\XINT_bezout_exiteven #1#2#3#4#5%
{%
    \space {#5}{#4}{#1}%
}%
```

```
\edef\XINT_bezout_exitodd #1#2#3#4#5%
{%
    \space {-#5}{-#4}{#1}%
}%
```

34.10 \xintEuclideAlgorithm

Pour Euclide: {N}{A}{D=r(n)}{B}{q1}{r1}{q2}{r2}{q3}{r3}....{qN}{rN=0}
u<2n> = u<2n+3>u<2n+2> + u<2n+4> à la n ième étape

```
\def\xintEuclideAlgorithm {\romannumeral0\xinteuclidealgorithm }%
\def\xinteuclidealgorithm #1%
{%
    \expandafter \XINT_euc \expandafter{\romannumeral0\xintiabs {#1}}%
}%
\def\XINT_euc #1#2%
{%
    \expandafter\XINT_euc_fork \romannumeral0\xintiabs {#2}\Z #1\Z
}%
```

Ici #3#4=A, #1#2=B

```
\def\XINT_euc_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\XINT_euc_BisZero
      #3\XINT_euc_AisZero
       0\XINT_euc_a
    \krof
    {0}{#1#2}{#3#4}{{#3#4}{#1#2}}{}\Z
}%
```

Le {} pour protéger {{A}{B}} si on s'arrête après une étape (B divise A). On va renvoyer:
{N}{A}{D=r(n)}{B}{q1}{r1}{q2}{r2}{q3}{r3}....{qN}{rN=0}

```
\def\XINT_euc_AisZero #1#2#3#4#5#6{ {1}{0}{#2}{#2}{0}{0}}%
\def\XINT_euc_BisZero #1#2#3#4#5#6{ {1}{0}{#3}{#3}{0}{0}}%
```

{n}{rn}{an}{{qn}{rn}}...{{A}{B}}{}\Z
a(n) = r(n-1). Pour n=0 on a juste {0}{B}{A}{{A}{B}}{}\Z
\XINT_div_prepare {u}{v} divise v par u

```
\def\XINT_euc_a #1#2#3%
{%
    \expandafter\XINT_euc_b
    \expandafter {\the\numexpr #1+1\expandafter }%
    \romannumeral0\XINT_div_prepare {#2}{#3}{#2}%
}%
```

{n+1}{q(n+1)}{r(n+1)}{rn}{{qn}{rn}}...

```
\def\XINT_euc_b #1#2#3#4%
{%
    \XINT_euc_c #3\Z {#1}{#3}{#4}{{#2}{#3}}%
}%
```

r(n+1)\Z {n+1}{r(n+1)}{r(n)}{{q(n+1)}{r(n+1)}}{{qn}{rn}}...
Test si r(n+1) est nul.

```
\def\XINT_euc_c #1#2\Z
{%
    \xint_gob_til_zero #1\xint_euc_end0\XINT_euc_a
}%
```

{n+1}{r(n+1)}{r(n)}{{q(n+1)}{r(n+1)}}...{}\Z Ici r(n+1) = 0. On arrête on se prépare à inverser {n+1}{0}{r(n)}{{q(n+1)}{r(n+1)}}.....{{q1}{r1}}{{A}{B}}{}\Z On veut renvoyer: {N=n+1}{A}{D=r(n)}{B}{q1}{r1}{q2}{r2}{q3}{r3}....{qN}{rN=0}

```
\def\xint_euc_end0\XINT_euc_a #1#2#3#4\Z%
{%
    \expandafter\xint_euc_end_
    \romannumeral0%
    \XINT_rord_main {}#4{{#1}{#3}}%
    \xint_relax
      \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_bye\xint_bye\xint_bye\xint_bye
    \xint_relax
}%
\edef\xint_euc_end_ #1#2#3%
{%
    \space {#1}{#3}{#2}%
}%
```

34.11 \xintBezoutAlgorithm

Pour Bezout: objectif, renvoyer
{N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
{q2}{r2}{alpha2}{beta2}....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}
alpha0=1, beta0=0, alpha(-1)=0, beta(-1)=1

```
\def\xintBezoutAlgorithm {\romannumeral0\xintbezoutalgorithm }%
\def\xintbezoutalgorithm #1%
{%
    \expandafter \XINT_bezalg \expandafter{\romannumeral0\xintiabs {#1}}%
}%
\def\XINT_bezalg #1#2%
{%
    \expandafter\XINT_bezalg_fork \romannumeral0\xintiabs {#2}\Z #1\Z
}%
```

Ici #3#4=A, #1#2=B

```
\def\XINT_bezalg_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\XINT_bezalg_BisZero
      #3\XINT_bezalg_AisZero
       0\XINT_bezalg_a
    \krof
    0{#1#2}{#3#4}1001{{#3#4}{#1#2}}{}\Z
}%
\def\XINT_bezalg_AisZero #1#2#3\Z{ {1}{0}{0}{1}{#2}{#2}{1}{0}{0}{0}{0}{1}}%
\def\XINT_bezalg_BisZero #1#2#3#4\Z{ {1}{0}{0}{1}{#3}{#3}{1}{0}{0}{0}{0}{1}}%
```

pour préparer l'étape n+1 il faut {n}{r(n)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}{{q(n)}{r(n)}{alpha(n)}{beta(n)}}... division de #3 par #2

```
\def\XINT_bezalg_a #1#2#3%
{%
    \expandafter\XINT_bezalg_b
    \expandafter {\the\numexpr #1+1\expandafter }%
    \romannumeral0\XINT_div_prepare {#2}{#3}{#2}%
}%
```

{n+1}{q(n+1)}{r(n+1)}{r(n)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}...

```
\def\XINT_bezalg_b #1#2#3#4#5#6#7#8%
{%
    \expandafter\XINT_bezalg_c\expandafter
     {\romannumeral0\xintiiadd {\xintiiMul {#6}{#2}}{#8}}%
     {\romannumeral0\xintiiadd {\xintiiMul {#5}{#2}}{#7}}%
     {#1}{#2}{#3}{#4}{#5}{#6}%
}%
```

{beta(n+1)}{alpha(n+1)}{n+1}{q(n+1)}{r(n+1)}{r(n)}{alpha(n)}{beta(n}}

```
\def\XINT_bezalg_c #1#2#3#4#5#6%
{%
    \expandafter\XINT_bezalg_d\expandafter {#2}{#3}{#4}{#5}{#6}{#1}%
}%
```

{alpha(n+1)}{n+1}{q(n+1)}{r(n+1)}{r(n)}{beta(n+1)}

```
\def\XINT_bezalg_d #1#2#3#4#5#6#7#8%
{%
    \XINT_bezalg_e #4\Z {#2}{#4}{#5}{#1}{#6}{#7}{#8}{{#3}{#4}{#1}{#6}}%
}%
```

r(n+1)\Z {n+1}{r(n+1)}{r(n)}{alpha(n+1)}{beta(n+1)}
{alpha(n)}{beta(n)}{q,r,alpha,beta(n+1)}
Test si r(n+1) est nul.

```
\def\XINT_bezalg_e #1#2\Z
{%
    \xint_gob_til_zero #1\xint_bezalg_end0\XINT_bezalg_a
}%
```

Ici r(n+1) = 0. On arrête on se prépare à inverser.
{n+1}{r(n+1)}{r(n)}{alpha(n+1)}{beta(n+1)}{alpha(n)}{beta(n)}
{q,r,alpha,beta(n+1)}...{{A}{B}}{}\Z
On veut renvoyer
{N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
{q2}{r2}{alpha2}{beta2}....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}

```
\def\xint_bezalg_end0\XINT_bezalg_a #1#2#3#4#5#6#7#8\Z
{%
    \expandafter\xint_bezalg_end_
    \romannumeral0%
    \XINT_rord_main {}#8{{#1}{#3}}%
    \xint_relax
      \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_bye\xint_bye\xint_bye\xint_bye
    \xint_relax
}%
```

{N}{D}{A}{B}{q1}{r1}{alpha1=q1}{beta1=1}{q2}{r2}{alpha2}{beta2}
....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}
On veut renvoyer
{N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
{q2}{r2}{alpha2}{beta2}....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}

```
\edef\xint_bezalg_end_ #1#2#3#4%
{%
    \space {#1}{#3}{0}{1}{#2}{#4}{1}{0}%
}%
```

34.12 \xintTypesetEuclideAlgorithm

TYPESETTING
  Organisation:
  {N}{A}{D}{B}{q1}{r1}{q2}{r2}{q3}{r3}....{qN}{rN=0}
\U1 = N = nombre d’étapes, \U3 = PGCD, \U2 = A, \U4=B q1 = \U5, q2 = \U7 --> qn = \U<2n+3>, rn = \U<2n+4> bn = rn. B = r0. A=r(-1)
  r(n-2) = q(n)r(n-1)+r(n) (n e étape)
  \U{2n} = \U{2n+3} \times \U{2n+2} + \U{2n+4}, n e étape. (avec n entre 1 et N)
  1.09h uses \xintloop, and \par rather than \endgraf; and \par rather than \hfill\break

```
\def\xintTypesetEuclideAlgorithm #1#2%
{% l’algo remplace #1 et #2 par |#1| et |#2|
  \par
```

```
  \begingroup
    \xintAssignArray\xintEuclideAlgorithm {#1}{#2}\to\U
    \edef\A{\U2}\edef\B{\U4}\edef\N{\U1}%
    \setbox 0 \vbox{\halign {$##$\cr \A\cr \B \cr}}%
    \count 255 1
    \xintloop
      \indent\hbox to \wd 0 {\hfil$\U{\numexpr 2*\count255\relax}$}%
      ${} =  \U{\numexpr 2*\count255 + 3\relax}
      \times \U{\numexpr 2*\count255 + 2\relax}
          +  \U{\numexpr 2*\count255 + 4\relax}$%
    \ifnum \count255 < \N
      \par
      \advance \count255 1
    \repeat
  \endgroup
}%
```

34.13 \xintTypesetBezoutAlgorithm

Pour Bezout on a: {N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1} {q2}{r2}{alpha2}{beta2}....{qN}{rN=0}{alphaN=A/D}{betaN=B/D} Donc 4N+8 termes: U1 = N, U2= A, U5=D, U6=B, q1 = U9, qn = U{4n+5}, n au moins 1
rn = U{4n+6}, n au moins -1
alpha(n) = U{4n+7}, n au moins -1
beta(n) = U{4n+8}, n au moins -1

1.09h uses \xintloop, and \par rather than \endgraf; and no more \parindent0pt

```
\def\xintTypesetBezoutAlgorithm #1#2%
{%
  \par
  \begingroup
    \xintAssignArray\xintBezoutAlgorithm {#1}{#2}\to\BEZ
    \edef\A{\BEZ2}\edef\B{\BEZ6}\edef\N{\BEZ1}% A = |#1|, B = |#2|
    \setbox 0 \vbox{\halign {$##$\cr \A\cr \B \cr}}%
    \count255 1
    \xintloop
      \indent\hbox to \wd 0 {\hfil$\BEZ{4*\count255 - 2}$}%
      ${} =  \BEZ{4*\count255 + 5}
      \times \BEZ{4*\count255 + 2}
          +  \BEZ{4*\count255 + 6}$\hfill\break
      \hbox to \wd 0 {\hfil$\BEZ{4*\count255 +7}$}%
      ${} = \BEZ{4*\count255 + 5}
      \times \BEZ{4*\count255 + 3}
          +  \BEZ{4*\count255 - 1}$\hfill\break
      \hbox to \wd 0 {\hfil$\BEZ{4*\count255 +8}$}%
      ${} =  \BEZ{4*\count255 + 5}
      \times \BEZ{4*\count255 + 4}
          +  \BEZ{4*\count255 }$
      \par
```

```
    \ifnum \count255 < \N
    \advance \count255 1
  \repeat
    \edef\U{\BEZ{4*\N + 4}}%
    \edef\V{\BEZ{4*\N + 3}}%
    \edef\D{\BEZ5}%
    \ifodd\N
       $\U\times\A  - \V\times \B = -\D$%
    \else
       $\U\times\A - \V\times\B = \D$%
    \fi
    \par
  \endgroup
}%
```

34.14 \xintGCDof:csv

1.09a. For use by \xintexpr.

```
\def\xintGCDof:csv #1{\expandafter\XINT_gcdof:_b\romannumeral-`0#1,,}%
\def\XINT_gcdof:_b #1,#2,{\expandafter\XINT_gcdof:_c\romannumeral-`0#2,{#1},}%
\def\XINT_gcdof:_c #1{\if #1,\expandafter\XINT_of:_e
                       \else\expandafter\XINT_gcdof:_d\fi #1}%
\def\XINT_gcdof:_d #1,{\expandafter\XINT_gcdof:_b\romannumeral0\xintgcd {#1}}%
```

34.15 \xintLCMof:csv

1.09a. For use by \xintexpr.

```
\def\xintLCMof:csv #1{\expandafter\XINT_lcmof:_a\romannumeral-`0#1,,}%
\def\XINT_lcmof:_a #1,#2,{\expandafter\XINT_lcmof:_c\romannumeral-`0#2,{#1},}%
\def\XINT_lcmof:_c #1{\if#1,\expandafter\XINT_of:_e
                       \else\expandafter\XINT_lcmof:_d\fi #1}%
\def\XINT_lcmof:_d #1,{\expandafter\XINT_lcmof:_a\romannumeral0\xintlcm {#1}}%
\XINT_restorecatcodes_endinput%
```

35 Package xintfrac implementation

The commenting is currently (2014/02/13) very sparse.

Contents

35.1 Catcodes, ε-TEX and reload detection

The code for reload detection is copied from Heiko Oberdiek's packages, and adapted here to check for previous loading of the master xint package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
```

```
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \def\space { }%
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintfrac.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintfrac}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintfrac.sty
      \ifx\w\relax % but xint.sty not yet loaded.
         \y{xintfrac}{now issuing \string\input\space xint.sty}%
         \def\z{\endgroup\input xint.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xint.sty not yet loaded.
            \y{xintfrac}{now issuing \string\RequirePackage{xint}}%
            \def\z{\endgroup\RequirePackage{xint}}%
          \fi
      \else
        \y{xintfrac}{I was already loaded, aborting input}%
        \aftergroup\endinput
      \fi
    \fi
  \fi
\z%
```

35.2 Confirmation of xint loading

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
```

```
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \ifdefined\PackageInfo
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \else
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
  \fi
  \def\empty {}%
  \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
  \ifx\w\relax % Plain TeX, user gave a file name at the prompt
      \y{xintfrac}{Loading of package xint failed, aborting input}%
      \aftergroup\endinput
  \fi
  \ifx\w\empty % LaTeX, user gave a file name at the prompt
      \y{xintfrac}{Loading of package xint failed, aborting input}%
      \aftergroup\endinput
  \fi
\endgroup%
```

35.3 Catcodes

```
\XINTsetupcatcodes%
```

35.4 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintfrac}%
  [2014/02/13 v1.09kb Expandable operations on fractions (jfB)]%
\chardef\xint_c_vi     6
\chardef\xint_c_vii    7
\chardef\xint_c_xviii  18
```

35.5 \xintLen

```
\def\xintLen {\romannumeral0\xintlen }%
\def\xintlen #1%
{%
    \expandafter\XINT_flen\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_flen #1#2#3%
{%
    \expandafter\space
    \the\numexpr -1+\XINT_Abs {#1}+\XINT_Len {#2}+\XINT_Len {#3}\relax
}%
```

35.6 \XINT_lenrord_loop

```
\def\XINT_lenrord_loop #1#2#3#4#5#6#7#8#9%
{%  faire \romannumeral-`0\XINT_lenrord_loop 0{}#1\Z\W\W\W\W\W\W\W\Z
    \xint_gob_til_W #9\XINT_lenrord_W\W
    \expandafter\XINT_lenrord_loop\expandafter
    {\the\numexpr #1+7}{#9#8#7#6#5#4#3#2}%
}%
\def\XINT_lenrord_W\W\expandafter\XINT_lenrord_loop\expandafter #1#2#3\Z
{%
    \expandafter\XINT_lenrord_X\expandafter {#1}#2\Z
}%
\def\XINT_lenrord_X #1#2\Z
{%
    \XINT_lenrord_Y #2\R\R\R\R\R\R\T {#1}%
}%
\def\XINT_lenrord_Y #1#2#3#4#5#6#7#8\T
{%
    \xint_gob_til_W
            #7\XINT_lenrord_Z \xint_c_viii
            #6\XINT_lenrord_Z \xint_c_vii
            #5\XINT_lenrord_Z \xint_c_vi
            #4\XINT_lenrord_Z \xint_c_v
            #3\XINT_lenrord_Z \xint_c_iv
            #2\XINT_lenrord_Z \xint_c_iii
            \W\XINT_lenrord_Z \xint_c_ii   \Z
}%
\def\XINT_lenrord_Z #1#2\Z #3% retourne: {longueur}renverse\Z
{%
    \expandafter{\the\numexpr #3-#1\relax}%
}%
```

35.7 \XINT_outfrac

1.06a version now outputs 0/1[0] and not 0[0] in case of zero. More generally all macros have been checked in xintfrac, xintseries, xintcfrac, to make sure the output format for fractions was always A/B[n]. (except \xintIrr, \xintJrr, \xintRawWithZeros)

The problem with statements like those in the previous paragraph is that it is hard to maintain consistencies across relases.

```
\def\XINT_outfrac #1#2#3%
{%
    \ifcase\XINT_cntSgn #3\Z
        \expandafter \XINT_outfrac_divisionbyzero
    \or
        \expandafter \XINT_outfrac_P
    \else
        \expandafter \XINT_outfrac_N
    \fi
    {#2}{#3}[#1]%
}%
```

```
\def\XINT_outfrac_divisionbyzero #1#2{\xintError:DivisionByZero\space #1/0}%
\edef\XINT_outfrac_P #1#2%
{%
    \noexpand\if0\noexpand\XINT_Sgn #1\noexpand\Z
        \noexpand\expandafter\noexpand\XINT_outfrac_Zero
    \noexpand\fi
    \space #1/#2%
}%
\def\XINT_outfrac_Zero #1[#2]{ 0/1[0]}%
\def\XINT_outfrac_N #1#2%
{%
    \expandafter\XINT_outfrac_N_a\expandafter
    {\romannumeral0\XINT_opp #2}{\romannumeral0\XINT_opp #1}%
}%
\def\XINT_outfrac_N_a #1#2%
{%
    \expandafter\XINT_outfrac_P\expandafter {#2}{#1}%
}%
```

35.8 \XINT_inFrac

Extended in 1.07 to accept scientific notation on input. With lowercase e only. The \xintexpr parser does accept uppercase E also.

```
\def\XINT_inFrac {\romannumeral0\XINT_infrac }%
\def\XINT_infrac #1%
{%
    \expandafter\XINT_infrac_ \romannumeral-`0#1[\W]\Z\T
}%
\def\XINT_infrac_ #1[#2#3]#4\Z
{%
    \xint_UDwfork
      #2\XINT_infrac_A
      \W\XINT_infrac_B
    \krof
    #1[#2#3]#4%
}%
\def\XINT_infrac_A #1[\W]\T
{%
    \XINT_frac #1/\W\Z
}%
\def\XINT_infrac_B #1%
{%
    \xint_gob_til_zero #1\XINT_infrac_Zero0\XINT_infrac_BB #1%
}%
\def\XINT_infrac_BB #1[\W]\T {\XINT_infrac_BC #1/\W\Z }%
\def\XINT_infrac_BC #1/#2#3\Z
{%
    \xint_UDwfork
```

```
     #2\XINT_infrac_BCa
     \W{\expandafter\XINT_infrac_BCb \romannumeral-`0#2}%
    \krof
    #3\Z #1\Z
}%
\def\XINT_infrac_BCa \Z #1[#2]#3\Z { {#2}{#1}{1}}%
\def\XINT_infrac_BCb #1[#2]/\W\Z #3\Z { {#2}{#3}{#1}}%
\def\XINT_infrac_Zero #1\T { {0}{0}{1}}%
```

35.9 \XINT_frac

Extended in 1.07 to recognize and accept scientific notation both at the numerator and (possible) denominator. Only a lowercase e will do here, but uppercase E is possible within an \xintexpr..\relax

```
\def\XINT_frac #1/#2#3\Z
{%
    \xint_UDwfork
     #2\XINT_frac_A
     \W{\expandafter\XINT_frac_U \romannumeral-`0#2}%
    \krof
    #3e\W\Z #1e\W\Z
}%
\def\XINT_frac_U #1e#2#3\Z
{%
    \xint_UDwfork
      #2\XINT_frac_Ua
      \W{\XINT_frac_Ub #2}%
    \krof
    #3\Z #1\Z
}%
\def\XINT_frac_Ua      \Z #1/\W\Z  {\XINT_frac_B #1.\W\Z {0}}%
\def\XINT_frac_Ub #1/\W e\W\Z #2\Z {\XINT_frac_B #2.\W\Z {#1}}%
\def\XINT_frac_B #1.#2#3\Z
{%
    \xint_UDwfork
      #2\XINT_frac_Ba
      \W{\XINT_frac_Bb #2}%
    \krof
    #3\Z #1\Z
}%
\def\XINT_frac_Ba \Z #1\Z {\XINT_frac_T {0}{#1}}%
\def\XINT_frac_Bb #1.\W\Z #2\Z
{%
    \expandafter \XINT_frac_T \expandafter
    {\romannumeral0\xintlength {#1}}{#2#1}%
}%
\def\XINT_frac_A e\W\Z {\XINT_frac_T {0}{1}{0}}%
\def\XINT_frac_T #1#2#3#4e#5#6\Z
```

```
{%
    \xint_UDwfork
      #5\XINT_frac_Ta
      \W{\XINT_frac_Tb #5}%
    \krof
    #6\Z #4\Z {#1}{#2}{#3}%
}%
\def\XINT_frac_Ta \Z #1\Z      {\XINT_frac_C #1.\W\Z {0}}%
\def\XINT_frac_Tb #1e\W\Z #2\Z {\XINT_frac_C #2.\W\Z {#1}}%
\def\XINT_frac_C #1.#2#3\Z
{%
    \xint_UDwfork
      #2\XINT_frac_Ca
      \W{\XINT_frac_Cb #2}%
    \krof
    #3\Z #1\Z
}%
\def\XINT_frac_Ca \Z #1\Z {\XINT_frac_D {0}{#1}}%
\def\XINT_frac_Cb #1.\W\Z #2\Z
{%
    \expandafter\XINT_frac_D\expandafter
    {\romannumeral0\xintlength {#1}}{#2#1}%
}%
\def\XINT_frac_D #1#2#3#4#5#6%
{%
    \expandafter \XINT_frac_E \expandafter
    {\the\numexpr -#1+#3+#4-#6\expandafter}\expandafter
    {\romannumeral0\XINT_num_loop #2%
     \xint_relax\xint_relax\xint_relax\xint_relax
     \xint_relax\xint_relax\xint_relax\xint_relax\Z }%
    {\romannumeral0\XINT_num_loop #5%
     \xint_relax\xint_relax\xint_relax\xint_relax
     \xint_relax\xint_relax\xint_relax\xint_relax\Z }%
}%
\def\XINT_frac_E #1#2#3%
{%
   \expandafter \XINT_frac_F  #3\Z {#2}{#1}%
}%
\def\XINT_frac_F #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_frac_Gdivisionbyzero
      0#1\XINT_frac_Gneg
       0-{\XINT_frac_Gpos #1}%
    \krof
}%
\edef\XINT_frac_Gdivisionbyzero #1\Z #2#3%
{%
   \noexpand\xintError:DivisionByZero\space {0}{#2}{0}%
```

```
}%
\def\XINT_frac_Gneg #1\Z #2#3%
{%
    \expandafter\XINT_frac_H \expandafter{\romannumeral0\XINT_opp #2}{#3}{#1}%
}%
\def\XINT_frac_H #1#2{ {#2}{#1}}%
\def\XINT_frac_Gpos #1\Z #2#3{ {#3}{#2}{#1}}%
```

35.10 \XINT_factortens, \XINT_cuz_cnt

```
\def\XINT_factortens #1%
{%
    \expandafter\XINT_cuz_cnt_loop\expandafter
    {\expandafter}\romannumeral0\XINT_rord_main {}#1%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax
    \R\R\R\R\R\R\R\R\Z
}%
\def\XINT_cuz_cnt #1%
{%
    \XINT_cuz_cnt_loop {}#1\R\R\R\R\R\R\R\R\Z
}%
\def\XINT_cuz_cnt_loop #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_R #9\XINT_cuz_cnt_toofara \R
    \expandafter\XINT_cuz_cnt_checka\expandafter
    {\the\numexpr #1+8\relax}{#2#3#4#5#6#7#8#9}%
}%
\def\XINT_cuz_cnt_toofara\R
    \expandafter\XINT_cuz_cnt_checka\expandafter #1#2%
{%
    \XINT_cuz_cnt_toofarb {#1}#2%
}%
\def\XINT_cuz_cnt_toofarb #1#2\Z {\XINT_cuz_cnt_toofarc #2\Z {#1}}%
\def\XINT_cuz_cnt_toofarc #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_R #2\XINT_cuz_cnt_toofard 7%
            #3\XINT_cuz_cnt_toofard 6%
            #4\XINT_cuz_cnt_toofard 5%
            #5\XINT_cuz_cnt_toofard 4%
            #6\XINT_cuz_cnt_toofard 3%
            #7\XINT_cuz_cnt_toofard 2%
            #8\XINT_cuz_cnt_toofard 1%
            \Z #1#2#3#4#5#6#7#8%
}%
\def\XINT_cuz_cnt_toofard #1#2\Z #3\R #4\Z #5%
{%
```

```
    \expandafter\XINT_cuz_cnt_toofare
    \the\numexpr #3\relax \R\R\R\R\R\R\R\R\Z
    {\the\numexpr #5-#1\relax}\R\Z
}%
\def\XINT_cuz_cnt_toofare #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_R #2\XINT_cuz_cnt_stopc 1%
            #3\XINT_cuz_cnt_stopc 2%
            #4\XINT_cuz_cnt_stopc 3%
            #5\XINT_cuz_cnt_stopc 4%
            #6\XINT_cuz_cnt_stopc 5%
            #7\XINT_cuz_cnt_stopc 6%
            #8\XINT_cuz_cnt_stopc 7%
            \Z #1#2#3#4#5#6#7#8%
}%
\def\XINT_cuz_cnt_checka #1#2%
{%
    \expandafter\XINT_cuz_cnt_checkb\the\numexpr #2\relax \Z {#1}%
}%
\def\XINT_cuz_cnt_checkb #1%
{%
    \xint_gob_til_zero #1\expandafter\XINT_cuz_cnt_loop\xint_gob_til_Z
    0\XINT_cuz_cnt_stopa #1%
}%
\def\XINT_cuz_cnt_stopa #1\Z
{%
    \XINT_cuz_cnt_stopb #1\R\R\R\R\R\R\R\R\Z %
}%
\def\XINT_cuz_cnt_stopb #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_R #2\XINT_cuz_cnt_stopc 1%
            #3\XINT_cuz_cnt_stopc 2%
            #4\XINT_cuz_cnt_stopc 3%
            #5\XINT_cuz_cnt_stopc 4%
            #6\XINT_cuz_cnt_stopc 5%
            #7\XINT_cuz_cnt_stopc 6%
            #8\XINT_cuz_cnt_stopc 7%
            #9\XINT_cuz_cnt_stopc 8%
            \Z #1#2#3#4#5#6#7#8#9%
}%
\def\XINT_cuz_cnt_stopc #1#2\Z #3\R #4\Z #5%
{%
    \expandafter\XINT_cuz_cnt_stopd\expandafter
    {\the\numexpr #5-#1}#3%
}%
\def\XINT_cuz_cnt_stopd #1#2\R #3\Z
{%
    \expandafter\space\expandafter
     {\romannumeral0\XINT_rord_main {}#2%
```

```
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax }{#1}%
}%
```

35.11 \xintRaw

1.07: this macro simply prints in a user readable form the fraction after its initial scanning. Useful when put inside braces in an \xintexpr, when the input is not yet in the A/B[n] form.

```
\def\xintRaw {\romannumeral0\xintraw }%
\def\xintraw
{%
    \expandafter\XINT_raw\romannumeral0\XINT_infrac
}%
\def\XINT_raw #1#2#3{ #2/#3[#1]}%
```

35.12 \xintPRaw

1.09b: these [n]'s and especially the possible /1 are truly annoying at times.

```
\def\xintPRaw {\romannumeral0\xintpraw }%
\def\xintpraw
{%
    \expandafter\XINT_praw\romannumeral0\XINT_infrac
}%
\def\XINT_praw #1%
{%
    \ifnum #1=\xint_c_ \expandafter\XINT_praw_a\fi \XINT_praw_A {#1}%
}%
\def\XINT_praw_A #1#2#3%
{%
    \if\XINT_isOne{#3}1\expandafter\xint_firstoftwo
                \else\expandafter\xint_secondoftwo
    \fi { #2[#1]}{ #2/#3[#1]}%
}%
\def\XINT_praw_a\XINT_praw_A #1#2#3%
{%
    \if\XINT_isOne{#3}1\expandafter\xint_firstoftwo
                \else\expandafter\xint_secondoftwo
    \fi { #2}{ #2/#3}%
}%
```

35.13 \xintRawWithZeros

This was called \xintRaw in versions earlier than 1.07

```
\def\xintRawWithZeros {\romannumeral0\xintrawwithzeros }%
\def\xintrawwithzeros
{%
    \expandafter\XINT_rawz\romannumeral0\XINT_infrac
}%
\def\XINT_rawz #1%
{%
    \ifcase\XINT_cntSgn #1\Z
      \expandafter\XINT_rawz_Ba
    \or
      \expandafter\XINT_rawz_A
    \else
      \expandafter\XINT_rawz_Ba
    \fi
    {#1}%
}%
\def\XINT_rawz_A  #1#2#3{\xint_dsh {#2}{-#1}/#3}%
\def\XINT_rawz_Ba #1#2#3{\expandafter\XINT_rawz_Bb
                        \expandafter{\romannumeral0\xint_dsh {#3}{#1}}{#2}}%
\def\XINT_rawz_Bb #1#2{ #2/#1}%
```

35.14 \xintFloor

1.09a

```
\def\xintFloor {\romannumeral0\xintfloor }%
\def\xintfloor #1{\expandafter\XINT_floor
                  \romannumeral0\xintrawwithzeros {#1}.}%
\def\XINT_floor #1/#2.{\xintiiquo {#1}{#2}}%
```

35.15 \xintCeil

1.09a

```
\def\xintCeil {\romannumeral0\xintceil }%
\def\xintceil #1{\xintiiopp {\xintFloor {\xintOpp{#1}}}}%
```

35.16 \xintNumerator

```
\def\xintNumerator {\romannumeral0\xintnumerator }%
\def\xintnumerator
{%
    \expandafter\XINT_numer\romannumeral0\XINT_infrac
}%
\def\XINT_numer #1%
{%
    \ifcase\XINT_cntSgn #1\Z
      \expandafter\XINT_numer_B
```

```
    \or
      \expandafter\XINT_numer_A
    \else
      \expandafter\XINT_numer_B
    \fi
    {#1}%
}%
\def\XINT_numer_A #1#2#3{\xint_dsh {#2}{-#1}}%
\def\XINT_numer_B #1#2#3{ #2}%
```

35.17 \xintDenominator

```
\def\xintDenominator {\romannumeral0\xintdenominator }%
\def\xintdenominator
{%
    \expandafter\XINT_denom\romannumeral0\XINT_infrac
}%
\def\XINT_denom #1%
{%
    \ifcase\XINT_cntSgn #1\Z
      \expandafter\XINT_denom_B
    \or
      \expandafter\XINT_denom_A
    \else
      \expandafter\XINT_denom_B
    \fi
    {#1}%
}%
\def\XINT_denom_A #1#2#3{ #3}%
\def\XINT_denom_B #1#2#3{\xint_dsh {#3}{#1}}%
```

35.18 \xintFrac

```
\def\xintFrac {\romannumeral0\xintfrac }%
\def\xintfrac #1%
{%
    \expandafter\XINT_fracfrac_A\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_fracfrac_A #1{\XINT_fracfrac_B #1\Z }%
\catcode`^=7
\def\XINT_fracfrac_B #1#2\Z
{%
    \xint_gob_til_zero #1\XINT_fracfrac_C 0\XINT_fracfrac_D {10^{#1#2}}%
}%
\def\XINT_fracfrac_C 0\XINT_fracfrac_D #1#2#3%
{%
    \if1\XINT_isOne {#3}%
        \xint_afterfi {\expandafter\xint_firstoftwo_thenstop\xint_gobble_ii }%
    \fi
    \space
    \frac {#2}{#3}%
```

```
}%
\def\XINT_fracfrac_D #1#2#3%
{%
    \if1\XINT_isOne {#3}\XINT_fracfrac_E\fi
    \space
    \frac {#2}{#3}#1%
}%
\def\XINT_fracfrac_E \fi\space\frac #1#2{\fi \space #1\cdot }%
```

35.19 \xintSignedFrac

```
\def\xintSignedFrac {\romannumeral0\xintsignedfrac }%
\def\xintsignedfrac #1%
{%
    \expandafter\XINT_sgnfrac_a\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_sgnfrac_a #1#2%
{%
    \XINT_sgnfrac_b #2\Z {#1}%
}%
\def\XINT_sgnfrac_b #1%
{%
    \xint_UDsignfork
      #1\XINT_sgnfrac_N
       -{\XINT_sgnfrac_P #1}%
    \krof
}%
\def\XINT_sgnfrac_P #1\Z #2%
{%
    \XINT_fracfrac_A {#2}{#1}%
}%
\def\XINT_sgnfrac_N
{%
    \expandafter\xint_minus_thenstop\romannumeral0\XINT_sgnfrac_P
}%
```

35.20 \xintFwOver

```
\def\xintFwOver {\romannumeral0\xintfwover }%
\def\xintfwover #1%
{%
    \expandafter\XINT_fwover_A\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_fwover_A #1{\XINT_fwover_B #1\Z }%
\def\XINT_fwover_B #1#2\Z
{%
    \xint_gob_til_zero #1\XINT_fwover_C 0\XINT_fwover_D {10^{#1#2}}%
}%
\catcode`^=11
\def\XINT_fwover_C #1#2#3#4#5%
{%
```

```
    \if0\XINT_isOne {#5}\xint_afterfi { {#4\over #5}}%
                   \else\xint_afterfi { #4}%
    \fi
}%
\def\XINT_fwover_D #1#2#3%
{%
    \if0\XINT_isOne {#3}\xint_afterfi { {#2\over #3}}%
                   \else\xint_afterfi { #2\cdot }%
    \fi
    #1%
}%
```

35.21 \xintSignedFwOver

```
\def\xintSignedFwOver {\romannumeral0\xintsignedfwover }%
\def\xintsignedfwover #1%
{%
    \expandafter\XINT_sgnfwover_a\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_sgnfwover_a #1#2%
{%
    \XINT_sgnfwover_b #2\Z {#1}%
}%
\def\XINT_sgnfwover_b #1%
{%
    \xint_UDsignfork
      #1\XINT_sgnfwover_N
       -{\XINT_sgnfwover_P #1}%
    \krof
}%
\def\XINT_sgnfwover_P #1\Z #2%
{%
    \XINT_fwover_A {#2}{#1}%
}%
\def\XINT_sgnfwover_N
{%
    \expandafter\xint_minus_thenstop\romannumeral0\XINT_sgnfwover_P
}%
```

35.22 \xintREZ

```
\def\xintREZ {\romannumeral0\xintrez }%
\def\xintrez
{%
    \expandafter\XINT_rez_A\romannumeral0\XINT_infrac
}%
\def\XINT_rez_A #1#2%
{%
    \XINT_rez_AB #2\Z {#1}%
}%
\def\XINT_rez_AB #1%
```

```
{%
    \xint_UDzerominusfork
      #1-\XINT_rez_zero
      0#1\XINT_rez_neg
       0-{\XINT_rez_B #1}%
    \krof
}%
\def\XINT_rez_zero #1\Z #2#3{ 0/1[0]}%
\def\XINT_rez_neg {\expandafter\xint_minus_thenstop\romannumeral0\XINT_rez_B }%
\def\XINT_rez_B #1\Z
{%
    \expandafter\XINT_rez_C\romannumeral0\XINT_factortens {#1}%
}%
\def\XINT_rez_C #1#2#3#4%
{%
    \expandafter\XINT_rez_D\romannumeral0\XINT_factortens {#4}{#3}{#2}{#1}%
}%
\def\XINT_rez_D #1#2#3#4#5%
{%
    \expandafter\XINT_rez_E\expandafter
    {\the\numexpr #3+#4-#2}{#1}{#5}%
}%
\def\XINT_rez_E #1#2#3{ #3/#2[#1]}%
```

35.23 \xintE

1.07: The fraction is the first argument contrarily to \xintTrunc and \xintRound.

\xintfE (1.07) and \xintiE (1.09i) are for \xintexpr and cousins. It is quite annoying that \numexpr does not know how to deal correctly with a minus sign - as prefix: \numexpr -(1)\relax is illegal! (one can do \numexpr 0-(1)\relax).

the 1.07 \xintE puts directly its second argument in a \numexpr. The \xintfE first uses \xintNum on it, this is necessary for use in \xintexpr. (but one cannot use directly infix notation in the second argument of \xintfE)

1.09i also adds \xintFloatE and modifies \XINTinFloatfE, although currently the latter is only used from \xintfloatexpr hence always with \XINTdigits, it comes equipped with its first argument withing brackets as the other \XINTinFloat... macros.

```
\def\xintE {\romannumeral0\xinte }%
\def\xinte #1%
{%
    \expandafter\XINT_e \romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_e #1#2#3#4%
{%
    \expandafter\XINT_e_end\expandafter{\the\numexpr #1+#4}{#2}{#3}%
}%
\def\XINT_e_end #1#2#3{ #2/#3[#1]}%
\def\xintfE {\romannumeral0\xintfe }%
\def\xintfe #1%
```

```
{%
    \expandafter\XINT_fe \romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_fe #1#2#3#4%
{%
    \expandafter\XINT_e_end\expandafter{\the\numexpr #1+\xintNum{#4}}{#2}{#3}%
}%
\def\xintFloatE   {\romannumeral0\xintfloate }%
\def\xintfloate #1{\XINT_floate_chkopt #1\Z }%
\def\XINT_floate_chkopt #1%
{%
    \ifx [#1\expandafter\XINT_floate_opt
       \else\expandafter\XINT_floate_noopt
    \fi  #1%
}%
\def\XINT_floate_noopt #1\Z
{%
    \expandafter\XINT_floate_a\expandafter\XINTdigits
                \romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_floate_opt [\Z #1]#2%
{%
    \expandafter\XINT_floate_a\expandafter
    {\the\numexpr #1\expandafter}\romannumeral0\XINT_infrac {#2}%
}%
\def\XINT_floate_a #1#2#3#4#5%
{%
    \expandafter\expandafter\expandafter\XINT_float_a
    \expandafter\xint_exchangetwo_keepbraces\expandafter
            {\the\numexpr #2+#5}{#1}{#3}{#4}\XINT_float_Q
}%
\def\XINTinFloatfE {\romannumeral0\XINTinfloatfe }%
\def\XINTinfloatfe [#1]#2%
{%
    \expandafter\XINT_infloatfe_a\expandafter
    {\the\numexpr #1\expandafter}\romannumeral0\XINT_infrac {#2}%
}%
\def\XINT_infloatfe_a #1#2#3#4#5%
{%
    \expandafter\expandafter\expandafter\XINT_infloat_a
    \expandafter\xint_exchangetwo_keepbraces\expandafter
            {\the\numexpr #2+\xintNum{#5}}{#1}{#3}{#4}\XINT_infloat_Q
}%
\def\xintiE {\romannumeral0\xintie }% for \xintiiexpr only
\def\xintie #1%
{%
    \expandafter\XINT_ie \romannumeral0\XINT_infrac {#1}% allows 3.123e3
}%
\def\XINT_ie #1#2#3#4% assumes #3=1 and uses \xint_dsh with its \numexpr
```

```
{%
    \xint_dsh {#2}{0-(#1+#4)}% could have \xintNum{#4} for a bit more general
}%
```

35.24 \xintIrr

1.04 fixes a buggy \xintIrr {0}. 1.05 modifies the initial parsing and post-processing to use \xintrawwithzeros and to more quickly deal with an input denominator equal to 1. 1.08 version does not remove a /1 denominator.

```
\def\xintIrr {\romannumeral0\xintirr }%
\def\xintirr #1%
{%
    \expandafter\XINT_irr_start\romannumeral0\xintrawwithzeros {#1}\Z
}%
\def\XINT_irr_start #1#2/#3\Z
{%
    \if0\XINT_isOne {#3}%
      \xint_afterfi
          {\xint_UDsignfork
               #1\XINT_irr_negative
                -{\XINT_irr_nonneg #1}%
           \krof}%
    \else
      \xint_afterfi{\XINT_irr_denomisone #1}%
    \fi
    #2\Z {#3}%
}%
\def\XINT_irr_denomisone #1\Z #2{ #1/1}% changed in 1.08
\def\XINT_irr_negative   #1\Z #2{\XINT_irr_D #1\Z #2\Z \xint_minus_thenstop}%
\def\XINT_irr_nonneg     #1\Z #2{\XINT_irr_D #1\Z #2\Z \space}%
\def\XINT_irr_D #1#2\Z #3#4\Z
{%
    \xint_UDzerosfork
       #3#1\XINT_irr_indeterminate
       #30\XINT_irr_divisionbyzero
       #10\XINT_irr_zero
        00\XINT_irr_loop_a
    \krof
    {#3#4}{#1#2}{#3#4}{#1#2}%
}%
\def\XINT_irr_indeterminate #1#2#3#4#5{\xintError:NaN\space 0/0}%
\def\XINT_irr_divisionbyzero #1#2#3#4#5{\xintError:DivisionByZero #5#2/0}%
\def\XINT_irr_zero #1#2#3#4#5{ 0/1}% changed in 1.08
\def\XINT_irr_loop_a #1#2%
{%
    \expandafter\XINT_irr_loop_d
    \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
}%
```

```
\def\XINT_irr_loop_d #1#2%
{%
    \XINT_irr_loop_e #2\Z
}%
\def\XINT_irr_loop_e #1#2\Z
{%
    \xint_gob_til_zero #1\xint_irr_loop_exit0\XINT_irr_loop_a {#1#2}%
}%
\def\xint_irr_loop_exit0\XINT_irr_loop_a #1#2#3#4%
{%
    \expandafter\XINT_irr_loop_exitb\expandafter
    {\romannumeral0\xintiiquo {#3}{#2}}%
    {\romannumeral0\xintiiquo {#4}{#2}}%
}%
\def\XINT_irr_loop_exitb #1#2%
{%
   \expandafter\XINT_irr_finish\expandafter {#2}{#1}%
}%
\def\XINT_irr_finish #1#2#3{#3#1/#2}% changed in 1.08
```

35.25 \xintNum

This extension of the xint original xintNum is added in 1.05, as a synonym to \xintIrr, but raising an error when the input does not evaluate to an integer. Usable with not too much overhead on integer input as \xintIrr checks quickly for a denominator equal to 1 (which will be put there by the \XINT_infrac called by \xintrawwithzeros). This way, macros such as \xintQuo can be modified with minimal overhead to accept fractional input as long as it evaluates to an integer.

```
\def\xintNum {\romannumeral0\xintnum }%
\def\xintnum #1{\expandafter\XINT_intcheck\romannumeral0\xintirr {#1}\Z }%
\edef\XINT_intcheck #1/#2\Z
{%
    \noexpand\if 0\noexpand\XINT_isOne {#2}\noexpand\xintError:NotAnInteger
    \noexpand\fi\space #1%
}%
```

35.26 \xintifInt

1.09e. xintfrac.sty only.

```
\def\xintifInt {\romannumeral0\xintifint }%
\def\xintifint #1{\expandafter\XINT_ifint\romannumeral0\xintirr {#1}\Z }%
\def\XINT_ifint #1/#2\Z
{%
    \if\XINT_isOne {#2}1%
     \expandafter\xint_firstoftwo_thenstop
    \else
```

```
     \expandafter\xint_secondoftwo_thenstop
    \fi
}%
```

35.27 \xintJrr

Modified similarly as \xintIrr in release 1.05. 1.08 version does not remove a /1 denominator.

```
\def\xintJrr {\romannumeral0\xintjrr }%
\def\xintjrr #1%
{%
    \expandafter\XINT_jrr_start\romannumeral0\xintrawwithzeros {#1}\Z
}%
\def\XINT_jrr_start #1#2/#3\Z
{%
    \if0\XINT_isOne {#3}\xint_afterfi
          {\xint_UDsignfork
               #1\XINT_jrr_negative
                -{\XINT_jrr_nonneg #1}%
           \krof}%
    \else
      \xint_afterfi{\XINT_jrr_denomisone #1}%
    \fi
    #2\Z {#3}%
}%
\def\XINT_jrr_denomisone #1\Z #2{ #1/1}% changed in 1.08
\def\XINT_jrr_negative   #1\Z #2{\XINT_jrr_D #1\Z #2\Z \xint_minus_thenstop }%
\def\XINT_jrr_nonneg     #1\Z #2{\XINT_jrr_D #1\Z #2\Z \space}%
\def\XINT_jrr_D #1#2\Z #3#4\Z
{%
    \xint_UDzerosfork
       #3#1\XINT_jrr_indeterminate
       #30\XINT_jrr_divisionbyzero
       #10\XINT_jrr_zero
        00\XINT_jrr_loop_a
    \krof
    {#3#4}{#1#2}1001%
}%
\def\XINT_jrr_indeterminate #1#2#3#4#5#6#7{\xintError:NaN\space 0/0}%
\def\XINT_jrr_divisionbyzero #1#2#3#4#5#6#7{\xintError:DivisionByZero #7#2/0}%
\def\XINT_jrr_zero #1#2#3#4#5#6#7{ 0/1}% changed in 1.08
\def\XINT_jrr_loop_a #1#2%
{%
    \expandafter\XINT_jrr_loop_b
    \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
}%
\def\XINT_jrr_loop_b #1#2#3#4#5#6#7%
{%
```

```
    \expandafter \XINT_jrr_loop_c \expandafter
        {\romannumeral0\xintiiadd{\XINT_Mul{#4}{#1}}{#6}}%
        {\romannumeral0\xintiiadd{\XINT_Mul{#5}{#1}}{#7}}%
    {#2}{#3}{#4}{#5}%
}%
\def\XINT_jrr_loop_c #1#2%
{%
    \expandafter \XINT_jrr_loop_d \expandafter{#2}{#1}%
}%
\def\XINT_jrr_loop_d #1#2#3#4%
{%
    \XINT_jrr_loop_e #3\Z {#4}{#2}{#1}%
}%
\def\XINT_jrr_loop_e #1#2\Z
{%
    \xint_gob_til_zero #1\xint_jrr_loop_exit0\XINT_jrr_loop_a {#1#2}%
}%
\def\xint_jrr_loop_exit0\XINT_jrr_loop_a #1#2#3#4#5#6%
{%
    \XINT_irr_finish {#3}{#4}%
}%
```

35.28 \xintTFrac

1.09i, for frac in \xintexpr. And \xintFrac is already assigned. T for truncation. However, potentially not very efficient with numbers in scientific notations, with big exponents. Will have to think it again some day. I hesitated how to call the macro. Same convention as in maple, but some people reserve fractional part to x - floor(x). Also, not clear if I had to make it negative (or zero) if x < 0, or rather always positive. There should be in fact such a thing for each rounding function, trunc, round, floor, ceil.

```
\def\xintTFrac {\romannumeral0\xinttfrac }%
\def\xinttfrac #1%
    {\expandafter\XINT_tfrac_fork\romannumeral0\xintrawwithzeros {#1}\Z }%
\def\XINT_tfrac_fork #1%
{%
    \xint_UDzerominusfork
        #1-\XINT_tfrac_zero
        0#1\XINT_tfrac_N
        0-{\XINT_tfrac_P #1}%
    \krof
}%
\def\XINT_tfrac_zero #1\Z { 0/1[0]}%
\def\XINT_tfrac_N {\expandafter\XINT_opp\romannumeral0\XINT_tfrac_P }%
\def\XINT_tfrac_P #1/#2\Z
{%
    \expandafter\XINT_rez_AB\romannumeral0\xintiirem{#1}{#2}\Z {0}{#2}%
}%
```

35.29 \XINTinFloatFrac

1.09i, for frac in \xintfloatexpr. This version computes exactly from the input the fractional part and then only converts it into a float with the asked-for number of digits. I will have to think it again some day, certainly.

```
\def\XINTinFloatFrac {\romannumeral0\XINTinfloatfrac }%
\def\XINTinfloatfrac [#1]#2%
{%
    \expandafter\XINT_infloatfrac_a\expandafter
          {\romannumeral0\xinttfrac{#2}}{#1}%
}%
\def\XINT_infloatfrac_a #1#2{\XINTinFloat [#2]{#1}}%
```

35.30 \xintTrunc, \xintiTrunc

Modified in 1.06 to give the first argument to a \numexpr.

1.09f fixes the overhead added in 1.09a to some inner routines when \xintiquo was redefined to use \xintnum. Now uses \xintiiquo, rather.

1.09j: minor improvements, \XINT_trunc_E was very strange and defined two never occuring branches; also, optimizes the call to the division routine, and the zero loops.

```
\def\xintTrunc  {\romannumeral0\xinttrunc }%
\def\xintiTrunc {\romannumeral0\xintitrunc }%
\def\xinttrunc #1%
{%
    \expandafter\XINT_trunc\expandafter {\the\numexpr #1}%
}%
\def\XINT_trunc #1#2%
{%
    \expandafter\XINT_trunc_G
    \romannumeral0\expandafter\XINT_trunc_A
    \romannumeral0\XINT_infrac {#2}{#1}{#1}%
}%
\def\xintitrunc #1%
{%
    \expandafter\XINT_itrunc\expandafter {\the\numexpr #1}%
}%
\def\XINT_itrunc #1#2%
{%
    \expandafter\XINT_itrunc_G
    \romannumeral0\expandafter\XINT_trunc_A
    \romannumeral0\XINT_infrac {#2}{#1}{#1}%
}%
\def\XINT_trunc_A #1#2#3#4%
{%
    \expandafter\XINT_trunc_checkifzero
    \expandafter{\the\numexpr #1+#4}#2\Z {#3}%
```

```
}%
\def\XINT_trunc_checkifzero #1#2#3\Z
{%
    \xint_gob_til_zero #2\XINT_trunc_iszero0\XINT_trunc_B {#1}{#2#3}%
}%
\def\XINT_trunc_iszero0\XINT_trunc_B #1#2#3{ 0\Z 0}%
\def\XINT_trunc_B #1%
{%
    \ifcase\XINT_cntSgn #1\Z
      \expandafter\XINT_trunc_D
    \or
      \expandafter\XINT_trunc_D
    \else
      \expandafter\XINT_trunc_C
    \fi
    {#1}%
}%
\def\XINT_trunc_C #1#2#3%
{%
    \expandafter\XINT_trunc_CE\expandafter
    {\romannumeral0\XINT_dsx_zeroloop {-#1}{}\Z {#3}}{#2}%
}%
\def\XINT_trunc_CE #1#2{\XINT_trunc_E #2.{#1}}%
\def\XINT_trunc_D #1#2%
{%
    \expandafter\XINT_trunc_E
    \romannumeral0\XINT_dsx_zeroloop {#1}{}\Z {#2}.%
}%
\def\XINT_trunc_E #1%
{%
    \xint_UDsignfork
       #1\XINT_trunc_Fneg
        -{\XINT_trunc_Fpos #1}%
    \krof
}%
\def\XINT_trunc_Fneg #1.#2{\expandafter\xint_firstoftwo_thenstop
              \romannumeral0\XINT_div_prepare {#2}{#1}\Z \xint_minus_thenstop}%
\def\XINT_trunc_Fpos #1.#2{\expandafter\xint_firstoftwo_thenstop
              \romannumeral0\XINT_div_prepare {#2}{#1}\Z \space }%
\def\XINT_itrunc_G #1#2\Z #3#4%
{%
    \xint_gob_til_zero #1\XINT_trunc_zero 0#3#1#2%
}%
\def\XINT_trunc_zero 0#1#20{ 0}%
\def\XINT_trunc_G #1\Z #2#3%
{%
    \xint_gob_til_zero #2\XINT_trunc_zero 0%
    \expandafter\XINT_trunc_H\expandafter
    {\the\numexpr\romannumeral0\xintlength {#1}-#3}{#3}{#1}#2%
```

```
}%
\def\XINT_trunc_H #1#2%
{%
    \ifnum #1 > \xint_c_
        \xint_afterfi {\XINT_trunc_Ha {#2}}%
    \else
        \xint_afterfi {\XINT_trunc_Hb {-#1}}% -0,--1,--2, ....
    \fi
}%
\def\XINT_trunc_Ha
{%
  \expandafter\XINT_trunc_Haa\romannumeral0\xintdecsplit
}%
\def\XINT_trunc_Haa #1#2#3%
{%
    #3#1.#2%
}%
\def\XINT_trunc_Hb #1#2#3%
{%
    \expandafter #3\expandafter0\expandafter.%
    \romannumeral0\XINT_dsx_zeroloop {#1}{}\Z {}#2% #1=-0 autorisé !
}%
```

35.31 \xintRound, \xintiRound

Modified in 1.06 to give the first argument to a \numexpr.

```
\def\xintRound {\romannumeral0\xintround }%
\def\xintiRound {\romannumeral0\xintiround }%
\def\xintround #1%
{%
    \expandafter\XINT_round\expandafter {\the\numexpr #1}%
}%
\def\XINT_round
{%
    \expandafter\XINT_trunc_G\romannumeral0\XINT_round_A
}%
\def\xintiround #1%
{%
    \expandafter\XINT_iround\expandafter {\the\numexpr #1}%
}%
\def\XINT_iround
{%
    \expandafter\XINT_itrunc_G\romannumeral0\XINT_round_A
}%
\def\XINT_round_A #1#2%
{%
    \expandafter\XINT_round_B
    \romannumeral0\expandafter\XINT_trunc_A
```

```
    \romannumeral0\XINT_infrac {#2}{\the\numexpr #1+1\relax}{#1}%
}%
\def\XINT_round_B #1\Z
{%
    \expandafter\XINT_round_C
    \romannumeral0\XINT_rord_main {}#1%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax
    \Z
}%
\def\XINT_round_C #1%
{%
    \ifnum #1<5
        \expandafter\XINT_round_Daa
    \else
        \expandafter\XINT_round_Dba
    \fi
}%
\def\XINT_round_Daa #1%
{%
    \xint_gob_til_Z #1\XINT_round_Daz\Z \XINT_round_Da #1%
}%
\def\XINT_round_Daz\Z \XINT_round_Da \Z { 0\Z }%
\def\XINT_round_Da #1\Z
{%
    \XINT_rord_main {}#1%
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax  \Z
}%
\def\XINT_round_Dba #1%
{%
    \xint_gob_til_Z #1\XINT_round_Dbz\Z \XINT_round_Db #1%
}%
\def\XINT_round_Dbz\Z \XINT_round_Db \Z { 1\Z }%
\def\XINT_round_Db #1\Z
{%
    \XINT_addm_A 0{}1000\W\X\Y\Z #1000\W\X\Y\Z \Z
}%
```

35.32 \xintXTrunc

1.09j [2014/01/06] This is completely expandable but not f-expandable. Designed be used inside an \edef or a \write, if one is interested in getting tens of thousands of digits from the decimal expansion of some fraction... it is not worth using it rather than \xintTrunc if for less than *hundreds* of digits. For efficiency it clones part of the preparatory division macros, as the same denominator will be used again and again. The D parameter which says how many digits to keep after decimal mark must be at least 1 (and it is forcefully set to such a value if found negative or zero, to avoid an eternal loop).

For reasons of efficiency I try to use the shortest possible denominator, so if the fraction is A/B[N], I want to use B. For N at least zero, just immediately replace A by A.10^N. The first division then may be a little longish but the next ones will be fast (if B is not too big). For N<0, this is a bit more complicated. I thought somewhat about this, and I would need a rather complicated approach going through a long division algorithm, forcing me to essentially clone the actual division with some differences; a side thing is that as this would use blocks of four digits I would have a hard time allowing a non-multiple of four number of post decimal mark digits.

Thus, for N<0, another method is followed. First the euclidean division A/B=Q+R/B is done. The number of digits of Q is M. If

N\leq D, we launch % inside a \csname the routine for obtaining D-N next digits (this may impact % TeX's memory if D is very big), call them T. We then need to position the % decimal mark D slots from the right of QT, which has length M+D-N, hence N % slots from the right of Q. We thus avoid having to work will the T, as D may % be very very big (\xintXTrunc's only goal is to make it possible to learn by % hearts decimal expansions with thousands of digits). We can use the % \xintDecSplit for that on Q . Computing the length M of Q was a more or less % unavoidable step. If N>D, the \csname step is skipped we need to remove the % D-N last digits from Q, etc.. we compare D-N with the length M of Q etc... % (well in this last, very uncommon, branch, I stopped trying to optimize thinsg % and I even do an \xintnum to ensure a 0 if something comes out empty from % \xintDecSplit).

```
\def\xintXTrunc #1#2%
{%
    \expandafter\XINT_xtrunc_a\expandafter
    {\the\numexpr #1\expandafter}\romannumeral0\xintraw {#2}%
}%
\def\XINT_xtrunc_a #1%
{%
    \expandafter\XINT_xtrunc_b\expandafter
    {\the\numexpr\ifnum#1<\xint_c_i \xint_c_i-\fi #1}%
}%
\def\XINT_xtrunc_b #1%
{%
    \expandafter\XINT_xtrunc_c\expandafter
    {\the\numexpr (#1+\xint_c_ii^v)/\xint_c_ii^vi-\xint_c_i}{#1}%
}%
\def\XINT_xtrunc_c #1#2%
{%
    \expandafter\XINT_xtrunc_d\expandafter
    {\the\numexpr #2-\xint_c_ii^vi*#1}{#1}{#2}%
}%
\def\XINT_xtrunc_d #1#2#3#4/#5[#6]%
```

```
{%
    \XINT_xtrunc_e #4.{#6}{#5}{#3}{#2}{#1}%
}%
% #1=numerator.#2=N,#3=B,#4=D,#5=Blocs,#6=extra
\def\XINT_xtrunc_e #1%
{%
    \xint_UDzerominusfork
        #1-\XINT_xtrunc_zero
        0#1\XINT_xtrunc_N
        0-{\XINT_xtrunc_P #1}%
    \krof
}%
\def\XINT_xtrunc_zero .#1#2#3#4#5%
{%
    0.\romannumeral0\expandafter\XINT_dsx_zeroloop\expandafter
                              {\the\numexpr #5}{}\Z {}%
    \xintiloop [#4+-1]
    \ifnum \xintiloopindex>\xint_c_
    0000000000000000000000000000000000000000000000000000000000000000%
    \repeat
}%
\def\XINT_xtrunc_N {-\XINT_xtrunc_P }%
\def\XINT_xtrunc_P #1.#2%
{%
    \ifnum #2<\xint_c_
        \expandafter\XINT_xtrunc_negN_Q
    \else
        \expandafter\XINT_xtrunc_Q
    \fi  {#2}{#1}.%
}%
\def\XINT_xtrunc_negN_Q #1#2.#3#4#5#6%
{%
    \expandafter\XINT_xtrunc_negN_R
    \romannumeral0\XINT_div_prepare {#3}{#2}{#3}{#1}{#4}%
}%
% #1=Q, #2=R, #3=B, #4=N<0, #5=D
\def\XINT_xtrunc_negN_R #1#2#3#4#5%
{%
    \expandafter\XINT_xtrunc_negN_S\expandafter
    {\the\numexpr -#4}{#5}{#2}{#3}{#1}%
}%
\def\XINT_xtrunc_negN_S #1#2%
{%
    \expandafter\XINT_xtrunc_negN_T\expandafter
    {\the\numexpr #2-#1}{#1}{#2}%
}%
\def\XINT_xtrunc_negN_T #1%
{%
    \ifnum \xint_c_<#1
```

```
      \expandafter\XINT_xtrunc_negNA
    \else
      \expandafter\XINT_xtrunc_negNW
    \fi {#1}%
}%
% #1=D-|N|>0, #2=|N|, #3=D, #4=R, #5=B, #6=Q
\def\XINT_xtrunc_unlock #10.{ }%
\def\XINT_xtrunc_negNA #1#2#3#4#5#6%
{%
   \expandafter\XINT_xtrunc_negNB\expandafter
   {\romannumeral0\expandafter\expandafter\expandafter
    \XINT_xtrunc_unlock\expandafter\string
    \csname\XINT_xtrunc_b {#1}#4/#5[0]\expandafter\endcsname
    \expandafter}\expandafter
    {\the\numexpr\xintLength{#6}-#2}{#6}%
}%
\def\XINT_xtrunc_negNB #1#2#3{\XINT_xtrunc_negNC {#2}{#3}#1}%
\def\XINT_xtrunc_negNC #1%
{%
    \ifnum \xint_c_ < #1
      \expandafter\XINT_xtrunc_negNDa
    \else
      \expandafter\XINT_xtrunc_negNE
    \fi {#1}%
}%
\def\XINT_xtrunc_negNDa #1#2%
{%
    \expandafter\XINT_xtrunc_negNDb%
    \romannumeral0\XINT_split_fromleft_loop {#1}{}#2\W\W\W\W\W\W\W\W\Z
}%
\def\XINT_xtrunc_negNDb #1#2{#1.#2}%
\def\XINT_xtrunc_negNE #1#2%
{%
    0.\romannumeral0\XINT_dsx_zeroloop {-#1}{}\Z {}#2%
}%
% #1=D-|N|<=0, #2=|N|, #3=D, #4=R, #5=B, #6=Q
\def\XINT_xtrunc_negNW #1#2#3#4#5#6%
{%
    \expandafter\XINT_xtrunc_negNX\expandafter
    {\romannumeral0\xintnum{\xintDecSplitL {-#1}{#6}}}{#3}%
}%
\def\XINT_xtrunc_negNX #1#2%
{%
    \expandafter\XINT_xtrunc_negNC\expandafter
    {\the\numexpr\xintLength {#1}-#2}{#1}%
}%
\def\XINT_xtrunc_Q #1%
{%
    \expandafter\XINT_xtrunc_prepare_I
```

```
    \romannumeral0\XINT_dsx_zeroloop {#1}{}\Z
}%
\def\XINT_xtrunc_prepare_I #1.#2#3%
{%
    \expandafter\XINT_xtrunc_prepareB_aa\expandafter
    {\romannumeral0\xintlength {#2}}{#2}{#1}%
}%
\def\XINT_xtrunc_prepareB_aa #1%
{%
    \ifnum #1=\xint_c_i
      \expandafter\XINT_xtrunc_prepareB_onedigit
    \else
      \expandafter\XINT_xtrunc_prepareB_PaBa
    \fi
    {#1}%
}%
\def\XINT_xtrunc_prepareB_onedigit #1#2%
{%
    \ifcase#2
    \or\expandafter\XINT_xtrunc_BisOne
    \or\expandafter\XINT_xtrunc_BisTwo
    \else\expandafter\XINT_xtrunc_prepareB_PaBe
    \fi {000}{0}{4}{#2}%
}%
\def\XINT_xtrunc_BisOne #1#2#3#4#5#6#7%
{%
    #5.\romannumeral0\expandafter\XINT_dsx_zeroloop\expandafter
                              {\the\numexpr #7}{}\Z {}%
    \xintiloop [#6+-1]
    \ifnum \xintiloopindex>\xint_c_
    0000000000000000000000000000000000000000000000000000000000000000%
    \repeat
}%
\def\XINT_xtrunc_BisTwo #1#2#3#4#5#6#7%
{%
    \xintHalf {#5}.\ifodd\xintiiLDg{#5} 5\else 0\fi
    \romannumeral0\expandafter\XINT_dsx_zeroloop\expandafter
                              {\the\numexpr #7-\xint_c_i}{}\Z {}%
    \xintiloop [#6+-1]
    \ifnum \xintiloopindex>\xint_c_
    0000000000000000000000000000000000000000000000000000000000000000%
    \repeat
}%
\def\XINT_xtrunc_prepareB_PaBa #1#2%
{%
    \expandafter\XINT_xtrunc_Pa\expandafter
    {\romannumeral0\XINT_xtrunc_prepareB_a {#1}{#2}}%
}%
\def\XINT_xtrunc_prepareB_a #1%
```

```
{%
  \expandafter\XINT_xtrunc_prepareB_c\expandafter
  {\the\numexpr \xint_c_iv*((#1+\xint_c_i)/\xint_c_iv)}{#1}%
}%
\def\XINT_xtrunc_prepareB_c #1#2%
{%
    \csname XINT_xtrunc_prepareB_d\romannumeral\numexpr#1-#2\endcsname
    {#1}%
}%
\def\XINT_xtrunc_prepareB_d    {\XINT_xtrunc_prepareB_e {}{0000}}%
\def\XINT_xtrunc_prepareB_di   {\XINT_xtrunc_prepareB_e {0}{000}}%
\def\XINT_xtrunc_prepareB_dii  {\XINT_xtrunc_prepareB_e {00}{00}}%
\def\XINT_xtrunc_prepareB_diii {\XINT_xtrunc_prepareB_e {000}{0}}%
\def\XINT_xtrunc_prepareB_PaBe #1#2#3#4%
{%
    \expandafter\XINT_xtrunc_Pa\expandafter
    {\romannumeral0\XINT_xtrunc_prepareB_e {#1}{#2}{#3}{#4}}%
}%
\def\XINT_xtrunc_prepareB_e #1#2#3#4%
{%
    \ifnum#3=\xint_c_iv\expandafter\XINT_xtrunc_prepareLittleB_f
                   \else\expandafter\XINT_xtrunc_prepareB_f
    \fi
    #4#1{#3}{#2}{#1}%
}%
\def\XINT_xtrunc_prepareB_f #1#2#3#4#5#{%
    \expandafter\space
    \expandafter\XINT_div_prepareB_g
     \the\numexpr  #1#2#3#4+\xint_c_i\expandafter
    .\the\numexpr (#1#2#3#4+\xint_c_i)/\xint_c_ii\expandafter
    .\romannumeral0\xintreverseorder {#1#2#3#4#5}.{#1#2#3#4}%
}%
\def\XINT_xtrunc_prepareLittleB_f #1#{%
    \expandafter\space\expandafter
    \XINT_div_prepareB_g \the\numexpr  #1/\xint_c_ii.{}.{}.{#1}%
}%
\def\XINT_xtrunc_Pa #1#2%
{%
    \expandafter\XINT_xtrunc_Pb\romannumeral0#1{#2}{#1}%
}%
\def\XINT_xtrunc_Pb #1#2#3#4{#1.\XINT_xtrunc_A {#4}{#2}{#3}}%
\def\XINT_xtrunc_A #1%
{%
    \unless\ifnum #1>\xint_c_ \XINT_xtrunc_transition\fi
    \expandafter\XINT_xtrunc_B\expandafter{\the\numexpr #1-\xint_c_i}%
}%
\def\XINT_xtrunc_B #1#2#3%
{%
    \expandafter\XINT_xtrunc_D\romannumeral0#3%
```

```
    {#20000000000000000000000000000000000000000000000000000000000000000}%
    {#1}{#3}%
}%
\def\XINT_xtrunc_D #1#2#3%
{%
    \romannumeral0\expandafter\XINT_dsx_zeroloop\expandafter
                  {\the\numexpr \xint_c_ii^vi-\xintLength{#1}}{}\Z {}#1%
    \XINT_xtrunc_A {#3}{#2}%
}%
\def\XINT_xtrunc_transition\fi
    \expandafter\XINT_xtrunc_B\expandafter #1#2#3#4%
{%
    \fi
    \ifnum #4=\xint_c_ \XINT_xtrunc_abort\fi
    \expandafter\XINT_xtrunc_x\expandafter
    {\romannumeral0\XINT_dsx_zeroloop {#4}{}\Z {#2}}{#3}{#4}%
}%
\def\XINT_xtrunc_x #1#2%
{%
    \expandafter\XINT_xtrunc_y\romannumeral0#2{#1}%
}%
\def\XINT_xtrunc_y #1#2#3%
{%
    \romannumeral0\expandafter\XINT_dsx_zeroloop\expandafter
                  {\the\numexpr #3-\xintLength{#1}}{}\Z {}#1%
}%
\def\XINT_xtrunc_abort\fi\expandafter\XINT_xtrunc_x\expandafter #1#2#3{\fi}%
```

35.33 \xintDigits

The mathchardef used to be called \XINT_digits, but for reasons originating in \xintNewExpr, release 1.09a uses \XINTdigits without underscore.

```
\mathchardef\XINTdigits 16
\def\xintDigits #1#2%
   {\afterassignment \xint_gobble_i \mathchardef\XINTdigits=}%
\def\xinttheDigits {\number\XINTdigits }%
```

35.34 \xintFloat

1.07. Completely re-written in 1.08a, with spectacular speed gains. The earlier version was seriously silly when dealing with inputs having a big power of ten. Again some modifications in 1.08b for a better treatment of cases with long explicit numerators or denominators.

Here again some inner macros used the \xintiquo with extra \xintnum overhead in 1.09a, 1.09f reinstalled use of \xintiiquo without this overhead.

```
\def\xintFloat    {\romannumeral0\xintfloat }%
```

```
\def\xintfloat #1{\XINT_float_chkopt #1\Z }%
\def\XINT_float_chkopt #1%
{%
    \ifx [#1\expandafter\XINT_float_opt
       \else\expandafter\XINT_float_noopt
    \fi  #1%
}%
\def\XINT_float_noopt #1\Z
{%
    \expandafter\XINT_float_a\expandafter\XINTdigits
    \romannumeral0\XINT_infrac {#1}\XINT_float_Q
}%
\def\XINT_float_opt [\Z #1]#2%
{%
    \expandafter\XINT_float_a\expandafter
    {\the\numexpr #1\expandafter}%
    \romannumeral0\XINT_infrac {#2}\XINT_float_Q
}%
\def\XINT_float_a #1#2#3% #1=P, #2=n, #3=A, #4=B
{%
    \XINT_float_fork #3\Z {#1}{#2}% #1 = precision, #2=n
}%
\def\XINT_float_fork #1%
{%
    \xint_UDzerominusfork
     #1-\XINT_float_zero
     0#1\XINT_float_J
      0-{\XINT_float_K #1}%
    \krof
}%
\def\XINT_float_zero #1\Z #2#3#4#5{ 0.e0}%
\def\XINT_float_J {\expandafter\xint_minus_thenstop\romannumeral0\XINT_float_K }%
\def\XINT_float_K #1\Z #2% #1=A, #2=P, #3=n, #4=B
{%
    \expandafter\XINT_float_L\expandafter
    {\the\numexpr\xintLength{#1}\expandafter}\expandafter
    {\the\numexpr #2+\xint_c_ii}{#1}{#2}%
}%
\def\XINT_float_L #1#2%
{%
    \ifnum #1>#2
      \expandafter\XINT_float_Ma
    \else
      \expandafter\XINT_float_Mc
    \fi {#1}{#2}%
}%
\def\XINT_float_Ma #1#2#3%
{%
    \expandafter\XINT_float_Mb\expandafter
```

```
    {\the\numexpr #1-#2\expandafter\expandafter\expandafter}%
    \expandafter\expandafter\expandafter
    {\expandafter\xint_firstoftwo
     \romannumeral0\XINT_split_fromleft_loop {#2}{}#3\W\W\W\W\W\W\W\W\W\Z
     }{#2}%
}%
\def\XINT_float_Mb #1#2#3#4#5#6% #2=A', #3=P+2, #4=P, #5=n, #6=B
{%
   \expandafter\XINT_float_N\expandafter
   {\the\numexpr\xintLength{#6}\expandafter}\expandafter
   {\the\numexpr #3\expandafter}\expandafter
   {\the\numexpr #1+#5}%
   {#6}{#3}{#2}{#4}%
}% long de B, P+2, n', B, |A'|=P+2, A', P
\def\XINT_float_Mc #1#2#3#4#5#6%
{%
   \expandafter\XINT_float_N\expandafter
   {\romannumeral0\xintlength{#6}}{#2}{#5}{#6}{#1}{#3}{#4}%
}% long de B, P+2, n, B, |A|, A, P
\def\XINT_float_N #1#2%
{%
    \ifnum #1>#2
      \expandafter\XINT_float_O
    \else
      \expandafter\XINT_float_P
    \fi {#1}{#2}%
}%
\def\XINT_float_O #1#2#3#4%
{%
    \expandafter\XINT_float_P\expandafter
    {\the\numexpr #2\expandafter}\expandafter
    {\the\numexpr #2\expandafter}\expandafter
    {\the\numexpr #3-#1+#2\expandafter\expandafter\expandafter}%
    \expandafter\expandafter\expandafter
    {\expandafter\xint_firstoftwo
     \romannumeral0\XINT_split_fromleft_loop {#2}{}#4\W\W\W\W\W\W\W\W\W\Z
     }%
}% |B|,P+2,n,B,|A|,A,P
\def\XINT_float_P #1#2#3#4#5#6#7#8%
{%
    \expandafter #8\expandafter {\the\numexpr #1-#5+#2-\xint_c_i}%
    {#6}{#4}{#7}{#3}%
}% |B|-|A|+P+1,A,B,P,n
\def\XINT_float_Q #1%
{%
    \ifnum #1<\xint_c_
      \expandafter\XINT_float_Ri
    \else
      \expandafter\XINT_float_Rii
```

```
    \fi {#1}%
}%
\def\XINT_float_Ri #1#2#3%
{%
    \expandafter\XINT_float_Sa
    \romannumeral0\xintiiquo {#2}%
         {\XINT_dsx_addzerosnofuss {-#1}{#3}}\Z {#1}%
}%
\def\XINT_float_Rii #1#2#3%
{%
    \expandafter\XINT_float_Sa
    \romannumeral0\xintiiquo
         {\XINT_dsx_addzerosnofuss {#1}{#2}}{#3}\Z {#1}%
}%
\def\XINT_float_Sa #1%
{%
    \if #19%
        \xint_afterfi {\XINT_float_Sb\XINT_float_Wb }%
    \else
        \xint_afterfi {\XINT_float_Sb\XINT_float_Wa }%
    \fi #1%
}%
\def\XINT_float_Sb #1#2\Z #3#4%
{%
    \expandafter\XINT_float_T\expandafter
    {\the\numexpr #4+\xint_c_i\expandafter}%
    \romannumeral-`0\XINT_lenrord_loop 0{}#2\Z\W\W\W\W\W\W\W\Z #1{#3}{#4}%
}%
\def\XINT_float_T #1#2#3%
{%
    \ifnum #2>#1
      \xint_afterfi{\XINT_float_U\XINT_float_Xb}%
    \else
      \xint_afterfi{\XINT_float_U\XINT_float_Xa #3}%
    \fi
}%
\def\XINT_float_U #1#2%
{%
    \ifnum #2<\xint_c_v
      \expandafter\XINT_float_Va
    \else
      \expandafter\XINT_float_Vb
    \fi #1%
}%
\def\XINT_float_Va #1#2\Z #3%
{%
    \expandafter#1%
    \romannumeral0\expandafter\XINT_float_Wa
    \romannumeral0\XINT_rord_main {}#2%
```

```
      \xint_relax
        \xint_bye\xint_bye\xint_bye\xint_bye
        \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_relax \Z
}%
\def\XINT_float_Vb #1#2\Z #3%
{%
    \expandafter #1%
    \romannumeral0\expandafter #3%
    \romannumeral0\XINT_addm_A 0{}1000\W\X\Y\Z #2000\W\X\Y\Z \Z
}%
\def\XINT_float_Wa #1{ #1.}%
\def\XINT_float_Wb #1#2%
    {\if #11\xint_afterfi{ 10.}\else\xint_afterfi{ #1.#2}\fi }%
\def\XINT_float_Xa #1\Z #2#3#4%
{%
    \expandafter\XINT_float_Y\expandafter
    {\the\numexpr #3+#4-#2}{#1}%
}%
\def\XINT_float_Xb #1\Z #2#3#4%
{%
    \expandafter\XINT_float_Y\expandafter
    {\the\numexpr #3+#4+\xint_c_i-#2}{#1}%
}%
\def\XINT_float_Y #1#2{ #2e#1}%
```

35.35 \XINTinFloat

1.07. Completely rewritten in 1.08a for immensely greater efficiency when the power of ten is big: previous version had some very serious bottlenecks arising from the creation of long strings of zeros, which made things such as 2^999999 completely impossible, but now even 2^999999999 with 24 significant digits is no problem! Again (slightly) improved in 1.08b.

I decide in 1.09a not to use anymore \romannumeral`-0 mais \romannumeral0 also in the float routines, for consistency of style.

Here again some inner macros used the \xintiquo with extra \xintnum overhead in 1.09a, 1.09f fixed that to use \xintiiquo for example.

1.09i added a stupid bug to \XINT_infloat_zero when it changed 0[0] to a silly 0/1[0], breaking in particular \xintFloatAdd when one of the argument is zero :(((

1.09j fixes this. Besides, for notational coherence \XINT_inFloat and \XINT_infloat have been renamed respectively \XINTinFloat and \XINTinfloat in release 1.09j.

```
\def\XINTinFloat {\romannumeral0\XINTinfloat }%
\def\XINTinfloat [#1]#2%
{%
    \expandafter\XINT_infloat_a\expandafter
    {\the\numexpr #1\expandafter}%
    \romannumeral0\XINT_infrac {#2}\XINT_infloat_Q
```

```
}%
\def\XINT_infloat_a #1#2#3% #1=P, #2=n, #3=A, #4=B
{%
    \XINT_infloat_fork #3\Z {#1}{#2}% #1 = precision, #2=n
}%
\def\XINT_infloat_fork #1%
{%
    \xint_UDzerominusfork
     #1-\XINT_infloat_zero
     0#1\XINT_infloat_J
      0-{\XINT_float_K #1}%
    \krof
}%
\def\XINT_infloat_zero #1\Z #2#3#4#5{ 0[0]}%
% the 0[0] was stupidly changed to 0/1[0] in 1.09i, with the result that the
% Float addition would crash when an operand was zero
\def\XINT_infloat_J {\expandafter-\romannumeral0\XINT_float_K }%
\def\XINT_infloat_Q #1%
{%
    \ifnum #1<\xint_c_
      \expandafter\XINT_infloat_Ri
    \else
      \expandafter\XINT_infloat_Rii
    \fi {#1}%
}%
\def\XINT_infloat_Ri #1#2#3%
{%
    \expandafter\XINT_infloat_S\expandafter
    {\romannumeral0\xintiiquo {#2}%
         {\XINT_dsx_addzerosnofuss {-#1}{#3}}}{#1}%
}%
\def\XINT_infloat_Rii #1#2#3%
{%
    \expandafter\XINT_infloat_S\expandafter
    {\romannumeral0\xintiiquo
         {\XINT_dsx_addzerosnofuss {#1}{#2}}{#3}}{#1}%
}%
\def\XINT_infloat_S #1#2#3%
{%
    \expandafter\XINT_infloat_T\expandafter
    {\the\numexpr #3+\xint_c_i\expandafter}%
    \romannumeral-`0\XINT_lenrord_loop 0{}#1\Z\W\W\W\W\W\W\W\Z
    {#2}%
}%
\def\XINT_infloat_T #1#2#3%
{%
    \ifnum #2>#1
      \xint_afterfi{\XINT_infloat_U\XINT_infloat_Wb}%
    \else
```

```
    \xint_afterfi{\XINT_infloat_U\XINT_infloat_Wa #3}%
  \fi
}%
\def\XINT_infloat_U #1#2%
{%
  \ifnum #2<\xint_c_v
    \expandafter\XINT_infloat_Va
  \else
    \expandafter\XINT_infloat_Vb
  \fi #1%
}%
\def\XINT_infloat_Va #1#2\Z
{%
  \expandafter#1%
  \romannumeral0\XINT_rord_main {}#2%
    \xint_relax
      \xint_bye\xint_bye\xint_bye\xint_bye
      \xint_bye\xint_bye\xint_bye\xint_bye
    \xint_relax \Z
}%
\def\XINT_infloat_Vb #1#2\Z
{%
  \expandafter #1%
  \romannumeral0\XINT_addm_A 0{}1000\W\X\Y\Z #2000\W\X\Y\Z \Z
}%
\def\XINT_infloat_Wa #1\Z #2#3%
{%
  \expandafter\XINT_infloat_X\expandafter
  {\the\numexpr #3+\xint_c_i-#2}{#1}%
}%
\def\XINT_infloat_Wb #1\Z #2#3%
{%
  \expandafter\XINT_infloat_X\expandafter
  {\the\numexpr #3+\xint_c_ii-#2}{#1}%
}%
\def\XINT_infloat_X #1#2{ #2[#1]}%
```

35.36 \xintAdd

```
\def\xintAdd {\romannumeral0\xintadd }%
\def\xintadd #1%
{%
  \expandafter\xint_fadd\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fadd #1#2{\expandafter\XINT_fadd_A\romannumeral0\XINT_infrac{#2}#1}%
\def\XINT_fadd_A #1#2#3#4%
{%
  \ifnum #4 > #1
     \xint_afterfi {\XINT_fadd_B {#1}}%
```

```
    \else
       \xint_afterfi {\XINT_fadd_B {#4}}%
    \fi
    {#1}{#4}{#2}{#3}%
}%
\def\XINT_fadd_B #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_fadd_C\expandafter
    {\romannumeral0\xintiimul {#7}{#5}}%
    {\romannumeral0\xintiiadd
    {\romannumeral0\xintiimul {\xintDSH {\the\numexpr -#3+#1\relax}{#6}}{#5}}%
    {\romannumeral0\xintiimul {#7}{\xintDSH {\the\numexpr -#2+#1\relax}{#4}}}%
    }%
    {#1}%
}%
\def\XINT_fadd_C #1#2#3%
{%
    \expandafter\XINT_fadd_D\expandafter {#2}{#3}{#1}%
}%
\def\XINT_fadd_D #1#2{\XINT_outfrac {#2}{#1}}%
```

35.37 \xintSub

```
\def\xintSub {\romannumeral0\xintsub }%
\def\xintsub #1%
{%
    \expandafter\xint_fsub\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fsub #1#2%
   {\expandafter\XINT_fsub_A\romannumeral0\XINT_infrac {#2}#1}%
\def\XINT_fsub_A #1#2#3#4%
{%
    \ifnum #4 > #1
       \xint_afterfi {\XINT_fsub_B {#1}}%
    \else
       \xint_afterfi {\XINT_fsub_B {#4}}%
    \fi
    {#1}{#4}{#2}{#3}%
}%
\def\XINT_fsub_B #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_fsub_C\expandafter
    {\romannumeral0\xintiimul {#7}{#5}}%
    {\romannumeral0\xintiisub
    {\romannumeral0\xintiimul {\xintDSH {\the\numexpr -#3+#1\relax}{#6}}{#5}}%
    {\romannumeral0\xintiimul {#7}{\xintDSH {\the\numexpr -#2+#1\relax}{#4}}}%
    }%
    {#1}%
}%
\def\XINT_fsub_C #1#2#3%
```

```
{%
    \expandafter\XINT_fsub_D\expandafter {#2}{#3}{#1}%
}%
\def\XINT_fsub_D #1#2{\XINT_outfrac {#2}{#1}}%
```

35.38 \xintSum

```
\def\xintSum {\romannumeral0\xintsum }%
\def\xintsum #1{\xintsumexpr #1\relax }%
\def\xintSumExpr {\romannumeral0\xintsumexpr }%
\def\xintsumexpr {\expandafter\XINT_fsumexpr\romannumeral-`0}%
\def\XINT_fsumexpr {\XINT_fsum_loop_a {0/1[0]}}%
\def\XINT_fsum_loop_a #1#2%
{%
    \expandafter\XINT_fsum_loop_b \romannumeral-`0#2\Z {#1}%
}%
\def\XINT_fsum_loop_b #1%
{%
    \xint_gob_til_relax #1\XINT_fsum_finished\relax
    \XINT_fsum_loop_c #1%
}%
\def\XINT_fsum_loop_c #1\Z #2%
{%
    \expandafter\XINT_fsum_loop_a\expandafter{\romannumeral0\xintadd {#2}{#1}}%
}%
\def\XINT_fsum_finished #1\Z #2{ #2}%
```

35.39 \xintMul

```
\def\xintMul {\romannumeral0\xintmul }%
\def\xintmul #1%
{%
    \expandafter\xint_fmul\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fmul #1#2%
   {\expandafter\XINT_fmul_A\romannumeral0\XINT_infrac {#2}#1}%
\def\XINT_fmul_A #1#2#3#4#5#6%
{%
    \expandafter\XINT_fmul_B
    \expandafter{\the\numexpr #1+#4\expandafter}%
    \expandafter{\romannumeral0\xintiimul {#6}{#3}}%
    {\romannumeral0\xintiimul {#5}{#2}}%
}%
\def\XINT_fmul_B #1#2#3%
{%
    \expandafter \XINT_fmul_C \expandafter{#3}{#1}{#2}%
}%
\def\XINT_fmul_C #1#2{\XINT_outfrac {#2}{#1}}%
```

35.40 \xintSqr

```
\def\xintSqr {\romannumeral0\xintsqr }%
\def\xintsqr #1%
{%
    \expandafter\xint_fsqr\expandafter{\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fsqr #1{\XINT_fmul_A #1#1}%
```

35.41 \xintPow

Modified in 1.06 to give the exponent to a \numexpr.

With 1.07 and for use within the \xintexpr parser, we must allow fractions (which are integers in disguise) as input to the exponent, so we must have a variant which uses \xintNum and not only \numexpr for normalizing the input. Hence the \xintfPow here.

1.08b: well actually I think that with xintfrac.sty loaded the exponent should always be allowed to be a fraction giving an integer. So I do as for \xintFac, and remove here the duplicated. Then \xintexpr can use the \xintPow as defined here.

```
\def\xintPow {\romannumeral0\xintpow }%
\def\xintpow #1%
{%
    \expandafter\xint_fpow\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fpow #1#2%
{%
    \expandafter\XINT_fpow_fork\the\numexpr \xintNum{#2}\relax\Z #1%
}%
\def\XINT_fpow_fork #1#2\Z
{%
    \xint_UDzerominusfork
      #1-\XINT_fpow_zero
      0#1\XINT_fpow_neg
       0-{\XINT_fpow_pos #1}%
    \krof
    {#2}%
}%
\def\XINT_fpow_zero #1#2#3#4{ 1/1[0]}%
\def\XINT_fpow_pos #1#2#3#4#5%
{%
    \expandafter\XINT_fpow_pos_A\expandafter
    {\the\numexpr #1#2*#3\expandafter}\expandafter
    {\romannumeral0\xintiipow {#5}{#1#2}}%
    {\romannumeral0\xintiipow {#4}{#1#2}}%
}%
\def\XINT_fpow_neg #1#2#3#4%
{%
    \expandafter\XINT_fpow_pos_A\expandafter
    {\the\numexpr -#1*#2\expandafter}\expandafter
    {\romannumeral0\xintiipow {#3}{#1}}%
    {\romannumeral0\xintiipow {#4}{#1}}%
```

```
}%
\def\XINT_fpow_pos_A #1#2#3%
{%
    \expandafter\XINT_fpow_pos_B\expandafter {#3}{#1}{#2}%
}%
\def\XINT_fpow_pos_B #1#2{\XINT_outfrac {#2}{#1}}%
```

35.42 \xintFac

1.07: to be used by the \xintexpr scanner which needs to be able to apply \xintFac to a fraction which is an integer in disguise; so we use \xintNum and not only \numexpr. Je modifie cela dans 1.08b, au lieu d'avoir un \xintfFac spécialement pour \xintexpr, tout simplement j'étends \xintFac comme les autres macros, pour qu'elle utilise \xintNum.

```
\def\xintFac {\romannumeral0\xintfac }%
\def\xintfac #1%
{%
    \expandafter\XINT_fac_fork\expandafter{\the\numexpr \xintNum{#1}}%
}%
```

35.43 \xintPrd

```
\def\xintPrd {\romannumeral0\xintprd }%
\def\xintprd #1{\xintprdexpr #1\relax }%
\def\xintPrdExpr {\romannumeral0\xintprdexpr }%
\def\xintprdexpr {\expandafter\XINT_fprdexpr \romannumeral-`0}%
\def\XINT_fprdexpr {\XINT_fprod_loop_a {1/1[0]}}%
\def\XINT_fprod_loop_a #1#2%
{%
    \expandafter\XINT_fprod_loop_b \romannumeral-`0#2\Z {#1}%
}%
\def\XINT_fprod_loop_b #1%
{%
    \xint_gob_til_relax #1\XINT_fprod_finished\relax
    \XINT_fprod_loop_c #1%
}%
\def\XINT_fprod_loop_c #1\Z #2%
{%
  \expandafter\XINT_fprod_loop_a\expandafter{\romannumeral0\xintmul {#1}{#2}}%
}%
\def\XINT_fprod_finished #1\Z #2{ #2}%
```

35.44 \xintDiv

```
\def\xintDiv {\romannumeral0\xintdiv }%
\def\xintdiv #1%
{%
    \expandafter\xint_fdiv\expandafter {\romannumeral0\XINT_infrac {#1}}%
```

```
}%
\def\xint_fdiv #1#2%
   {\expandafter\XINT_fdiv_A\romannumeral0\XINT_infrac {#2}#1}%
\def\XINT_fdiv_A #1#2#3#4#5#6%
{%
    \expandafter\XINT_fdiv_B
    \expandafter{\the\numexpr #4-#1\expandafter}%
    \expandafter{\romannumeral0\xintiimul {#2}{#6}}%
    {\romannumeral0\xintiimul {#3}{#5}}%
}%
\def\XINT_fdiv_B #1#2#3%
{%
    \expandafter\XINT_fdiv_C
    \expandafter{#3}{#1}{#2}%
}%
\def\XINT_fdiv_C #1#2{\XINT_outfrac {#2}{#1}}%
```

35.45 \xintIsOne

New with 1.09a. Could be more efficient. For fractions with big powers of tens, it is better to use \xintCmp{f}{1}. Restyled in 1.09i.

```
\def\xintIsOne {\romannumeral0\xintisone }%
\def\xintisone #1{\expandafter\XINT_fracisone
                  \romannumeral0\xintrawwithzeros{#1}\Z }%
\def\XINT_fracisone #1/#2\Z
    {\if0\XINT_Cmp {#1}{#2}\xint_afterfi{ 1}\else\xint_afterfi{ 0}\fi}%
```

35.46 \xintGeq

Rewritten completely in 1.08a to be less dumb when comparing fractions having big powers of tens.

```
\def\xintGeq {\romannumeral0\xintgeq }%
\def\xintgeq #1%
{%
    \expandafter\xint_fgeq\expandafter {\romannumeral0\xintabs {#1}}%
}%
\def\xint_fgeq #1#2%
{%
    \expandafter\XINT_fgeq_A \romannumeral0\xintabs {#2}#1%
}%
\def\XINT_fgeq_A #1%
{%
    \xint_gob_til_zero #1\XINT_fgeq_Zii 0%
    \XINT_fgeq_B #1%
}%
\def\XINT_fgeq_Zii 0\XINT_fgeq_B #1[#2]#3[#4]{ 1}%
\def\XINT_fgeq_B #1/#2[#3]#4#5/#6[#7]%
```

```
{%
    \xint_gob_til_zero #4\XINT_fgeq_Zi 0%
    \expandafter\XINT_fgeq_C\expandafter
    {\the\numexpr #7-#3\expandafter}\expandafter
    {\romannumeral0\xintiimul {#4#5}{#2}}%
    {\romannumeral0\xintiimul {#6}{#1}}%
}%
\def\XINT_fgeq_Zi 0#1#2#3#4#5#6#7{ 0}%
\def\XINT_fgeq_C #1#2#3%
{%
    \expandafter\XINT_fgeq_D\expandafter
    {#3}{#1}{#2}%
}%
\def\XINT_fgeq_D #1#2#3%
{%
    \expandafter\XINT_cntSgnFork\romannumeral-‘0\expandafter\XINT_cntSgn
     \the\numexpr #2+\xintLength{#3}-\xintLength{#1}\relax\Z
    { 0}{\XINT_fgeq_E #2\Z {#3}{#1}}{ 1}%
}%
\def\XINT_fgeq_E #1%
{%
    \xint_UDsignfork
        #1\XINT_fgeq_Fd
         -{\XINT_fgeq_Fn #1}%
    \krof
}%
\def\XINT_fgeq_Fd #1\Z #2#3%
{%
    \expandafter\XINT_fgeq_Fe\expandafter
    {\romannumeral0\XINT_dsx_addzerosnofuss {#1}{#3}}{#2}%
}%
\def\XINT_fgeq_Fe #1#2{\XINT_geq_pre {#2}{#1}}%
\def\XINT_fgeq_Fn #1\Z #2#3%
{%
    \expandafter\XINT_geq_pre\expandafter
    {\romannumeral0\XINT_dsx_addzerosnofuss {#1}{#2}}{#3}%
}%
```

35.47 \xintMax

Rewritten completely in 1.08a.

```
\def\xintMax {\romannumeral0\xintmax }%
\def\xintmax #1%
{%
    \expandafter\xint_fmax\expandafter {\romannumeral0\xintraw {#1}}%
}%
\def\xint_fmax #1#2%
{%
```

```
    \expandafter\XINT_fmax_A\romannumeral0\xintraw {#2}#1%
}%
\def\XINT_fmax_A #1#2/#3[#4]#5#6/#7[#8]%
{%
    \xint_UDsignsfork
      #1#5\XINT_fmax_minusminus
       -#5\XINT_fmax_firstneg
       #1-\XINT_fmax_secondneg
        --\XINT_fmax_nonneg_a
    \krof
    #1#5{#2/#3[#4]}{#6/#7[#8]}%
}%
\def\XINT_fmax_minusminus --%
   {\expandafter\xint_minus_thenstop\romannumeral0\XINT_fmin_nonneg_b }%
\def\XINT_fmax_firstneg #1-#2#3{ #1#2}%
\def\XINT_fmax_secondneg -#1#2#3{ #1#3}%
\def\XINT_fmax_nonneg_a #1#2#3#4%
{%
    \XINT_fmax_nonneg_b {#1#3}{#2#4}%
}%
\def\XINT_fmax_nonneg_b #1#2%
{%
    \if0\romannumeral0\XINT_fgeq_A #1#2%
          \xint_afterfi{ #1}%
    \else \xint_afterfi{ #2}%
    \fi
}%
```

35.48 \xintMaxof

```
\def\xintMaxof      {\romannumeral0\xintmaxof }%
\def\xintmaxof    #1{\expandafter\XINT_maxof_a\romannumeral-`0#1\relax }%
\def\XINT_maxof_a #1{\expandafter\XINT_maxof_b\romannumeral0\xintraw{#1}\Z }%
\def\XINT_maxof_b #1\Z #2%
           {\expandafter\XINT_maxof_c\romannumeral-`0#2\Z {#1}\Z}%
\def\XINT_maxof_c #1%
           {\xint_gob_til_relax #1\XINT_maxof_e\relax\XINT_maxof_d #1}%
\def\XINT_maxof_d #1\Z
           {\expandafter\XINT_maxof_b\romannumeral0\xintmax {#1}}%
\def\XINT_maxof_e #1\Z #2\Z { #2}%
```

35.49 \xintMin

Rewritten completely in 1.08a.

```
\def\xintMin {\romannumeral0\xintmin }%
\def\xintmin #1%
{%
    \expandafter\xint_fmin\expandafter {\romannumeral0\xintraw {#1}}%
}%
```

```
\def\xint_fmin #1#2%
{%
    \expandafter\XINT_fmin_A\romannumeral0\xintraw {#2}#1%
}%
\def\XINT_fmin_A #1#2/#3[#4]#5#6/#7[#8]%
{%
    \xint_UDsignsfork
      #1#5\XINT_fmin_minusminus
       -#5\XINT_fmin_firstneg
       #1-\XINT_fmin_secondneg
        --\XINT_fmin_nonneg_a
    \krof
    #1#5{#2/#3[#4]}{#6/#7[#8]}%
}%
\def\XINT_fmin_minusminus --%
   {\expandafter\xint_minus_thenstop\romannumeral0\XINT_fmax_nonneg_b }%
\def\XINT_fmin_firstneg #1-#2#3{ -#3}%
\def\XINT_fmin_secondneg -#1#2#3{ -#2}%
\def\XINT_fmin_nonneg_a #1#2#3#4%
{%
    \XINT_fmin_nonneg_b {#1#3}{#2#4}%
}%
\def\XINT_fmin_nonneg_b #1#2%
{%
    \if0\romannumeral0\XINT_fgeq_A #1#2%
          \xint_afterfi{ #2}%
    \else \xint_afterfi{ #1}%
    \fi
}%
```

35.50 \xintMinof

```
\def\xintMinof      {\romannumeral0\xintminof }%
\def\xintminof    #1{\expandafter\XINT_minof_a\romannumeral-`0#1\relax }%
\def\XINT_minof_a #1{\expandafter\XINT_minof_b\romannumeral0\xintraw{#1}\Z }%
\def\XINT_minof_b #1\Z #2%
            {\expandafter\XINT_minof_c\romannumeral-`0#2\Z {#1}\Z}%
\def\XINT_minof_c #1%
            {\xint_gob_til_relax #1\XINT_minof_e\relax\XINT_minof_d #1}%
\def\XINT_minof_d #1\Z
            {\expandafter\XINT_minof_b\romannumeral0\xintmin {#1}}%
\def\XINT_minof_e #1\Z #2\Z { #2}%
```

35.51 \xintCmp

Rewritten completely in 1.08a to be less dumb when comparing fractions having big powers of tens. Incredibly, it seems that 1.08b introduced a bug in delimited arguments making the macro just non-functional when one of the input was zero! I did not detect this until working on release 1.09a, somehow I had not tested that \xintCmp just did NOT work! I must have done some last minute change...

```
\def\xintCmp {\romannumeral0\xintcmp }%
\def\xintcmp #1%
{%
    \expandafter\xint_fcmp\expandafter {\romannumeral0\xintraw {#1}}%
}%
\def\xint_fcmp #1#2%
{%
    \expandafter\XINT_fcmp_A\romannumeral0\xintraw {#2}#1%
}%
\def\XINT_fcmp_A #1#2/#3[#4]#5#6/#7[#8]%
{%
    \xint_UDsignsfork
      #1#5\XINT_fcmp_minusminus
       -#5\XINT_fcmp_firstneg
       #1-\XINT_fcmp_secondneg
        --\XINT_fcmp_nonneg_a
    \krof
    #1#5{#2/#3[#4]}{#6/#7[#8]}%
}%
\def\XINT_fcmp_minusminus --#1#2{\XINT_fcmp_B #2#1}%
\def\XINT_fcmp_firstneg #1-#2#3{ -1}%
\def\XINT_fcmp_secondneg -#1#2#3{ 1}%
\def\XINT_fcmp_nonneg_a #1#2%
{%
    \xint_UDzerosfork
      #1#2\XINT_fcmp_zerozero
       0#2\XINT_fcmp_firstzero
       #10\XINT_fcmp_secondzero
        00\XINT_fcmp_pos
    \krof
    #1#2%
}%
\def\XINT_fcmp_zerozero   #1#2#3#4{ 0}%  1.08b had some [ and ] here!!!
\def\XINT_fcmp_firstzero  #1#2#3#4{ -1}% incredibly I never saw that until
\def\XINT_fcmp_secondzero #1#2#3#4{ 1}%  preparing 1.09a.
\def\XINT_fcmp_pos #1#2#3#4%
{%
    \XINT_fcmp_B #1#3#2#4%
}%
\def\XINT_fcmp_B #1/#2[#3]#4/#5[#6]%
{%
    \expandafter\XINT_fcmp_C\expandafter
    {\the\numexpr #6-#3\expandafter}\expandafter
    {\romannumeral0\xintiimul {#4}{#2}}%
    {\romannumeral0\xintiimul {#5}{#1}}%
}%
\def\XINT_fcmp_C #1#2#3%
{%
    \expandafter\XINT_fcmp_D\expandafter
```

```
    {#3}{#1}{#2}%
}%
\def\XINT_fcmp_D #1#2#3%
{%
    \expandafter\XINT_cntSgnFork\romannumeral-`0\expandafter\XINT_cntSgn
    \the\numexpr #2+\xintLength{#3}-\xintLength{#1}\relax\Z
    { -1}{\XINT_fcmp_E #2\Z {#3}{#1}}{ 1}%
}%
\def\XINT_fcmp_E #1%
{%
    \xint_UDsignfork
        #1\XINT_fcmp_Fd
         -{\XINT_fcmp_Fn #1}%
    \krof
}%
\def\XINT_fcmp_Fd #1\Z #2#3%
{%
    \expandafter\XINT_fcmp_Fe\expandafter
    {\romannumeral0\XINT_dsx_addzerosnofuss {#1}{#3}}{#2}%
}%
\def\XINT_fcmp_Fe #1#2{\XINT_cmp_pre {#2}{#1}}%
\def\XINT_fcmp_Fn #1\Z #2#3%
{%
    \expandafter\XINT_cmp_pre\expandafter
    {\romannumeral0\XINT_dsx_addzerosnofuss {#1}{#2}}{#3}%
}%
```

35.52 \xintAbs

Simplified in 1.09i. (original macro was written before \xintRaw)

```
\def\xintAbs   {\romannumeral0\xintabs }%
\def\xintabs #1{\expandafter\XINT_abs\romannumeral0\xintraw {#1}}%
```

35.53 \xintOpp

caution that -#1 would not be ok if #1 has [n] stuff. Simplified in 1.09i. (original macro was written before \xintRaw)

```
\def\xintOpp   {\romannumeral0\xintopp }%
\def\xintopp #1{\expandafter\XINT_opp\romannumeral0\xintraw {#1}}%
```

35.54 \xintSgn

Simplified in 1.09i. (original macro was written before \xintRaw)

```
\def\xintSgn   {\romannumeral0\xintsgn }%
\def\xintsgn #1{\expandafter\XINT_sgn\romannumeral0\xintraw {#1}\Z }%
```

35.55 \xintFloatAdd, \XINTinFloatAdd

1.07; 1.09ka improves a bit the efficieny of the coding of \XINT_FL_Add_d.

```
\def\xintFloatAdd      {\romannumeral0\xintfloatadd }%
\def\xintfloatadd    #1{\XINT_fladd_chkopt \xintfloat #1\Z }%
\def\XINTinFloatAdd    {\romannumeral0\XINTinfloatadd }%
\def\XINTinfloatadd  #1{\XINT_fladd_chkopt \XINTinfloat #1\Z }%
\def\XINT_fladd_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_fladd_opt
       \else\expandafter\XINT_fladd_noopt
    \fi  #1#2%
}%
\def\XINT_fladd_noopt #1#2\Z #3%
{%
    #1[\XINTdigits]{\XINT_FL_Add {\XINTdigits+\xint_c_ii}{#2}{#3}}%
}%
\def\XINT_fladd_opt #1[\Z #2]#3#4%
{%
    #1[#2]{\XINT_FL_Add {#2+\xint_c_ii}{#3}{#4}}%
}%
\def\XINT_FL_Add #1#2%
{%
    \expandafter\XINT_FL_Add_a\expandafter{\the\numexpr #1\expandafter}%
    \expandafter{\romannumeral0\XINTinfloat [#1]{#2}}%
}%
\def\XINT_FL_Add_a #1#2#3%
{%
    \expandafter\XINT_FL_Add_b\romannumeral0\XINTinfloat [#1]{#3}#2{#1}%
}%
\def\XINT_FL_Add_b #1%
{%
    \xint_gob_til_zero #1\XINT_FL_Add_zero 0\XINT_FL_Add_c #1%
}%
\def\XINT_FL_Add_c #1[#2]#3%
{%
    \xint_gob_til_zero #3\XINT_FL_Add_zerobis 0\XINT_FL_Add_d #1[#2]#3%
}%
\def\XINT_FL_Add_d #1[#2]#3[#4]#5%
{%
    \ifnum \numexpr #2-#4-#5>\xint_c_i
       \expandafter \xint_secondofthree_thenstop
    \else
       \ifnum \numexpr #4-#2-#5>\xint_c_i
              \expandafter\expandafter\expandafter\xint_thirdofthree_thenstop
       \fi
    \fi
    \xintadd {#1[#2]}{#3[#4]}%
```

```
}%
\def\XINT_FL_Add_zero 0\XINT_FL_Add_c 0[0]#1[#2]#3{#1[#2]}%
\def\XINT_FL_Add_zerobis 0\XINT_FL_Add_d #1[#2]0[0]#3{#1[#2]}%
```

35.56 \xintFloatSub, \XINTinFloatSub

1.07

```
\def\xintFloatSub {\romannumeral0\xintfloatsub }%
\def\xintfloatsub    #1{\XINT_flsub_chkopt \xintfloat #1\Z }%
\def\XINTinFloatSub {\romannumeral0\XINTinfloatsub }%
\def\XINTinfloatsub #1{\XINT_flsub_chkopt \XINTinfloat #1\Z }%
\def\XINT_flsub_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_flsub_opt
       \else\expandafter\XINT_flsub_noopt
    \fi  #1#2%
}%
\def\XINT_flsub_noopt #1#2\Z #3%
{%
    #1[\XINTdigits]{\XINT_FL_Add {\XINTdigits+\xint_c_ii}{#2}{\xintOpp{#3}}}%
}%
\def\XINT_flsub_opt #1[\Z #2]#3#4%
{%
    #1[#2]{\XINT_FL_Add {#2+\xint_c_ii}{#3}{\xintOpp{#4}}}%
}%
```

35.57 \xintFloatMul, \XINTinFloatMul

1.07

```
\def\xintFloatMul    {\romannumeral0\xintfloatmul}%
\def\xintfloatmul    #1{\XINT_flmul_chkopt \xintfloat #1\Z }%
\def\XINTinFloatMul {\romannumeral0\XINTinfloatmul }%
\def\XINTinfloatmul #1{\XINT_flmul_chkopt \XINTinfloat #1\Z }%
\def\XINT_flmul_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_flmul_opt
       \else\expandafter\XINT_flmul_noopt
    \fi  #1#2%
}%
\def\XINT_flmul_noopt #1#2\Z #3%
{%
    #1[\XINTdigits]{\XINT_FL_Mul {\XINTdigits+\xint_c_ii}{#2}{#3}}%
}%
\def\XINT_flmul_opt #1[\Z #2]#3#4%
{%
    #1[#2]{\XINT_FL_Mul {#2+\xint_c_ii}{#3}{#4}}%
```

```
}%
\def\XINT_FL_Mul #1#2%
{%
    \expandafter\XINT_FL_Mul_a\expandafter{\the\numexpr #1\expandafter}%
    \expandafter{\romannumeral0\XINTinfloat [#1]{#2}}%
}%
\def\XINT_FL_Mul_a #1#2#3%
{%
    \expandafter\XINT_FL_Mul_b\romannumeral0\XINTinfloat [#1]{#3}#2%
}%
\def\XINT_FL_Mul_b #1[#2]#3[#4]{\xintE{\xintiiMul {#1}{#3}}{#2+#4}}%
```

35.58 \xintFloatDiv, \XINTinFloatDiv

1.07

```
\def\xintFloatDiv    {\romannumeral0\xintfloatdiv}%
\def\xintfloatdiv    #1{\XINT_fldiv_chkopt \xintfloat #1\Z }%
\def\XINTinFloatDiv  {\romannumeral0\XINTinfloatdiv }%
\def\XINTinfloatdiv  #1{\XINT_fldiv_chkopt \XINTinfloat #1\Z }%
\def\XINT_fldiv_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_fldiv_opt
       \else\expandafter\XINT_fldiv_noopt
    \fi  #1#2%
}%
\def\XINT_fldiv_noopt #1#2\Z #3%
{%
    #1[\XINTdigits]{\XINT_FL_Div {\XINTdigits+\xint_c_ii}{#2}{#3}}%
}%
\def\XINT_fldiv_opt #1[\Z #2]#3#4%
{%
    #1[#2]{\XINT_FL_Div {#2+\xint_c_ii}{#3}{#4}}%
}%
\def\XINT_FL_Div #1#2%
{%
    \expandafter\XINT_FL_Div_a\expandafter{\the\numexpr #1\expandafter}%
    \expandafter{\romannumeral0\XINTinfloat [#1]{#2}}%
}%
\def\XINT_FL_Div_a #1#2#3%
{%
    \expandafter\XINT_FL_Div_b\romannumeral0\XINTinfloat [#1]{#3}#2%
}%
\def\XINT_FL_Div_b #1[#2]#3[#4]{\xintE{#3/#1}{#4-#2}}%
```

35.59 \XINTinFloatSum

1.09a: quick write-up, for use by \xintfloatexpr, will need to be thought through again. Renamed (and slightly modified) in 1.09h. Should be extended for optional precision. Should be rewritten for optimization.

```
\def\XINTinFloatSum    {\romannumeral0\XINTinfloatsum }%
\def\XINTinfloatsum #1{\expandafter\XINT_floatsum_a\romannumeral-‘0#1\relax }%
\def\XINT_floatsum_a #1{\expandafter\XINT_floatsum_b
                         \romannumeral0\XINTinfloat[\XINTdigits]{#1}\Z }%
\def\XINT_floatsum_b #1\Z #2%
            {\expandafter\XINT_floatsum_c\romannumeral-‘0#2\Z {#1}\Z}%
\def\XINT_floatsum_c #1%
            {\xint_gob_til_relax #1\XINT_floatsum_e\relax\XINT_floatsum_d #1}%
\def\XINT_floatsum_d #1\Z
            {\expandafter\XINT_floatsum_b\romannumeral0\XINTinfloatadd {#1}}%
\def\XINT_floatsum_e #1\Z #2\Z { #2}%
```

35.60 \XINTinFloatPrd

1.09a: quick write-up, for use by \xintfloatexpr, will need to be thought through again. Renamed (and slightly modified) in 1.09h. Should be extended for optional precision. Should be rewritten for optimization.

```
\def\XINTinFloatPrd    {\romannumeral0\XINTinfloatprd }%
\def\XINTinfloatprd #1{\expandafter\XINT_floatprd_a\romannumeral-‘0#1\relax }%
\def\XINT_floatprd_a #1{\expandafter\XINT_floatprd_b
                         \romannumeral0\XINTinfloat[\XINTdigits]{#1}\Z }%
\def\XINT_floatprd_b #1\Z #2%
            {\expandafter\XINT_floatprd_c\romannumeral-‘0#2\Z {#1}\Z}%
\def\XINT_floatprd_c #1%
            {\xint_gob_til_relax #1\XINT_floatprd_e\relax\XINT_floatprd_d #1}%
\def\XINT_floatprd_d #1\Z
            {\expandafter\XINT_floatprd_b\romannumeral0\XINTinfloatmul {#1}}%
\def\XINT_floatprd_e #1\Z #2\Z { #2}%
```

35.61 \xintFloatPow, \XINTinFloatPow

1.07. Release 1.09j has re-organized the core loop, and \XINT_flpow_prd subroutine has been removed.

```
\def\xintFloatPow {\romannumeral0\xintfloatpow}%
\def\xintfloatpow #1{\XINT_flpow_chkopt \xintfloat #1\Z }%
\def\XINTinFloatPow {\romannumeral0\XINTinfloatpow }%
\def\XINTinfloatpow #1{\XINT_flpow_chkopt \XINTinfloat #1\Z }%
\def\XINT_flpow_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_flpow_opt
       \else\expandafter\XINT_flpow_noopt
    \fi
     #1#2%
}%
```

```
\def\XINT_flpow_noopt  #1#2\Z #3%
{%
   \expandafter\XINT_flpow_checkB_start\expandafter
                 {\the\numexpr #3\expandafter}\expandafter
                 {\the\numexpr \XINTdigits}{#2}{#1[\XINTdigits]}%
}%
\def\XINT_flpow_opt #1[\Z #2]#3#4%
{%
   \expandafter\XINT_flpow_checkB_start\expandafter
                {\the\numexpr #4\expandafter}\expandafter
                {\the\numexpr #2}{#3}{#1[#2]}%
}%
\def\XINT_flpow_checkB_start #1{\XINT_flpow_checkB_a #1\Z }%
\def\XINT_flpow_checkB_a #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_flpow_BisZero
      0#1{\XINT_flpow_checkB_b 1}%
       0-{\XINT_flpow_checkB_b 0#1}%
    \krof
}%
\def\XINT_flpow_BisZero \Z #1#2#3{#3{1/1[0]}}%
\def\XINT_flpow_checkB_b #1#2\Z #3%
{%
    \expandafter\XINT_flpow_checkB_c \expandafter
    {\romannumeral0\xintlength{#2}}{#3}{#2}#1%
}%
\def\XINT_flpow_checkB_c #1#2%
{%
    \expandafter\XINT_flpow_checkB_d \expandafter
    {\the\numexpr \expandafter\xintLength\expandafter
                   {\the\numexpr #1*20/\xint_c_iii }+#1+#2+\xint_c_i }%
}%
\def\XINT_flpow_checkB_d #1#2#3#4%
{%
    \expandafter \XINT_flpow_a
    \romannumeral0\XINTinfloat [#1]{#4}{#1}{#2}#3%
}%
\def\XINT_flpow_a #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_flpow_zero
      0#1{\XINT_flpow_b 1}%
       0-{\XINT_flpow_b 0#1}%
    \krof
}%
\def\XINT_flpow_b #1#2[#3]#4#5%
{%
    \XINT_flpow_loopI {#5}{#2[#3]}{\romannumeral0\XINTinfloatmul [#4]}%
```

```
    {#1*\ifodd #5 1\else 0\fi}%
}%
\def\XINT_flpow_zero [#1]#2#3#4#5%
% xint is not equipped to signal infinity, the 2^31 will provoke
% deliberately a number too big and arithmetic overflow in \XINT_float_Xb
{%
    \if #41\xint_afterfi {\xintError:DivisionByZero #5{1[2147483648]}}%
    \else \xint_afterfi {#5{0[0]}}\fi
}%
\def\XINT_flpow_loopI #1%
{%
    \ifnum #1=\xint_c_i\XINT_flpow_ItoIII\fi
    \ifodd #1
       \expandafter\XINT_flpow_loopI_odd
    \else
       \expandafter\XINT_flpow_loopI_even
    \fi
    {#1}%
}%
\def\XINT_flpow_ItoIII\fi #1\fi #2#3#4#5%
{%
    \fi\expandafter\XINT_flpow_III\the\numexpr #5\relax #3%
}%
\def\XINT_flpow_loopI_even #1#2#3%
{%
    \expandafter\XINT_flpow_loopI\expandafter
    {\the\numexpr #1/\xint_c_ii\expandafter}\expandafter
    {#3{#2}{#2}}{#3}%
}%
\def\XINT_flpow_loopI_odd #1#2#3%
{%
    \expandafter\XINT_flpow_loopII\expandafter
    {\the\numexpr #1/\xint_c_ii-\xint_c_i\expandafter}\expandafter
    {#3{#2}{#2}}{#3}{#2}%
}%
\def\XINT_flpow_loopII #1%
{%
    \ifnum #1 = \xint_c_i\XINT_flpow_IItoIII\fi
    \ifodd #1
       \expandafter\XINT_flpow_loopII_odd
    \else
       \expandafter\XINT_flpow_loopII_even
    \fi
    {#1}%
}%
\def\XINT_flpow_loopII_even #1#2#3%
{%
    \expandafter\XINT_flpow_loopII\expandafter
    {\the\numexpr #1/\xint_c_ii\expandafter}\expandafter
```

```
    {#3{#2}{#2}}{#3}%
}%
\def\XINT_flpow_loopII_odd #1#2#3#4%
{%
    \expandafter\XINT_flpow_loopII_odda\expandafter
    {#3{#2}{#4}}{#1}{#2}{#3}%
}%
\def\XINT_flpow_loopII_odda #1#2#3#4%
{%
    \expandafter\XINT_flpow_loopII\expandafter
    {\the\numexpr #2/\xint_c_ii-\xint_c_i\expandafter}\expandafter
    {#4{#3}{#3}}{#4}{#1}%
}%
\def\XINT_flpow_IItoIII\fi #1\fi #2#3#4#5#6%
{%
    \fi\expandafter\XINT_flpow_III\the\numexpr #6\expandafter\relax
    #4{#3}{#5}%
}%
\def\XINT_flpow_III #1#2[#3]#4%
{%
    \expandafter\XINT_flpow_IIIend\expandafter
    {\the\numexpr\if #41-\fi#3\expandafter}%
    \xint_UDzerofork
        #4{{#2}}%
         0{{1/#2}}%
    \krof #1%
}%
\def\XINT_flpow_IIIend #1#2#3#4%
{%
    \xint_UDzerofork
    #3{#4{#2[#1]}}%
     0{#4{-#2[#1]}}%
    \krof
}%
```

35.62 \xintFloatPower, \XINTinFloatPower

1.07. The core loop has been re-organized in 1.09j for some slight efficiency gain.

```
\def\xintFloatPower {\romannumeral0\xintfloatpower}%
\def\xintfloatpower #1{\XINT_flpower_chkopt \xintfloat #1\Z }%
\def\XINTinFloatPower {\romannumeral0\XINTinfloatpower}%
\def\XINTinfloatpower #1{\XINT_flpower_chkopt \XINTinfloat #1\Z }%
\def\XINT_flpower_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_flpower_opt
       \else\expandafter\XINT_flpower_noopt
    \fi
```

```
        #1#2%
}%
\def\XINT_flpower_noopt  #1#2\Z #3%
{%
   \expandafter\XINT_flpower_checkB_start\expandafter
                 {\the\numexpr \XINTdigits\expandafter}\expandafter
                 {\romannumeral0\xintnum{#3}}{#2}{#1[\XINTdigits]}%
}%
\def\XINT_flpower_opt #1[\Z #2]#3#4%
{%
   \expandafter\XINT_flpower_checkB_start\expandafter
                {\the\numexpr #2\expandafter}\expandafter
                {\romannumeral0\xintnum{#4}}{#3}{#1[#2]}%
}%
\def\XINT_flpower_checkB_start #1#2{\XINT_flpower_checkB_a #2\Z {#1}}%
\def\XINT_flpower_checkB_a #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_flpower_BisZero
      0#1{\XINT_flpower_checkB_b 1}%
       0-{\XINT_flpower_checkB_b 0#1}%
    \krof
}%
\def\XINT_flpower_BisZero \Z #1#2#3{#3{1/1[0]}}%
\def\XINT_flpower_checkB_b #1#2\Z #3%
{%
    \expandafter\XINT_flpower_checkB_c \expandafter
    {\romannumeral0\xintlength{#2}}{#3}{#2}#1%
}%
\def\XINT_flpower_checkB_c #1#2%
{%
    \expandafter\XINT_flpower_checkB_d \expandafter
    {\the\numexpr \expandafter\xintLength\expandafter
                  {\the\numexpr #1*20/\xint_c_iii }+#1+#2+\xint_c_i }%
}%
\def\XINT_flpower_checkB_d #1#2#3#4%
{%
    \expandafter \XINT_flpower_a
    \romannumeral0\XINTinfloat [#1]{#4}{#1}{#2}#3%
}%
\def\XINT_flpower_a #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_flpow_zero
      0#1{\XINT_flpower_b 1}%
       0-{\XINT_flpower_b 0#1}%
    \krof
}%
\def\XINT_flpower_b #1#2[#3]#4#5%
```

```
{%
    \XINT_flpower_loopI {#5}{#2[#3]}{\romannumeral0\XINTinfloatmul [#4]}%
    {#1*\xintiiOdd {#5}}%
}%
\def\XINT_flpower_loopI #1%
{%
    \if1\XINT_isOne {#1}\XINT_flpower_ItoIII\fi
    \if1\xintiiOdd{#1}%
       \expandafter\expandafter\expandafter\XINT_flpower_loopI_odd
    \else
       \expandafter\expandafter\expandafter\XINT_flpower_loopI_even
    \fi
    \expandafter {\romannumeral0\xinthalf{#1}}%
}%
\def\XINT_flpower_ItoIII\fi #1\fi\expandafter #2#3#4#5%
{%
    \fi\expandafter\XINT_flpow_III \the\numexpr #5\relax #3%
}%
\def\XINT_flpower_loopI_even #1#2#3%
{%
    \expandafter\XINT_flpower_toI\expandafter {#3{#2}{#2}}{#1}{#3}%
}%
\def\XINT_flpower_loopI_odd #1#2#3%
{%
    \expandafter\XINT_flpower_toII\expandafter {#3{#2}{#2}}{#1}{#3}{#2}%
}%
\def\XINT_flpower_toI  #1#2{\XINT_flpower_loopI  {#2}{#1}}%
\def\XINT_flpower_toII #1#2{\XINT_flpower_loopII {#2}{#1}}%
\def\XINT_flpower_loopII #1%
{%
    \if1\XINT_isOne {#1}\XINT_flpower_IItoIII\fi
    \if1\xintiiOdd{#1}%
       \expandafter\expandafter\expandafter\XINT_flpower_loopII_odd
    \else
       \expandafter\expandafter\expandafter\XINT_flpower_loopII_even
    \fi
    \expandafter {\romannumeral0\xinthalf{#1}}%
}%
\def\XINT_flpower_loopII_even #1#2#3%
{%
    \expandafter\XINT_flpower_toII\expandafter
    {#3{#2}{#2}}{#1}{#3}%
}%
\def\XINT_flpower_loopII_odd #1#2#3#4%
{%
    \expandafter\XINT_flpower_loopII_odda\expandafter
    {#3{#2}{#4}}{#2}{#3}{#1}%
}%
\def\XINT_flpower_loopII_odda #1#2#3#4%
```

```
{%
    \expandafter\XINT_flpower_toII\expandafter
    {#3{#2}{#2}}{#4}{#3}{#1}%
}%
\def\XINT_flpower_IItoIII\fi #1\fi\expandafter #2#3#4#5#6%
{%
    \fi\expandafter\XINT_flpow_III\the\numexpr #6\expandafter\relax
    #4{#3}{#5}%
}%
```

35.63 \xintFloatSqrt, \XINTinFloatSqrt

1.08

```
\def\xintFloatSqrt     {\romannumeral0\xintfloatsqrt }%
\def\xintfloatsqrt   #1{\XINT_flsqrt_chkopt \xintfloat #1\Z }%
\def\XINTinFloatSqrt   {\romannumeral0\XINTinfloatsqrt }%
\def\XINTinfloatsqrt #1{\XINT_flsqrt_chkopt \XINTinfloat #1\Z }%
\def\XINT_flsqrt_chkopt #1#2%
{%
    \ifx [#2\expandafter\XINT_flsqrt_opt
       \else\expandafter\XINT_flsqrt_noopt
    \fi  #1#2%
}%
\def\XINT_flsqrt_noopt #1#2\Z
{%
    #1[\XINTdigits]{\XINT_FL_sqrt \XINTdigits {#2}}%
}%
\def\XINT_flsqrt_opt #1[\Z #2]#3%
{%
    #1[#2]{\XINT_FL_sqrt {#2}{#3}}%
}%
\def\XINT_FL_sqrt #1%
{%
    \ifnum\numexpr #1<\xint_c_xviii
        \xint_afterfi {\XINT_FL_sqrt_a\xint_c_xviii}%
    \else
        \xint_afterfi {\XINT_FL_sqrt_a {#1+\xint_c_i}}%
    \fi
}%
\def\XINT_FL_sqrt_a #1#2%
{%
    \expandafter\XINT_FL_sqrt_checkifzeroorneg
    \romannumeral0\XINTinfloat [#1]{#2}%
}%
\def\XINT_FL_sqrt_checkifzeroorneg #1%
{%
    \xint_UDzerominusfork
     #1-\XINT_FL_sqrt_iszero
```

```
        0#1\XINT_FL_sqrt_isneg
         0-{\XINT_FL_sqrt_b #1}%
    \krof
}%
\def\XINT_FL_sqrt_iszero #1[#2]{0[0]}%
\def\XINT_FL_sqrt_isneg  #1[#2]{\xintError:RootOfNegative 0[0]}%
\def\XINT_FL_sqrt_b #1[#2]%
{%
    \ifodd #2
        \xint_afterfi{\XINT_FL_sqrt_c 01}%
    \else
        \xint_afterfi{\XINT_FL_sqrt_c {}0}%
    \fi
    {#1}{#2}%
}%
\def\XINT_FL_sqrt_c #1#2#3#4%
{%
    \expandafter\XINT_flsqrt\expandafter {\the\numexpr #4-#2}{#3#1}%
}%
\def\XINT_flsqrt #1#2%
{%
    \expandafter\XINT_sqrt_a
    \expandafter{\romannumeral0\xintlength {#2}}\XINT_flsqrt_big_d {#2}{#1}%
}%
\def\XINT_flsqrt_big_d #1#2%
{%
   \ifodd #2
     \expandafter\expandafter\expandafter\XINT_flsqrt_big_eB
   \else
     \expandafter\expandafter\expandafter\XINT_flsqrt_big_eA
   \fi
   \expandafter {\the\numexpr (#2-\xint_c_i)/\xint_c_ii }{#1}%
}%
\def\XINT_flsqrt_big_eA  #1#2#3%
{%
    \XINT_flsqrt_big_eA_a #3\Z {#2}{#1}{#3}%
}%
\def\XINT_flsqrt_big_eA_a #1#2#3#4#5#6#7#8#9\Z
{%
    \XINT_flsqrt_big_eA_b {#1#2#3#4#5#6#7#8}%
}%
\def\XINT_flsqrt_big_eA_b #1#2%
{%
    \expandafter\XINT_flsqrt_big_f
    \romannumeral0\XINT_flsqrt_small_e {#2001}{#1}%
}%
\def\XINT_flsqrt_big_eB #1#2#3%
{%
    \XINT_flsqrt_big_eB_a #3\Z {#2}{#1}{#3}%
```

```
}%
\def\XINT_flsqrt_big_eB_a #1#2#3#4#5#6#7#8#9%
{%
    \XINT_flsqrt_big_eB_b {#1#2#3#4#5#6#7#8#9}%
}%
\def\XINT_flsqrt_big_eB_b #1#2\Z #3%
{%
    \expandafter\XINT_flsqrt_big_f
    \romannumeral0\XINT_flsqrt_small_e {#30001}{#1}%
}%
\def\XINT_flsqrt_small_e #1#2%
{%
   \expandafter\XINT_flsqrt_small_f\expandafter
   {\the\numexpr #1*#1-#2-\xint_c_i}{#1}%
}%
\def\XINT_flsqrt_small_f #1#2%
{%
   \expandafter\XINT_flsqrt_small_g\expandafter
   {\the\numexpr (#1+#2)/(2*#2)-\xint_c_i }{#1}{#2}%
}%
\def\XINT_flsqrt_small_g #1%
{%
    \ifnum #1>\xint_c_
       \expandafter\XINT_flsqrt_small_h
    \else
       \expandafter\XINT_flsqrt_small_end
    \fi
    {#1}%
}%
\def\XINT_flsqrt_small_h #1#2#3%
{%
    \expandafter\XINT_flsqrt_small_f\expandafter
    {\the\numexpr #2-\xint_c_ii*#1*#3+#1*#1\expandafter}\expandafter
    {\the\numexpr #3-#1}%
}%
\def\XINT_flsqrt_small_end #1#2#3%
{%
    \expandafter\space\expandafter
    {\the\numexpr \xint_c_i+#3*\xint_c_x^iv-
                      (#2*\xint_c_x^iv+#3)/(\xint_c_ii*#3)}%
}%
\def\XINT_flsqrt_big_f #1%
{%
    \expandafter\XINT_flsqrt_big_fa\expandafter
    {\romannumeral0\xintiisqr {#1}}{#1}%
}%
\def\XINT_flsqrt_big_fa #1#2#3#4%
{%
    \expandafter\XINT_flsqrt_big_fb\expandafter
```

```
    {\romannumeral0\XINT_dsx_addzerosnofuss
                     {\numexpr #3-\xint_c_viii\relax}{#2}}%
    {\romannumeral0\xintiisub
      {\XINT_dsx_addzerosnofuss
           {\numexpr \xint_c_ii*(#3-\xint_c_viii)\relax}{#1}}{#4}}%
    {#3}%
}%
\def\XINT_flsqrt_big_fb #1#2%
{%
    \expandafter\XINT_flsqrt_big_g\expandafter {#2}{#1}%
}%
\def\XINT_flsqrt_big_g #1#2%
{%
    \expandafter\XINT_flsqrt_big_j
    \romannumeral0\xintiidivision
    {#1}{\romannumeral0\XINT_dbl_pos #2\R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W }{#2}%
}%
\def\XINT_flsqrt_big_j #1%
{%
    \if0\XINT_Sgn #1\Z
        \expandafter \XINT_flsqrt_big_end_a
    \else \expandafter \XINT_flsqrt_big_k
    \fi {#1}%
}%
\def\XINT_flsqrt_big_k #1#2#3%
{%
    \expandafter\XINT_flsqrt_big_l\expandafter
    {\romannumeral0\XINT_sub_pre {#3}{#1}}%
    {\romannumeral0\xintiiadd {#2}{\romannumeral0\XINT_sqr {#1}}}%
}%
\def\XINT_flsqrt_big_l #1#2%
{%
   \expandafter\XINT_flsqrt_big_g\expandafter
   {#2}{#1}%
}%
\def\XINT_flsqrt_big_end_a #1#2#3#4#5%
{%
   \expandafter\XINT_flsqrt_big_end_b\expandafter
   {\the\numexpr -#4+#5/\xint_c_ii\expandafter}\expandafter
   {\romannumeral0\xintiisub
    {\XINT_dsx_addzerosnofuss {#4}{#3}}%
    {\xintHalf{\xintiiQuo{\XINT_dsx_addzerosnofuss {#4}{#2}}{#3}}}}%
}%
\def\XINT_flsqrt_big_end_b #1#2{#2[#1]}%
```

35.64 \XINTinFloatMaxof

1.09a, for use by \xintNewFloatExpr. Name changed in 1.09h

```
\def\XINTinFloatMaxof {\romannumeral0\XINTinfloatmaxof }%
\def\XINTinfloatmaxof #1{\expandafter\XINT_flmaxof_a\romannumeral-`0#1\relax }%
\def\XINT_flmaxof_a #1{\expandafter\XINT_flmaxof_b
                       \romannumeral0\XINTinfloat [\XINTdigits]{#1}\Z }%
\def\XINT_flmaxof_b #1\Z #2%
           {\expandafter\XINT_flmaxof_c\romannumeral-`0#2\Z {#1}\Z}%
\def\XINT_flmaxof_c #1%
           {\xint_gob_til_relax #1\XINT_flmaxof_e\relax\XINT_flmaxof_d #1}%
\def\XINT_flmaxof_d #1\Z
           {\expandafter\XINT_flmaxof_b\romannumeral0\xintmax
                         {\XINTinFloat [\XINTdigits]{#1}}}%
\def\XINT_flmaxof_e #1\Z #2\Z { #2}%
```

35.65 \XINTinFloatMinof

1.09a, for use by \xintNewFloatExpr. Name changed in 1.09h

```
\def\XINTinFloatMinof   {\romannumeral0\XINTinfloatminof }%
\def\XINTinfloatminof #1{\expandafter\XINT_flminof_a\romannumeral-`0#1\relax }%
\def\XINT_flminof_a #1{\expandafter\XINT_flminof_b
                       \romannumeral0\XINTinfloat [\XINTdigits]{#1}\Z }%
\def\XINT_flminof_b #1\Z #2%
           {\expandafter\XINT_flminof_c\romannumeral-`0#2\Z {#1}\Z}%
\def\XINT_flminof_c #1%
           {\xint_gob_til_relax #1\XINT_flminof_e\relax\XINT_flminof_d #1}%
\def\XINT_flminof_d #1\Z
           {\expandafter\XINT_flminof_b\romannumeral0\xintmin
                         {\XINTinFloat [\XINTdigits]{#1}}}%
\def\XINT_flminof_e #1\Z #2\Z { #2}%
```

35.66 \xintRound:csv

1.09a. For use by \xinttheiexpr.

```
\def\xintRound:csv #1{\expandafter\XINT_round:_a\romannumeral-`0#1,,^}%
\def\XINT_round:_a {\XINT_round:_b {}}%
\def\XINT_round:_b #1#2,%
             {\expandafter\XINT_round:_c\romannumeral-`0#2,{#1}}%
\def\XINT_round:_c #1{\if #1,\expandafter\XINT_:_f
                      \else\expandafter\XINT_round:_d\fi #1}%
\def\XINT_round:_d #1,%
         {\expandafter\XINT_round:_e\romannumeral0\xintiround 0{#1},}%
\def\XINT_round:_e #1,#2{\XINT_round:_b {#2,#1}}%
```

35.67 \xintFloat:csv

1.09a. For use by \xintthefloatexpr.

```
\def\xintFloat:csv #1{\expandafter\XINT_float:_a\romannumeral-`0#1,,^}%
\def\XINT_float:_a {\XINT_float:_b {}}%
\def\XINT_float:_b #1#2,%
             {\expandafter\XINT_float:_c\romannumeral-`0#2,{#1}}%
\def\XINT_float:_c #1{\if #1,\expandafter\XINT_:_f
                      \else\expandafter\XINT_float:_d\fi #1}%
\def\XINT_float:_d #1,%
         {\expandafter\XINT_float:_e\romannumeral0\xintfloat {#1},}%
\def\XINT_float:_e #1,#2{\XINT_float:_b {#2,#1}}%
```

35.68 \xintSum:csv

1.09a. For use by \xintexpr.

```
\def\xintSum:csv #1{\expandafter\XINT_sum:_a\romannumeral-`0#1,,^}%
\def\XINT_sum:_a {\XINT_sum:_b {0/1[0]}}%
\def\XINT_sum:_b #1#2,{\expandafter\XINT_sum:_c\romannumeral-`0#2,{#1}}%
\def\XINT_sum:_c #1{\if #1,\expandafter\XINT_:_e
                      \else\expandafter\XINT_sum:_d\fi #1}%
\def\XINT_sum:_d #1,#2{\expandafter\XINT_sum:_b\expandafter
                        {\romannumeral0\xintadd {#2}{#1}}}%
```

35.69 \xintPrd:csv

1.09a. For use by \xintexpr.

```
\def\xintPrd:csv #1{\expandafter\XINT_prd:_a\romannumeral-`0#1,,^}%
\def\XINT_prd:_a {\XINT_prd:_b {1/1[0]}}%
\def\XINT_prd:_b #1#2,{\expandafter\XINT_prd:_c\romannumeral-`0#2,{#1}}%
\def\XINT_prd:_c #1{\if #1,\expandafter\XINT_:_e
                      \else\expandafter\XINT_prd:_d\fi #1}%
\def\XINT_prd:_d #1,#2{\expandafter\XINT_prd:_b\expandafter
                        {\romannumeral0\xintmul {#2}{#1}}}%
```

35.70 \xintMaxof:csv

1.09a. For use by \xintexpr. Even with only one argument, there does not seem to be really a motive for using \xintraw?

```
\def\xintMaxof:csv #1{\expandafter\XINT_maxof:_b\romannumeral-`0#1,,}%
\def\XINT_maxof:_b #1,#2,{\expandafter\XINT_maxof:_c\romannumeral-`0#2,{#1},}%
\def\XINT_maxof:_c #1{\if #1,\expandafter\XINT_of:_e
                       \else\expandafter\XINT_maxof:_d\fi #1}%
\def\XINT_maxof:_d #1,{\expandafter\XINT_maxof:_b\romannumeral0\xintmax {#1}}%
```

35.71 `\xintMinof:csv`

1.09a. For use by \xintexpr.

```
\def\xintMinof:csv #1{\expandafter\XINT_minof:_b\romannumeral-`0#1,,}%
\def\XINT_minof:_b #1,#2,{\expandafter\XINT_minof:_c\romannumeral-`0#2,{#1},}%
\def\XINT_minof:_c #1{\if #1,\expandafter\XINT_of:_e
                       \else\expandafter\XINT_minof:_d\fi #1}%
\def\XINT_minof:_d #1,{\expandafter\XINT_minof:_b\romannumeral0\xintmin {#1}}%
```

35.72 `\XINTinFloatMinof:csv`

1.09a. For use by \xintfloatexpr. Name changed in 1.09h

```
\def\XINTinFloatMinof:csv #1{\expandafter\XINT_flminof:_a\romannumeral-`0#1,,}%
\def\XINT_flminof:_a #1,{\expandafter\XINT_flminof:_b
                         \romannumeral0\XINTinfloat [\XINTdigits]{#1},}%
\def\XINT_flminof:_b #1,#2,%
      {\expandafter\XINT_flminof:_c\romannumeral-`0#2,{#1},}%
\def\XINT_flminof:_c #1{\if #1,\expandafter\XINT_of:_e
                       \else\expandafter\XINT_flminof:_d\fi #1}%
\def\XINT_flminof:_d #1,%
      {\expandafter\XINT_flminof:_b\romannumeral0\xintmin
                  {\XINTinFloat [\XINTdigits]{#1}}}%
```

35.73 `\XINTinFloatMaxof:csv`

1.09a. For use by \xintfloatexpr. Name changed in 1.09h

```
\def\XINTinFloatMaxof:csv #1{\expandafter\XINT_flmaxof:_a\romannumeral-`0#1,,}%
\def\XINT_flmaxof:_a #1,{\expandafter\XINT_flmaxof:_b
                         \romannumeral0\XINTinfloat [\XINTdigits]{#1},}%
\def\XINT_flmaxof:_b #1,#2,%
      {\expandafter\XINT_flmaxof:_c\romannumeral-`0#2,{#1},}%
\def\XINT_flmaxof:_c #1{\if #1,\expandafter\XINT_of:_e
                       \else\expandafter\XINT_flmaxof:_d\fi #1}%
\def\XINT_flmaxof:_d #1,%
      {\expandafter\XINT_flmaxof:_b\romannumeral0\xintmax
                  {\XINTinFloat [\XINTdigits]{#1}}}%
```

35.74 `\XINTinFloatSum:csv`

1.09a. For use by \xintfloatexpr. Renamed in 1.09h

```
\def\XINTinFloatSum:csv #1{\expandafter\XINT_floatsum:_a\romannumeral-`0#1,,^}%
\def\XINT_floatsum:_a {\XINT_floatsum:_b {0[0]}}%
\def\XINT_floatsum:_b #1#2,%
            {\expandafter\XINT_floatsum:_c\romannumeral-`0#2,{#1}}%
```

```
\def\XINT_floatsum:_c #1{\if #1,\expandafter\XINT_:_e
                        \else\expandafter\XINT_floatsum:_d\fi #1}%
\def\XINT_floatsum:_d #1,#2{\expandafter\XINT_floatsum:_b\expandafter
                          {\romannumeral0\XINTinfloatadd {#2}{#1}}}%
```

35.75 \XINTinFloatPrd:csv

1.09a. For use by \xintfloatexpr. Renamed in 1.09h

```
\def\XINTinFloatPred:csv #1{\expandafter\XINT_floatprd:_a\romannumeral-`0#1,,^}%
\def\XINT_floatprd:_a {\XINT_floatprd:_b {1[0]}}%
\def\XINT_floatprd:_b #1#2,%
             {\expandafter\XINT_floatprd:_c\romannumeral-`0#2,{#1}}%
\def\XINT_floatprd:_c #1{\if #1,\expandafter\XINT_:_e
                        \else\expandafter\XINT_floatprd:_d\fi #1}%
\def\XINT_floatprd:_d #1,#2{\expandafter\XINT_floatprd:_b\expandafter
                          {\romannumeral0\XINTinfloatmul {#2}{#1}}}%
\XINT_restorecatcodes_endinput%
```

36 Package xintseries implementation

The commenting is currently (2014/02/13) very sparse.

Contents

36.1 Catcodes, ε-TeX and reload detection

The code for reload detection is copied from Heiko Oberdiek's packages, and adapted here to check for previous loading of the xintfrac package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
```

```
\catcode44=12   % ,
\catcode45=12   % -
\catcode46=12   % .
\catcode58=12   % :
\def\space { }%
\let\z\endgroup
\expandafter\let\expandafter\x\csname ver@xintseries.sty\endcsname
\expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
\expandafter
  \ifx\csname PackageInfo\endcsname\relax
    \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
  \else
    \def\y#1#2{\PackageInfo{#1}{#2}}%
  \fi
\expandafter
\ifx\csname numexpr\endcsname\relax
   \y{xintseries}{\numexpr not available, aborting input}%
   \aftergroup\endinput
\else
  \ifx\x\relax   % plain-TeX, first loading of xintseries.sty
    \ifx\w\relax % but xintfrac.sty not yet loaded.
       \y{xintseries}{now issuing \string\input\space xintfrac.sty}%
       \def\z{\endgroup\input xintfrac.sty\relax}%
    \fi
  \else
    \def\empty {}%
    \ifx\x\empty % LaTeX, first loading,
    % variable is initialized, but \ProvidesPackage not yet seen
        \ifx\w\relax % xintfrac.sty not yet loaded.
          \y{xintseries}{now issuing \string\RequirePackage{xintfrac}}%
          \def\z{\endgroup\RequirePackage{xintfrac}}%
        \fi
    \else
      \y{xintseries}{I was already loaded, aborting input}%
      \aftergroup\endinput
    \fi
  \fi
\fi
\z%
```

36.2 Confirmation of xintfrac loading

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
\catcode13=5    % ^^M
\endlinechar=13 %
\catcode123=1   % {
\catcode125=2   % }
\catcode64=11   % @
\catcode35=6    % #
```

```
\catcode44=12   % ,
\catcode45=12   % -
\catcode46=12   % .
\catcode58=12   % :
\ifdefined\PackageInfo
    \def\y#1#2{\PackageInfo{#1}{#2}}%
  \else
    \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
\fi
\def\empty {}%
\expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
\ifx\w\relax % Plain TeX, user gave a file name at the prompt
    \y{xintseries}{Loading of package xintfrac failed, aborting input}%
    \aftergroup\endinput
\fi
\ifx\w\empty % LaTeX, user gave a file name at the prompt
    \y{xintseries}{Loading of package xintfrac failed, aborting input}%
    \aftergroup\endinput
\fi
\endgroup%
```

36.3 Catcodes

```
\XINTsetupcatcodes%
```

36.4 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintseries}%
  [2014/02/13 v1.09kb Expandable partial sums with xint package (jfB)]%
```

36.5 \xintSeries

Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```
\def\xintSeries {\romannumeral0\xintseries }%
\def\xintseries #1#2%
{%
  \expandafter\XINT_series\expandafter
  {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_series #1#2#3%
{%
  \ifnum #2<#1
    \xint_afterfi { 0/1[0]}%
  \else
    \xint_afterfi {\XINT_series_loop {#1}{0}{#2}{#3}}%
  \fi
}%
```

```
\def\XINT_series_loop #1#2#3#4%
{%
    \ifnum #3>#1 \else \XINT_series_exit \fi
    \expandafter\XINT_series_loop\expandafter
    {\the\numexpr #1+1\expandafter }\expandafter
    {\romannumeral0\xintadd {#2}{#4{#1}}}%
    {#3}{#4}%
}%
\def\XINT_series_exit \fi #1#2#3#4#5#6#7#8%
{%
    \fi\xint_gobble_ii #6%
}%
```

36.6 \xintiSeries

Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```
\def\xintiSeries {\romannumeral0\xintiseries }%
\def\xintiseries #1#2%
{%
    \expandafter\XINT_iseries\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_iseries #1#2#3%
{%
   \ifnum #2<#1
      \xint_afterfi { 0}%
   \else
      \xint_afterfi {\XINT_iseries_loop {#1}{0}{#2}{#3}}%
   \fi
}%
\def\XINT_iseries_loop #1#2#3#4%
{%
    \ifnum #3>#1 \else \XINT_iseries_exit \fi
    \expandafter\XINT_iseries_loop\expandafter
    {\the\numexpr #1+1\expandafter }\expandafter
    {\romannumeral0\xintiiadd {#2}{#4{#1}}}%
    {#3}{#4}%
}%
\def\XINT_iseries_exit \fi #1#2#3#4#5#6#7#8%
{%
    \fi\xint_gobble_ii #6%
}%
```

36.7 \xintPowerSeries

The 1.03 version was very lame and created a build-up of denominators. The Horner scheme for polynomial evaluation is used in 1.04, this cures the denominator problem and drastically improves the efficiency of the macro. Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```
\def\xintPowerSeries {\romannumeral0\xintpowerseries }%
\def\xintpowerseries #1#2%
{%
    \expandafter\XINT_powseries\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_powseries #1#2#3#4%
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi
      {\XINT_powseries_loop_i {#3{#2}}{#1}{#2}{#3}{#4}}%
   \fi
}%
\def\XINT_powseries_loop_i #1#2#3#4#5%
{%
    \ifnum #3>#2 \else\XINT_powseries_exit_i\fi
    \expandafter\XINT_powseries_loop_ii\expandafter
    {\the\numexpr #3-1\expandafter}\expandafter
    {\romannumeral0\xintmul {#1}{#5}}{#2}{#4}{#5}%
}%
\def\XINT_powseries_loop_ii #1#2#3#4%
{%
   \expandafter\XINT_powseries_loop_i\expandafter
   {\romannumeral0\xintadd {#4{#1}}{#2}}{#3}{#1}{#4}%
}%
\def\XINT_powseries_exit_i\fi #1#2#3#4#5#6#7#8#9%
{%
    \fi \XINT_powseries_exit_ii  #6{#7}%
}%
\def\XINT_powseries_exit_ii #1#2#3#4#5#6%
{%
    \xintmul{\xintPow {#5}{#6}}{#4}%
}%
```

36.8 \xintPowerSeriesX

Same as \xintPowerSeries except for the initial expansion of the x parameter. Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```
\def\xintPowerSeriesX {\romannumeral0\xintpowerseriesx }%
\def\xintpowerseriesx #1#2%
{%
    \expandafter\XINT_powseriesx\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_powseriesx #1#2#3#4%
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi
      {\expandafter\XINT_powseriesx_pre\expandafter
                  {\romannumeral-`0#4}{#1}{#2}{#3}%
      }%
   \fi
}%
\def\XINT_powseriesx_pre #1#2#3#4%
{%
    \XINT_powseries_loop_i {#4{#3}}{#2}{#3}{#4}{#1}%
}%
```

36.9 \xintRationalSeries

This computes F(a)+...+F(b) on the basis of the value of F(a) and the ratios F(n)/F(n-1). As in \xintPowerSeries we use an iterative scheme which has the great advantage to avoid denominator build-up. This makes exact computations possible with exponential type series, which would be completely inaccessible to \xintSeries. #1=a, #2=b, #3=F(a), #4=ratio function Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```
\def\xintRationalSeries {\romannumeral0\xintratseries }%
\def\xintratseries #1#2%
{%
    \expandafter\XINT_ratseries\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_ratseries #1#2#3#4%
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi
      {\XINT_ratseries_loop {#2}{1}{#1}{#4}{#3}}%
```

```
   \fi
}%
\def\XINT_ratseries_loop #1#2#3#4%
{%
    \ifnum #1>#3 \else\XINT_ratseries_exit_i\fi
    \expandafter\XINT_ratseries_loop\expandafter
    {\the\numexpr #1-1\expandafter}\expandafter
    {\romannumeral0\xintadd {1}{\xintMul {#2}{#4{#1}}}}{#3}{#4}%
}%
\def\XINT_ratseries_exit_i\fi #1#2#3#4#5#6#7#8%
{%
    \fi \XINT_ratseries_exit_ii  #6%
}%
\def\XINT_ratseries_exit_ii #1#2#3#4#5%
{%
    \XINT_ratseries_exit_iii #5%
}%
\def\XINT_ratseries_exit_iii #1#2#3#4%
{%
    \xintmul{#2}{#4}%
}%
```

36.10 \xintRationalSeriesX

a,b,initial,ratiofunction,x

This computes F(a,x)+...+F(b,x) on the basis of the value of F(a,x) and the ratios F(n,x)/F(n-1,x). The argument x is first expanded and it is the value resulting from this which is used then throughout. The initial term F(a,x) must be defined as one-parameter macro which will be given x. Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```
\def\xintRationalSeriesX {\romannumeral0\xintratseriesx }%
\def\xintratseriesx #1#2%
{%
    \expandafter\XINT_ratseriesx\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_ratseriesx #1#2#3#4#5%
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi
      {\expandafter\XINT_ratseriesx_pre\expandafter
                     {\romannumeral-`0#5}{#2}{#1}{#4}{#3}%
      }%
   \fi
```

```
}%
\def\XINT_ratseriesx_pre #1#2#3#4#5%
{%
    \XINT_ratseries_loop {#2}{1}{#3}{#4{#1}}{#5{#1}}%
}%
```

36.11 \xintFxPtPowerSeries

I am not two happy with this piece of code. Will make it more economical another day. Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a: forgot last time some optimization from the change to \numexpr.

```
\def\xintFxPtPowerSeries {\romannumeral0\xintfxptpowerseries }%
\def\xintfxptpowerseries #1#2%
{%
    \expandafter\XINT_fppowseries\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_fppowseries #1#2#3#4#5%
{%
   \ifnum #2<#1
      \xint_afterfi { 0}%
   \else
      \xint_afterfi
        {\expandafter\XINT_fppowseries_loop_pre\expandafter
           {\romannumeral0\xinttrunc {#5}{\xintPow {#4}{#1}}}%
          {#1}{#4}{#2}{#3}{#5}%
        }%
   \fi
}%
\def\XINT_fppowseries_loop_pre #1#2#3#4#5#6%
{%
    \ifnum #4>#2 \else\XINT_fppowseries_dont_i \fi
    \expandafter\XINT_fppowseries_loop_i\expandafter
    {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
    {\romannumeral0\xintitrunc {#6}{\xintMul {#5{#2}}{#1}}}%
    {#1}{#3}{#4}{#5}{#6}%
}%
\def\XINT_fppowseries_dont_i \fi\expandafter\XINT_fppowseries_loop_i
    {\fi \expandafter\XINT_fppowseries_dont_ii }%
\def\XINT_fppowseries_dont_ii #1#2#3#4#5#6#7{\xinttrunc {#7}{#2[-#7]}}%
\def\XINT_fppowseries_loop_i #1#2#3#4#5#6#7%
{%
    \ifnum #5>#1 \else \XINT_fppowseries_exit_i \fi
    \expandafter\XINT_fppowseries_loop_ii\expandafter
    {\romannumeral0\xinttrunc {#7}{\xintMul {#3}{#4}}}%
    {#1}{#4}{#2}{#5}{#6}{#7}%
}%
```

```
\def\XINT_fppowerseries_loop_ii #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_fppowerseries_loop_i\expandafter
    {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
    {\romannumeral0\xintiiadd {#4}{\xintiTrunc {#7}{\xintMul {#6{#2}}{#1}}}}%
    {#1}{#3}{#5}{#6}{#7}%
}%
\def\XINT_fppowerseries_exit_i\fi\expandafter\XINT_fppowerseries_loop_ii
    {\fi \expandafter\XINT_fppowerseries_exit_ii }%
\def\XINT_fppowerseries_exit_ii #1#2#3#4#5#6#7%
{%
    \xinttrunc {#7}
    {\xintiiadd {#4}{\xintiTrunc {#7}{\xintMul {#6{#2}}{#1}}}[-#7]}%
}%
```

36.12 \xintFxPtPowerSeriesX

a,b,coeff,x,D
Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```
\def\xintFxPtPowerSeriesX {\romannumeral0\xintfxptpowerseriesx }%
\def\xintfxptpowerseriesx #1#2%
{%
    \expandafter\XINT_fppowerseriesx\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_fppowerseriesx #1#2#3#4#5%
{%
   \ifnum #2<#1
      \xint_afterfi { 0}%
   \else
      \xint_afterfi
        {\expandafter \XINT_fppowerseriesx_pre \expandafter
         {\romannumeral-`0#4}{#1}{#2}{#3}{#5}%
        }%
   \fi
}%
\def\XINT_fppowerseriesx_pre #1#2#3#4#5%
{%
    \expandafter\XINT_fppowerseries_loop_pre\expandafter
       {\romannumeral0\xinttrunc {#5}{\xintPow {#1}{#2}}}%
       {#2}{#1}{#3}{#4}{#5}%
}%
```

36.13 \xintFloatPowerSeries

1.08a. I still have to re-visit \xintFxPtPowerSeries; temporarily I just adapted the code to the case of floats.

```
\def\xintFloatPowerSeries {\romannumeral0\xintfloatpowerseries }%
\def\xintfloatpowerseries #1{\XINT_flpowseries_chkopt #1\Z }%
\def\XINT_flpowseries_chkopt #1%
{%
    \ifx [#1\expandafter\XINT_flpowseries_opt
       \else\expandafter\XINT_flpowseries_noopt
    \fi
    #1%
}%
\def\XINT_flpowseries_noopt  #1\Z #2%
{%
    \expandafter\XINT_flpowseries\expandafter
    {\the\numexpr #1\expandafter}\expandafter
    {\the\numexpr #2}\XINTdigits
}%
\def\XINT_flpowseries_opt [\Z #1]#2#3%
{%
    \expandafter\XINT_flpowseries\expandafter
    {\the\numexpr #2\expandafter}\expandafter
    {\the\numexpr #3\expandafter}{\the\numexpr #1}%
}%
\def\XINT_flpowseries #1#2#3#4#5%
{%
   \ifnum #2<#1
      \xint_afterfi { 0.e0}%
   \else
      \xint_afterfi
        {\expandafter\XINT_flpowseries_loop_pre\expandafter
           {\romannumeral0\XINTinfloatpow [#3]{#5}{#1}}%
          {#1}{#5}{#2}{#4}{#3}%
        }%
   \fi
}%
\def\XINT_flpowseries_loop_pre #1#2#3#4#5#6%
{%
    \ifnum #4>#2 \else\XINT_flpowseries_dont_i \fi
    \expandafter\XINT_flpowseries_loop_i\expandafter
    {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
    {\romannumeral0\XINTinfloatmul [#6]{#5{#2}}{#1}}%
    {#1}{#3}{#4}{#5}{#6}%
}%
\def\XINT_flpowseries_dont_i \fi\expandafter\XINT_flpowseries_loop_i
    {\fi \expandafter\XINT_flpowseries_dont_ii }%
\def\XINT_flpowseries_dont_ii #1#2#3#4#5#6#7{\xintfloat [#7]{#2}}%
```

```
\def\XINT_flpowseries_loop_i #1#2#3#4#5#6#7%
{%
    \ifnum #5>#1 \else \XINT_flpowseries_exit_i \fi
    \expandafter\XINT_flpowseries_loop_ii\expandafter
    {\romannumeral0\XINTinfloatmul [#7]{#3}{#4}}%
    {#1}{#4}{#2}{#5}{#6}{#7}%
}%
\def\XINT_flpowseries_loop_ii #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_flpowseries_loop_i\expandafter
    {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
    {\romannumeral0\XINTinfloatadd [#7]{#4}%
                            {\XINTinfloatmul [#7]{#6{#2}}{#1}}}%
    {#1}{#3}{#5}{#6}{#7}%
}%
\def\XINT_flpowseries_exit_i\fi\expandafter\XINT_flpowseries_loop_ii
    {\fi \expandafter\XINT_flpowseries_exit_ii }%
\def\XINT_flpowseries_exit_ii #1#2#3#4#5#6#7%
{%
    \xintfloatadd [#7]{#4}{\XINTinfloatmul [#7]{#6{#2}}{#1}}%
}%
```

36.14 \xintFloatPowerSeriesX

1.08a

```
\def\xintFloatPowerSeriesX {\romannumeral0\xintfloatpowerseriesx }%
\def\xintfloatpowerseriesx #1{\XINT_flpowseriesx_chkopt #1\Z }%
\def\XINT_flpowseriesx_chkopt #1%
{%
    \ifx [#1\expandafter\XINT_flpowseriesx_opt
       \else\expandafter\XINT_flpowseriesx_noopt
    \fi
    #1%
}%
\def\XINT_flpowseriesx_noopt  #1\Z #2%
{%
    \expandafter\XINT_flpowseriesx\expandafter
    {\the\numexpr #1\expandafter}\expandafter
    {\the\numexpr #2}\XINTdigits
}%
\def\XINT_flpowseriesx_opt [\Z #1]#2#3%
{%
    \expandafter\XINT_flpowseriesx\expandafter
    {\the\numexpr #2\expandafter}\expandafter
    {\the\numexpr #3\expandafter}{\the\numexpr #1}%
}%
\def\XINT_flpowseriesx #1#2#3#4#5%
{%
```

```
    \ifnum #2<#1
       \xint_afterfi { 0.e0}%
    \else
       \xint_afterfi
         {\expandafter \XINT_flpowseriesx_pre \expandafter
          {\romannumeral-`0#5}{#1}{#2}{#4}{#3}%
         }%
    \fi
}%
\def\XINT_flpowseriesx_pre #1#2#3#4#5%
{%
    \expandafter\XINT_flpowseries_loop_pre\expandafter
       {\romannumeral0\XINTinfloatpow [#5]{#1}{#2}}%
       {#2}{#1}{#3}{#4}{#5}%
}%
\XINT_restorecatcodes_endinput%
```

37 Package xintcfrac implementation

The commenting is currently (2014/02/13) very sparse.

Contents

37.1 Catcodes, ε-TEX and reload detection

The code for reload detection is copied from Heiko Oberdiek's packages, and adapted here to check for previous loading of the xintfrac package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
```

```
\catcode13=5    % ^^M
\endlinechar=13 %
\catcode123=1   % {
\catcode125=2   % }
\catcode64=11   % @
\catcode35=6    % #
\catcode44=12   % ,
\catcode45=12   % -
\catcode46=12   % .
\catcode58=12   % :
\def\space { }%
\let\z\endgroup
\expandafter\let\expandafter\x\csname ver@xintcfrac.sty\endcsname
\expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
\expandafter
  \ifx\csname PackageInfo\endcsname\relax
    \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
  \else
    \def\y#1#2{\PackageInfo{#1}{#2}}%
  \fi
\expandafter
\ifx\csname numexpr\endcsname\relax
   \y{xintcfrac}{\numexpr not available, aborting input}%
   \aftergroup\endinput
\else
  \ifx\x\relax   % plain-TeX, first loading of xintcfrac.sty
    \ifx\w\relax % but xintfrac.sty not yet loaded.
       \y{xintcfrac}{now issuing \string\input\space xintfrac.sty}%
       \def\z{\endgroup\input xintfrac.sty\relax}%
    \fi
  \else
    \def\empty {}%
    \ifx\x\empty % LaTeX, first loading,
    % variable is initialized, but \ProvidesPackage not yet seen
        \ifx\w\relax % xintfrac.sty not yet loaded.
          \y{xintcfrac}{now issuing \string\RequirePackage{xintfrac}}%
          \def\z{\endgroup\RequirePackage{xintfrac}}%
        \fi
    \else
      \y{xintcfrac}{I was already loaded, aborting input}%
      \aftergroup\endinput
    \fi
  \fi
\fi
\z%
```

37.2 Confirmation of xintfrac loading

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
```

```
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \ifdefined\PackageInfo
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \else
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
  \fi
  \def\empty {}%
  \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
  \ifx\w\relax % Plain TeX, user gave a file name at the prompt
      \y{xintcfrac}{Loading of package xintfrac failed, aborting input}%
      \aftergroup\endinput
  \fi
  \ifx\w\empty % LaTeX, user gave a file name at the prompt
      \y{xintcfrac}{Loading of package xintfrac failed, aborting input}%
      \aftergroup\endinput
  \fi
\endgroup%
```

37.3 Catcodes

```
\XINTsetupcatcodes%
```

37.4 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintcfrac}%
  [2014/02/13 v1.09kb Expandable continued fractions with xint package (jfB)]%
```

37.5 \xintCFrac

```
\def\xintCFrac {\romannumeral0\xintcfrac }%
\def\xintcfrac #1%
{%
    \XINT_cfrac_opt_a #1\Z
}%
\def\XINT_cfrac_opt_a #1%
{%
    \ifx[#1\XINT_cfrac_opt_b\fi \XINT_cfrac_noopt #1%
}%
\def\XINT_cfrac_noopt #1\Z
{%
    \expandafter\XINT_cfrac_A\romannumeral0\xintrawwithzeros {#1}\Z
```

```
    \relax\relax
}%
\def\XINT_cfrac_opt_b\fi\XINT_cfrac_noopt [\Z #1]%
{%
    \fi\csname XINT_cfrac_opt#1\endcsname
}%
\def\XINT_cfrac_optl #1%
{%
    \expandafter\XINT_cfrac_A\romannumeral0\xintrawwithzeros {#1}\Z
    \relax\hfill
}%
\def\XINT_cfrac_optc #1%
{%
    \expandafter\XINT_cfrac_A\romannumeral0\xintrawwithzeros {#1}\Z
    \relax\relax
}%
\def\XINT_cfrac_optr #1%
{%
    \expandafter\XINT_cfrac_A\romannumeral0\xintrawwithzeros {#1}\Z
    \hfill\relax
}%
\def\XINT_cfrac_A #1/#2\Z
{%
    \expandafter\XINT_cfrac_B\romannumeral0\xintiidivision {#1}{#2}{#2}%
}%
\def\XINT_cfrac_B #1#2%
{%
    \XINT_cfrac_C #2\Z {#1}%
}%
\def\XINT_cfrac_C #1%
{%
    \xint_gob_til_zero #1\XINT_cfrac_integer 0\XINT_cfrac_D #1%
}%
\def\XINT_cfrac_integer 0\XINT_cfrac_D 0#1\Z #2#3#4#5{ #2}%
\def\XINT_cfrac_D #1\Z #2#3{\XINT_cfrac_loop_a {#1}{#3}{#1}{{#2}}}%
\def\XINT_cfrac_loop_a
{%
    \expandafter\XINT_cfrac_loop_d\romannumeral0\XINT_div_prepare
}%
\def\XINT_cfrac_loop_d #1#2%
{%
    \XINT_cfrac_loop_e #2.{#1}%
}%
\def\XINT_cfrac_loop_e #1%
{%
    \xint_gob_til_zero #1\xint_cfrac_loop_exit0\XINT_cfrac_loop_f #1%
}%
\def\XINT_cfrac_loop_f #1.#2#3#4%
{%
```

```
    \XINT_cfrac_loop_a {#1}{#3}{#1}{{#2}#4}%
}%
\def\xint_cfrac_loop_exit0\XINT_cfrac_loop_f #1.#2#3#4#5#6%
   {\XINT_cfrac_T #5#6{#2}#4\Z }%
\def\XINT_cfrac_T #1#2#3#4%
{%
  \xint_gob_til_Z #4\XINT_cfrac_end\Z\XINT_cfrac_T #1#2{#4+\cfrac{#11#2}{#3}}%
}%
\def\XINT_cfrac_end\Z\XINT_cfrac_T #1#2#3%
{%
    \XINT_cfrac_end_b #3%
}%
\def\XINT_cfrac_end_b \Z+\cfrac#1#2{ #2}%
```

37.6 \xintGCFrac

```
\def\xintGCFrac {\romannumeral0\xintgcfrac }%
\def\xintgcfrac #1{\XINT_gcfrac_opt_a #1\Z }%
\def\XINT_gcfrac_opt_a #1%
{%
    \ifx[#1\XINT_gcfrac_opt_b\fi \XINT_gcfrac_noopt #1%
}%
\def\XINT_gcfrac_noopt #1\Z
{%
    \XINT_gcfrac #1+\W/\relax\relax
}%
\def\XINT_gcfrac_opt_b\fi\XINT_gcfrac_noopt [\Z #1]%
{%
    \fi\csname XINT_gcfrac_opt#1\endcsname
}%
\def\XINT_gcfrac_optl #1%
{%
    \XINT_gcfrac #1+\W/\relax\hfill
}%
\def\XINT_gcfrac_optc #1%
{%
    \XINT_gcfrac #1+\W/\relax\relax
}%
\def\XINT_gcfrac_optr #1%
{%
    \XINT_gcfrac #1+\W/\hfill\relax
}%
\def\XINT_gcfrac
{%
    \expandafter\XINT_gcfrac_enter\romannumeral-`0%
}%
\def\XINT_gcfrac_enter {\XINT_gcfrac_loop {}}%
\def\XINT_gcfrac_loop #1#2+#3/%
{%
    \xint_gob_til_W #3\XINT_gcfrac_endloop\W
```

```
    \XINT_gcfrac_loop {{#3}{#2}#1}%
}%
\def\XINT_gcfrac_endloop\W\XINT_gcfrac_loop #1#2#3%
{%
    \XINT_gcfrac_T #2#3#1\Z\Z
}%
\def\XINT_gcfrac_T #1#2#3#4{\XINT_gcfrac_U #1#2{\xintFrac{#4}}}%
\def\XINT_gcfrac_U #1#2#3#4#5%
{%
    \xint_gob_til_Z #5\XINT_gcfrac_end\Z\XINT_gcfrac_U
              #1#2{\xintFrac{#5}%
               \ifcase\xintSgn{#4}
               +\or+\else-\fi
               \cfrac{#1\xintFrac{\xintAbs{#4}}#2}{#3}}%
}%
\def\XINT_gcfrac_end\Z\XINT_gcfrac_U #1#2#3%
{%
    \XINT_gcfrac_end_b #3%
}%
\def\XINT_gcfrac_end_b #1\cfrac#2#3{ #3}%
```

37.7 \xintGCtoGCx

```
\def\xintGCtoGCx {\romannumeral0\xintgctogcx }%
\def\xintgctogcx #1#2#3%
{%
    \expandafter\XINT_gctgcx_start\expandafter {\romannumeral-`0#3}{#1}{#2}%
}%
\def\XINT_gctgcx_start #1#2#3{\XINT_gctgcx_loop_a {}{#2}{#3}#1+\W/}%
\def\XINT_gctgcx_loop_a #1#2#3#4+#5/%
{%
    \xint_gob_til_W #5\XINT_gctgcx_end\W
    \XINT_gctgcx_loop_b {#1{#4}}{#2{#5}#3}{#2}{#3}%
}%
\def\XINT_gctgcx_loop_b #1#2%
{%
    \XINT_gctgcx_loop_a {#1#2}%
}%
\def\XINT_gctgcx_end\W\XINT_gctgcx_loop_b #1#2#3#4{ #1}%
```

37.8 \xintFtoCs

```
\def\xintFtoCs {\romannumeral0\xintftocs }%
\def\xintftocs #1%
{%
    \expandafter\XINT_ftc_A\romannumeral0\xintrawwithzeros {#1}\Z
}%
\def\XINT_ftc_A #1/#2\Z
{%
    \expandafter\XINT_ftc_B\romannumeral0\xintiidivision {#1}{#2}{#2}%
}%
```

```
\def\XINT_ftc_B #1#2%
{%
    \XINT_ftc_C #2.{#1}%
}%
\def\XINT_ftc_C #1%
{%
    \xint_gob_til_zero #1\XINT_ftc_integer 0\XINT_ftc_D #1%
}%
\def\XINT_ftc_integer 0\XINT_ftc_D 0#1.#2#3{ #2}%
\def\XINT_ftc_D #1.#2#3{\XINT_ftc_loop_a {#1}{#3}{#1}{#2,}}%
\def\XINT_ftc_loop_a
{%
    \expandafter\XINT_ftc_loop_d\romannumeral0\XINT_div_prepare
}%
\def\XINT_ftc_loop_d #1#2%
{%
    \XINT_ftc_loop_e #2.{#1}%
}%
\def\XINT_ftc_loop_e #1%
{%
    \xint_gob_til_zero #1\xint_ftc_loop_exit0\XINT_ftc_loop_f #1%
}%
\def\XINT_ftc_loop_f #1.#2#3#4%
{%
    \XINT_ftc_loop_a {#1}{#3}{#1}{#4#2,}%
}%
\def\xint_ftc_loop_exit0\XINT_ftc_loop_f #1.#2#3#4{ #4#2}%
```

37.9 \xintFtoCx

```
\def\xintFtoCx {\romannumeral0\xintftocx }%
\def\xintftocx #1#2%
{%
    \expandafter\XINT_ftcx_A\romannumeral0\xintrawwithzeros {#2}\Z {#1}%
}%
\def\XINT_ftcx_A #1/#2\Z
{%
    \expandafter\XINT_ftcx_B\romannumeral0\xintiidivision {#1}{#2}{#2}%
}%
\def\XINT_ftcx_B #1#2%
{%
    \XINT_ftcx_C #2.{#1}%
}%
\def\XINT_ftcx_C #1%
{%
    \xint_gob_til_zero #1\XINT_ftcx_integer 0\XINT_ftcx_D #1%
}%
\def\XINT_ftcx_integer 0\XINT_ftcx_D 0#1.#2#3#4{ #2}%
\def\XINT_ftcx_D #1.#2#3#4{\XINT_ftcx_loop_a {#1}{#3}{#1}{#2#4}{#4}}%
\def\XINT_ftcx_loop_a
```

```
{%
    \expandafter\XINT_ftcx_loop_d\romannumeral0\XINT_div_prepare
}%
\def\XINT_ftcx_loop_d #1#2%
{%
    \XINT_ftcx_loop_e #2.{#1}%
}%
\def\XINT_ftcx_loop_e #1%
{%
    \xint_gob_til_zero #1\xint_ftcx_loop_exit0\XINT_ftcx_loop_f #1%
}%
\def\XINT_ftcx_loop_f #1.#2#3#4#5%
{%
    \XINT_ftcx_loop_a {#1}{#3}{#1}{#4{#2}#5}{#5}%
}%
\def\xint_ftcx_loop_exit0\XINT_ftcx_loop_f #1.#2#3#4#5{ #4{#2}}%
```

37.10 \xintFtoGC

```
\def\xintFtoGC {\romannumeral0\xintftogc }%
\def\xintftogc {\xintftocx {+1/}}%
```

37.11 \xintFtoCC

```
\def\xintFtoCC {\romannumeral0\xintftocc }%
\def\xintftocc #1%
{%
    \expandafter\XINT_ftcc_A\expandafter {\romannumeral0\xintrawwithzeros {#1}}%
}%
\def\XINT_ftcc_A #1%
{%
    \expandafter\XINT_ftcc_B
    \romannumeral0\xintrawwithzeros {\xintAdd {1/2[0]}{#1[0]}}\Z {#1[0]}%
}%
\def\XINT_ftcc_B #1/#2\Z
{%
    \expandafter\XINT_ftcc_C\expandafter {\romannumeral0\xintiiquo {#1}{#2}}%
}%
\def\XINT_ftcc_C #1#2%
{%
    \expandafter\XINT_ftcc_D\romannumeral0\xintsub {#2}{#1}\Z {#1}%
}%
\def\XINT_ftcc_D #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_ftcc_integer
      0#1\XINT_ftcc_En
       0-{\XINT_ftcc_Ep #1}%
    \krof
}%
\def\XINT_ftcc_Ep #1\Z #2%
```

```
{%
    \expandafter\XINT_ftcc_loop_a\expandafter
    {\romannumeral0\xintdiv {1[0]}{#1}}{#2+1/}%
}%
\def\XINT_ftcc_En #1\Z #2%
{%
    \expandafter\XINT_ftcc_loop_a\expandafter
    {\romannumeral0\xintdiv {1[0]}{#1}}{#2+-1/}%
}%
\def\XINT_ftcc_integer #1\Z #2{ #2}%
\def\XINT_ftcc_loop_a #1%
{%
    \expandafter\XINT_ftcc_loop_b
    \romannumeral0\xintrawwithzeros {\xintAdd {1/2[0]}{#1}}\Z {#1}%
}%
\def\XINT_ftcc_loop_b #1/#2\Z
{%
    \expandafter\XINT_ftcc_loop_c\expandafter
    {\romannumeral0\xintiiquo {#1}{#2}}%
}%
\def\XINT_ftcc_loop_c #1#2%
{%
    \expandafter\XINT_ftcc_loop_d
    \romannumeral0\xintsub {#2}{#1[0]}\Z {#1}%
}%
\def\XINT_ftcc_loop_d #1%
{%
    \xint_UDzerominusfork
      #1-\XINT_ftcc_end
      0#1\XINT_ftcc_loop_N
       0-{\XINT_ftcc_loop_P #1}%
    \krof
}%
\def\XINT_ftcc_end #1\Z #2#3{ #3#2}%
\def\XINT_ftcc_loop_P #1\Z #2#3%
{%
    \expandafter\XINT_ftcc_loop_a\expandafter
    {\romannumeral0\xintdiv {1[0]}{#1}}{#3#2+1/}%
}%
\def\XINT_ftcc_loop_N #1\Z #2#3%
{%
    \expandafter\XINT_ftcc_loop_a\expandafter
    {\romannumeral0\xintdiv {1[0]}{#1}}{#3#2+-1/}%
}%
```

37.12 \xintFtoCv

```
\def\xintFtoCv {\romannumeral0\xintftocv }%
\def\xintftocv #1%
{%
```

```
    \xinticstocv {\xintFtoCs {#1}}%
}%
```

37.13 \xintFtoCCv

```
\def\xintFtoCCv {\romannumeral0\xintftoccv }%
\def\xintftoccv #1%
{%
    \xintigctocv {\xintFtoCC {#1}}%
}%
```

37.14 \xintCstoF

```
\def\xintCstoF {\romannumeral0\xintcstof }%
\def\xintcstof #1%
{%
    \expandafter\XINT_cstf_prep \romannumeral-`0#1,\W,%
}%
\def\XINT_cstf_prep
{%
    \XINT_cstf_loop_a 1001%
}%
\def\XINT_cstf_loop_a #1#2#3#4#5,%
{%
    \xint_gob_til_W #5\XINT_cstf_end\W
    \expandafter\XINT_cstf_loop_b
    \romannumeral0\xintrawwithzeros {#5}.{#1}{#2}{#3}{#4}%
}%
\def\XINT_cstf_loop_b #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_cstf_loop_c\expandafter
    {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
    {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
    {\romannumeral0\xintiiadd {\XINT_Mul {#2}{#6}}{\XINT_Mul {#1}{#4}}}%
    {\romannumeral0\xintiiadd {\XINT_Mul {#2}{#5}}{\XINT_Mul {#1}{#3}}}%
}%
\def\XINT_cstf_loop_c #1#2%
{%
    \expandafter\XINT_cstf_loop_d\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_cstf_loop_d #1#2%
{%
    \expandafter\XINT_cstf_loop_e\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_cstf_loop_e #1#2%
{%
    \expandafter\XINT_cstf_loop_a\expandafter{#2}#1%
}%
\def\XINT_cstf_end #1.#2#3#4#5{\xintrawwithzeros {#2/#3}}% 1.09b removes [0]
```

37.15 \xintiCstoF

```
\def\xintiCstoF {\romannumeral0\xinticstof }%
\def\xinticstof #1%
{%
    \expandafter\XINT_icstf_prep \romannumeral-`0#1,\W,%
}%
\def\XINT_icstf_prep
{%
    \XINT_icstf_loop_a 1001%
}%
\def\XINT_icstf_loop_a #1#2#3#4#5,%
{%
    \xint_gob_til_W #5\XINT_icstf_end\W
    \expandafter
    \XINT_icstf_loop_b \romannumeral-`0#5.{#1}{#2}{#3}{#4}%
}%
\def\XINT_icstf_loop_b #1.#2#3#4#5%
{%
    \expandafter\XINT_icstf_loop_c\expandafter
    {\romannumeral0\xintiiadd {#5}{\XINT_Mul {#1}{#3}}}%
    {\romannumeral0\xintiiadd {#4}{\XINT_Mul {#1}{#2}}}%
    {#2}{#3}%
}%
\def\XINT_icstf_loop_c #1#2%
{%
    \expandafter\XINT_icstf_loop_a\expandafter {#2}{#1}%
}%
\def\XINT_icstf_end#1.#2#3#4#5{\xintrawwithzeros {#2/#3}}% 1.09b removes [0]
```

37.16 \xintGCtoF

```
\def\xintGCtoF {\romannumeral0\xintgctof }%
\def\xintgctof #1%
{%
    \expandafter\XINT_gctf_prep \romannumeral-`0#1+\W/%
}%
\def\XINT_gctf_prep
{%
    \XINT_gctf_loop_a 1001%
}%
\def\XINT_gctf_loop_a #1#2#3#4#5+%
{%
    \expandafter\XINT_gctf_loop_b
    \romannumeral0\xintrawwithzeros {#5}.{#1}{#2}{#3}{#4}%
}%
\def\XINT_gctf_loop_b #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_gctf_loop_c\expandafter
    {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
```

```
    {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
    {\romannumeral0\xintiiadd {\XINT_Mul {#2}{#6}}{\XINT_Mul {#1}{#4}}}%
    {\romannumeral0\xintiiadd {\XINT_Mul {#2}{#5}}{\XINT_Mul {#1}{#3}}}%
}%
\def\XINT_gctf_loop_c #1#2%
{%
    \expandafter\XINT_gctf_loop_d\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_gctf_loop_d #1#2%
{%
    \expandafter\XINT_gctf_loop_e\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_gctf_loop_e #1#2%
{%
    \expandafter\XINT_gctf_loop_f\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_gctf_loop_f #1#2/%
{%
    \xint_gob_til_W #2\XINT_gctf_end\W
    \expandafter\XINT_gctf_loop_g
    \romannumeral0\xintrawwithzeros {#2}.#1%
}%
\def\XINT_gctf_loop_g #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_gctf_loop_h\expandafter
    {\romannumeral0\XINT_mul_fork #1\Z #6\Z }%
    {\romannumeral0\XINT_mul_fork #1\Z #5\Z }%
    {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
    {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
}%
\def\XINT_gctf_loop_h #1#2%
{%
    \expandafter\XINT_gctf_loop_i\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_gctf_loop_i #1#2%
{%
    \expandafter\XINT_gctf_loop_j\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_gctf_loop_j #1#2%
{%
    \expandafter\XINT_gctf_loop_a\expandafter {#2}#1%
}%
\def\XINT_gctf_end #1.#2#3#4#5{\xintrawwithzeros {#2/#3}}% 1.09b removes [0]
```

37.17 \xintiGCtoF

```
\def\xintiGCtoF {\romannumeral0\xintigctof }%
\def\xintigctof #1%
{%
    \expandafter\XINT_igctf_prep \romannumeral-`0#1+\W/%
```

```
}%
\def\XINT_igctf_prep
{%
    \XINT_igctf_loop_a 1001%
}%
\def\XINT_igctf_loop_a #1#2#3#4#5+%
{%
    \expandafter\XINT_igctf_loop_b
    \romannumeral-`0#5.{#1}{#2}{#3}{#4}%
}%
\def\XINT_igctf_loop_b #1.#2#3#4#5%
{%
    \expandafter\XINT_igctf_loop_c\expandafter
    {\romannumeral0\xintiiadd {#5}{\XINT_Mul {#1}{#3}}}%
    {\romannumeral0\xintiiadd {#4}{\XINT_Mul {#1}{#2}}}%
    {#2}{#3}%
}%
\def\XINT_igctf_loop_c #1#2%
{%
    \expandafter\XINT_igctf_loop_f\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_igctf_loop_f #1#2#3#4/%
{%
    \xint_gob_til_W #4\XINT_igctf_end\W
    \expandafter\XINT_igctf_loop_g
    \romannumeral-`0#4.{#2}{#3}#1%
}%
\def\XINT_igctf_loop_g #1.#2#3%
{%
    \expandafter\XINT_igctf_loop_h\expandafter
    {\romannumeral0\XINT_mul_fork #1\Z #3\Z }%
    {\romannumeral0\XINT_mul_fork #1\Z #2\Z }%
}%
\def\XINT_igctf_loop_h #1#2%
{%
    \expandafter\XINT_igctf_loop_i\expandafter {#2}{#1}%
}%
\def\XINT_igctf_loop_i #1#2#3#4%
{%
    \XINT_igctf_loop_a {#3}{#4}{#1}{#2}%
}%
\def\XINT_igctf_end #1.#2#3#4#5{\xintrawwithzeros {#4/#5}}% 1.09b removes [0]
```

37.18 \xintCstoCv

```
\def\xintCstoCv {\romannumeral0\xintcstocv }%
\def\xintcstocv #1%
{%
    \expandafter\XINT_cstcv_prep \romannumeral-`0#1,\W,%
}%
```

```
\def\XINT_cstcv_prep
{%
    \XINT_cstcv_loop_a {}1001%
}%
\def\XINT_cstcv_loop_a #1#2#3#4#5#6,%
{%
    \xint_gob_til_W #6\XINT_cstcv_end\W
    \expandafter\XINT_cstcv_loop_b
    \romannumeral0\xintrawwithzeros {#6}.{#2}{#3}{#4}{#5}{#1}%
}%
\def\XINT_cstcv_loop_b #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_cstcv_loop_c\expandafter
    {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
    {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
    {\romannumeral0\xintiiadd {\XINT_Mul {#2}{#6}}{\XINT_Mul {#1}{#4}}}%
    {\romannumeral0\xintiiadd {\XINT_Mul {#2}{#5}}{\XINT_Mul {#1}{#3}}}%
}%
\def\XINT_cstcv_loop_c #1#2%
{%
    \expandafter\XINT_cstcv_loop_d\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_cstcv_loop_d #1#2%
{%
    \expandafter\XINT_cstcv_loop_e\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_cstcv_loop_e #1#2%
{%
    \expandafter\XINT_cstcv_loop_f\expandafter{#2}#1%
}%
\def\XINT_cstcv_loop_f #1#2#3#4#5%
{%
    \expandafter\XINT_cstcv_loop_g\expandafter
    {\romannumeral0\xintrawwithzeros {#1/#2}}{#5}{#1}{#2}{#3}{#4}%
}%
\def\XINT_cstcv_loop_g #1#2{\XINT_cstcv_loop_a {#2{#1}}}% 1.09b removes [0]
\def\XINT_cstcv_end #1.#2#3#4#5#6{ #6}%
```

37.19 \xintiCstoCv

```
\def\xintiCstoCv {\romannumeral0\xinticstocv }%
\def\xinticstocv #1%
{%
    \expandafter\XINT_icstcv_prep \romannumeral-`0#1,\W,%
}%
\def\XINT_icstcv_prep
{%
    \XINT_icstcv_loop_a {}1001%
}%
\def\XINT_icstcv_loop_a #1#2#3#4#5#6,%
```

```
{%
    \xint_gob_til_W #6\XINT_icstcv_end\W
    \expandafter
    \XINT_icstcv_loop_b \romannumeral-`0#6.{#2}{#3}{#4}{#5}{#1}%
}%
\def\XINT_icstcv_loop_b #1.#2#3#4#5%
{%
    \expandafter\XINT_icstcv_loop_c\expandafter
    {\romannumeral0\xintiiadd {#5}{\XINT_Mul {#1}{#3}}}%
    {\romannumeral0\xintiiadd {#4}{\XINT_Mul {#1}{#2}}}%
    {{#2}{#3}}%
}%
\def\XINT_icstcv_loop_c #1#2%
{%
    \expandafter\XINT_icstcv_loop_d\expandafter {#2}{#1}%
}%
\def\XINT_icstcv_loop_d #1#2%
{%
    \expandafter\XINT_icstcv_loop_e\expandafter
    {\romannumeral0\xintrawwithzeros {#1/#2}}{{#1}{#2}}%
}%
\def\XINT_icstcv_loop_e #1#2#3#4{\XINT_icstcv_loop_a {#4{#1}}#2#3}%
\def\XINT_icstcv_end #1.#2#3#4#5#6{ #6}%  1.09b removes [0]
```

37.20 \xintGCtoCv

```
\def\xintGCtoCv {\romannumeral0\xintgctocv }%
\def\xintgctocv #1%
{%
    \expandafter\XINT_gctcv_prep \romannumeral-`0#1+\W/%
}%
\def\XINT_gctcv_prep
{%
    \XINT_gctcv_loop_a {}1001%
}%
\def\XINT_gctcv_loop_a #1#2#3#4#5#6+%
{%
    \expandafter\XINT_gctcv_loop_b
    \romannumeral0\xintrawwithzeros {#6}.{#2}{#3}{#4}{#5}{#1}%
}%
\def\XINT_gctcv_loop_b #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_gctcv_loop_c\expandafter
    {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
    {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
    {\romannumeral0\xintiiadd {\XINT_Mul {#2}{#6}}{\XINT_Mul {#1}{#4}}}%
    {\romannumeral0\xintiiadd {\XINT_Mul {#2}{#5}}{\XINT_Mul {#1}{#3}}}%
}%
\def\XINT_gctcv_loop_c #1#2%
{%
```

```
    \expandafter\XINT_gctcv_loop_d\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_gctcv_loop_d #1#2%
{%
    \expandafter\XINT_gctcv_loop_e\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_gctcv_loop_e #1#2%
{%
    \expandafter\XINT_gctcv_loop_f\expandafter {#2}#1%
}%
\def\XINT_gctcv_loop_f #1#2%
{%
    \expandafter\XINT_gctcv_loop_g\expandafter
    {\romannumeral0\xintrawwithzeros {#1/#2}}{{#1}{#2}}%
}%
\def\XINT_gctcv_loop_g #1#2#3#4%
{%
    \XINT_gctcv_loop_h {#4{#1}}{#2#3}% 1.09b removes [0]
}%
\def\XINT_gctcv_loop_h #1#2#3/%
{%
    \xint_gob_til_W #3\XINT_gctcv_end\W
    \expandafter\XINT_gctcv_loop_i
    \romannumeral0\xintrawwithzeros {#3}.#2{#1}%
}%
\def\XINT_gctcv_loop_i #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_gctcv_loop_j\expandafter
    {\romannumeral0\XINT_mul_fork #1\Z #6\Z }%
    {\romannumeral0\XINT_mul_fork #1\Z #5\Z }%
    {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
    {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
}%
\def\XINT_gctcv_loop_j #1#2%
{%
    \expandafter\XINT_gctcv_loop_k\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_gctcv_loop_k #1#2%
{%
    \expandafter\XINT_gctcv_loop_l\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_gctcv_loop_l #1#2%
{%
    \expandafter\XINT_gctcv_loop_m\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_gctcv_loop_m #1#2{\XINT_gctcv_loop_a {#2}#1}%
\def\XINT_gctcv_end #1.#2#3#4#5#6{ #6}%
```

37.21 \xintiGCtoCv

```
\def\xintiGCtoCv {\romannumeral0\xintigctocv }%
\def\xintigctocv #1%
{%
    \expandafter\XINT_igctcv_prep \romannumeral-`0#1+\W/%
}%
\def\XINT_igctcv_prep
{%
    \XINT_igctcv_loop_a {}1001%
}%
\def\XINT_igctcv_loop_a #1#2#3#4#5#6+%
{%
    \expandafter\XINT_igctcv_loop_b
    \romannumeral-`0#6.{#2}{#3}{#4}{#5}{#1}%
}%
\def\XINT_igctcv_loop_b #1.#2#3#4#5%
{%
    \expandafter\XINT_igctcv_loop_c\expandafter
    {\romannumeral0\xintiiadd {#5}{\XINT_Mul {#1}{#3}}}%
    {\romannumeral0\xintiiadd {#4}{\XINT_Mul {#1}{#2}}}%
    {{#2}{#3}}%
}%
\def\XINT_igctcv_loop_c #1#2%
{%
    \expandafter\XINT_igctcv_loop_f\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_igctcv_loop_f #1#2#3#4/%
{%
    \xint_gob_til_W #4\XINT_igctcv_end_a\W
    \expandafter\XINT_igctcv_loop_g
    \romannumeral-`0#4.#1#2{#3}%
}%
\def\XINT_igctcv_loop_g #1.#2#3#4#5%
{%
    \expandafter\XINT_igctcv_loop_h\expandafter
    {\romannumeral0\XINT_mul_fork #1\Z #5\Z }%
    {\romannumeral0\XINT_mul_fork #1\Z #4\Z }%
    {{#2}{#3}}%
}%
\def\XINT_igctcv_loop_h #1#2%
{%
    \expandafter\XINT_igctcv_loop_i\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_igctcv_loop_i #1#2{\XINT_igctcv_loop_k #2{#2#1}}%
\def\XINT_igctcv_loop_k #1#2%
{%
    \expandafter\XINT_igctcv_loop_l\expandafter
    {\romannumeral0\xintrawwithzeros {#1/#2}}%
}%
\def\XINT_igctcv_loop_l #1#2#3{\XINT_igctcv_loop_a {#3{#1}}#2}%1.09i removes [0]
```

```
\def\XINT_igctcv_end_a #1.#2#3#4#5%
{%
    \expandafter\XINT_igctcv_end_b\expandafter
    {\romannumeral0\xintrawwithzeros {#2/#3}}%
}%
\def\XINT_igctcv_end_b #1#2{ #2{#1}}% 1.09b removes [0]
```

37.22 \xintCntoF

Modified in 1.06 to give the N first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

```
\def\xintCntoF {\romannumeral0\xintcntof }%
\def\xintcntof #1%
{%
    \expandafter\XINT_cntf\expandafter {\the\numexpr #1}%
}%
\def\XINT_cntf #1#2%
{%
   \ifnum #1>\xint_c_
      \xint_afterfi {\expandafter\XINT_cntf_loop\expandafter
                     {\the\numexpr #1-1\expandafter}\expandafter
                     {\romannumeral-`0#2{#1}}{#2}}%
   \else
      \xint_afterfi
         {\ifnum #1=\xint_c_
              \xint_afterfi {\expandafter\space \romannumeral-`0#2{0}}%
          \else \xint_afterfi { 0/1[0]}%
          \fi}%
   \fi
}%
\def\XINT_cntf_loop #1#2#3%
{%
    \ifnum #1>\xint_c_ \else \XINT_cntf_exit \fi
    \expandafter\XINT_cntf_loop\expandafter
    {\the\numexpr #1-1\expandafter }\expandafter
    {\romannumeral0\xintadd {\xintDiv {1[0]}{#2}}{#3{#1}}}%
    {#3}%
}%
\def\XINT_cntf_exit \fi
    \expandafter\XINT_cntf_loop\expandafter
    #1\expandafter #2#3%
{%
    \fi\xint_gobble_ii #2%
}%
```

37.23 \xintGCntoF

Modified in 1.06 to give the N first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

```
\def\xintGCntoF {\romannumeral0\xintgcntof }%
\def\xintgcntof #1%
{%
    \expandafter\XINT_gcntf\expandafter {\the\numexpr #1}%
}%
\def\XINT_gcntf #1#2#3%
{%
   \ifnum #1>\xint_c_
      \xint_afterfi {\expandafter\XINT_gcntf_loop\expandafter
                     {\the\numexpr #1-1\expandafter}\expandafter
                     {\romannumeral-`0#2{#1}}{#2}{#3}}%
   \else
      \xint_afterfi
         {\ifnum #1=\xint_c_
              \xint_afterfi {\expandafter\space\romannumeral-`0#2{0}}%
          \else \xint_afterfi { 0/1[0]}%
          \fi}%
   \fi
}%
\def\XINT_gcntf_loop #1#2#3#4%
{%
    \ifnum #1>\xint_c_ \else \XINT_gcntf_exit \fi
    \expandafter\XINT_gcntf_loop\expandafter
    {\the\numexpr #1-1\expandafter }\expandafter
    {\romannumeral0\xintadd {\xintDiv {#4{#1}}{#2}}{#3{#1}}}%
    {#3}{#4}%
}%
\def\XINT_gcntf_exit \fi
    \expandafter\XINT_gcntf_loop\expandafter
    #1\expandafter #2#3#4%
{%
    \fi\xint_gobble_ii #2%
}%
```

37.24 \xintCntoCs

Modified in 1.06 to give the N first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

```
\def\xintCntoCs {\romannumeral0\xintcntocs }%
\def\xintcntocs #1%
{%
    \expandafter\XINT_cntcs\expandafter {\the\numexpr #1}%
}%
\def\XINT_cntcs #1#2%
{%
   \ifnum #1<0
      \xint_afterfi { }% 1.09i: a 0/1[0] was strangely here, removed
   \else
```

```
      \xint_afterfi {\expandafter\XINT_cntcs_loop\expandafter
                      {\the\numexpr #1-1\expandafter}\expandafter
                      {\expandafter{\romannumeral-`0#2{#1}}}{#2}}%
   \fi
}%
\def\XINT_cntcs_loop #1#2#3%
{%
    \ifnum #1>-1 \else \XINT_cntcs_exit \fi
    \expandafter\XINT_cntcs_loop\expandafter
    {\the\numexpr #1-1\expandafter }\expandafter
    {\expandafter{\romannumeral-`0#3{#1}},#2}{#3}%
}%
\def\XINT_cntcs_exit \fi
    \expandafter\XINT_cntcs_loop\expandafter
    #1\expandafter #2#3%
{%
    \fi\XINT_cntcs_exit_b #2%
}%
\def\XINT_cntcs_exit_b #1,{ }%
```

37.25 \xintCntoGC

Modified in 1.06 to give the N first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

```
\def\xintCntoGC {\romannumeral0\xintcntogc }%
\def\xintcntogc #1%
{%
    \expandafter\XINT_cntgc\expandafter {\the\numexpr #1}%
}%
\def\XINT_cntgc #1#2%
{%
   \ifnum #1<0
      \xint_afterfi { }% 1.09i there was as strange 0/1[0] here, removed
   \else
      \xint_afterfi {\expandafter\XINT_cntgc_loop\expandafter
                      {\the\numexpr #1-1\expandafter}\expandafter
                      {\expandafter{\romannumeral-`0#2{#1}}}{#2}}%
   \fi
}%
\def\XINT_cntgc_loop #1#2#3%
{%
    \ifnum #1>-1 \else \XINT_cntgc_exit \fi
    \expandafter\XINT_cntgc_loop\expandafter
    {\the\numexpr #1-1\expandafter }\expandafter
    {\expandafter{\romannumeral-`0#3{#1}}+1/#2}{#3}%
}%
\def\XINT_cntgc_exit \fi
    \expandafter\XINT_cntgc_loop\expandafter
```

```
    #1\expandafter #2#3%
{%
    \fi\XINT_cntgc_exit_b #2%
}%
\def\XINT_cntgc_exit_b #1+1/{ }%
```

37.26 \xintGCntoGC

Modified in 1.06 to give the N first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

```
\def\xintGCntoGC {\romannumeral0\xintgcntogc }%
\def\xintgcntogc #1%
{%
    \expandafter\XINT_gcntgc\expandafter {\the\numexpr #1}%
}%
\def\XINT_gcntgc #1#2#3%
{%
   \ifnum #1<0
      \xint_afterfi { }% 1.09i now returns nothing
   \else
      \xint_afterfi {\expandafter\XINT_gcntgc_loop\expandafter
                     {\the\numexpr #1-1\expandafter}\expandafter
                     {\expandafter{\romannumeral-`0#2{#1}}}{#2}{#3}}%
   \fi
}%
\def\XINT_gcntgc_loop #1#2#3#4%
{%
    \ifnum #1>-1 \else \XINT_gcntgc_exit \fi
    \expandafter\XINT_gcntgc_loop_b\expandafter
    {\expandafter{\romannumeral-`0#4{#1}}/#2}{#3{#1}}{#1}{#3}{#4}%
}%
\def\XINT_gcntgc_loop_b #1#2#3%
{%
    \expandafter\XINT_gcntgc_loop\expandafter
    {\the\numexpr #3-1\expandafter}\expandafter
    {\expandafter{\romannumeral-`0#2}+#1}%
}%
\def\XINT_gcntgc_exit \fi
    \expandafter\XINT_gcntgc_loop_b\expandafter #1#2#3#4#5%
{%
    \fi\XINT_gcntgc_exit_b #1%
}%
\def\XINT_gcntgc_exit_b #1/{ }%
```

37.27 \xintCstoGC

```
\def\xintCstoGC {\romannumeral0\xintcstogc }%
\def\xintcstogc #1%
```

```
{%
    \expandafter\XINT_cstc_prep \romannumeral-`0#1,\W,%
}%
\def\XINT_cstc_prep #1,{\XINT_cstc_loop_a {{#1}}}%
\def\XINT_cstc_loop_a #1#2,%
{%
    \xint_gob_til_W #2\XINT_cstc_end\W
    \XINT_cstc_loop_b {#1}{#2}%
}%
\def\XINT_cstc_loop_b #1#2{\XINT_cstc_loop_a {#1+1/{#2}}}%
\def\XINT_cstc_end\W\XINT_cstc_loop_b #1#2{ #1}%
```

37.28 \xintGCtoGC

```
\def\xintGCtoGC {\romannumeral0\xintgctogc }%
\def\xintgctogc #1%
{%
    \expandafter\XINT_gctgc_start \romannumeral-`0#1+\W/%
}%
\def\XINT_gctgc_start {\XINT_gctgc_loop_a {}}%
\def\XINT_gctgc_loop_a #1#2+#3/%
{%
    \xint_gob_til_W #3\XINT_gctgc_end\W
    \expandafter\XINT_gctgc_loop_b\expandafter
    {\romannumeral-`0#2}{#3}{#1}%
}%
\def\XINT_gctgc_loop_b #1#2%
{%
    \expandafter\XINT_gctgc_loop_c\expandafter
    {\romannumeral-`0#2}{#1}%
}%
\def\XINT_gctgc_loop_c #1#2#3%
{%
    \XINT_gctgc_loop_a {#3{#2}+{#1}/}%
}%
\def\XINT_gctgc_end\W\expandafter\XINT_gctgc_loop_b
{%
    \expandafter\XINT_gctgc_end_b
}%
\def\XINT_gctgc_end_b #1#2#3{ #3{#1}}%
\XINT_restorecatcodes_endinput%
```

38 Package xintexpr implementation

The first version was released in June 2013. I was greatly helped in this task of writing an expandable parser of infix operations by the comments provided in `l3fp-parse.dtx` (in its version as available in April-May 2013). One will recognize in particular the idea of the 'until' macros; I have not looked into the actual `l3fp` code beyond the very useful comments provided in its documentation.

A main worry was that my data has no a priori bound on its size; to keep the code reasonably efficient, I experimented with a technique of storing and retrieving data expandably as *names* of control sequences. Intermediate computation results are stored as control sequences `\.=a/b[n]`.

Another peculiarity is that the input is allowed to contain (but only where the scanner looks for a number or fraction) material within braces `{...}`. This will be expanded completely and must give an integer, decimal number or fraction (not in scientific notation). Conversely any explict fraction `A/B[n]` *with the brackets* or macro expanding to such a thing **must** be enclosed within such braces: square brackets are not acceptable by the expression parser.

These two things are a bit *experimental* and perhaps I will opt for another approach at a later stage. To circumvent the potential hash-table impact of the `\.=a/b[n]` I have provided the macro creators `\xintNewExpr` and `\xintNewFloatExpr`.

Roughly speaking, the parser mechanism is as follows: at any given time the last found "operator" has its associated `until` macro awaiting some news from the token flow; first `getnext` expands forward in the hope to construct some number, which may come from a parenthesized sub-expression, from some braced material, or from a digit by digit scan. After this number has been formed the next operator is looked for by the `getop` macro. Once `getop` has finished its job, `until` is presented with three tokens: the first one is the precedence level of the new found operator (which may be an end of expression marker), the second is the operator character token (earlier versions had here already some macro name, but in order to keep as much common code to expr and floatexpr common as possible, this was modied) of the new found operator, and the third one is the newly found number (which was encountered just before the new operator).

The `until` macro of the earlier operator examines the precedence level of the new found one, and either executes the earlier operator (in the case of a binary operation, with the found number and a previously stored one) or it delays execution, giving the hand to the `until` macro of the operator having been found of higher precedence.

A minus sign acting as prefix gets converted into a (unary) operator inheriting the precedence level of the previous operator.

Once the end of the expression is found (it has to be marked by a `\relax`) the final result is output as four tokens: the first one a catcode 11 exclamation mark, the second one an error generating macro, the third one a printing macro and the fourth is `\.=a/b[n]`. The prefix `\xintthe` makes the output printable by killing the first two tokens.

Version `1.08b [2013/06/14]` corrected a problem originating in the attempt to attribute a special rôle to braces: expansion could be stopped by space tokens, as various macros tried to expand without grabbing what came next. They now have a doubled `\romannumeral-'0`.

Version `1.09a [2013/09/24]` has a better mechanism regarding `\xintthe`, more commenting and better organization of the code, and most importantly it implements functions, comparison operators, logic operators, conditionals. The code was reorganized and expansion proceeds a bit differently in order to have the `_getnext` and `_getop` codes entirely shared by `\xintexpr` and `\xintfloatexpr`. `\xintNewExpr` was rewritten in order to work with the standard macro parameter character `#`, to be catcode protected and to also allow comma separated expressions.

Version `1.09c [2013/10/09]` added the `bool` and `togl` operators, `\xintboolexpr`, and `\xintNewNumExpr`, `\xintNewBoolExpr`. The code for `\xintNewExpr` is shared with `float`, `num`, and `bool`-expressions. Also the precedence level of the postfix operators `!`, `?` and `:` has been made lower than the one of functions.

Version 1.09i [2013/12/18] unpacks count and dimen registers and control squences, with tacit multiplication. It has also made small improvements. (speed gains in macro expansions in quite a few places.)

Also, 1.09i implements \xintiiexpr, \xinttheiiexpr. New function frac. And encapsulation in \csname..\endcsname is done with .= as first tokens, so unpacking with \string can be done in a completely escape char agnostic way.

Version 1.09j [2014/01/09] extends the tacit multiplication to the case of a sub \xintexpressions. Also, it now \xint_protects the result of the \xintexpr full expansions, thus, an \xintexpr without \xintthe prefix can be used not only as the first item within an "\fdef" as previously but also now anywhere within an \edef. Five tokens are used to pack the computation result rather than the possibly hundreds or thousands of digits of an \xintthe unlocked result. I deliberately omit a second \xint_protect which, however would be necessary if some macro \.=digits/digits[digits] had acquired some expandable meaning elsewhere. But this seems not that probable, and adding the protection would mean impacting everything only to allow some crazy user which has loaded something else than xint to do an \edef... the \xintexpr computations are otherwise in no way affected if such control sequences have a meaning.

Version 1.09k [2014/01/21] does tacit multiplication also for an opening parenthesis encountered during the scanning of a number, or at a time when the parser expects an infix operator.

And it adds to the syntax recognition of hexadecimal numbers starting with a ", and having possibly a fractional part (except in \xintiiexpr, naturally).

Release 1.09kb fixes the bug introduced in \xintNewExpr in 1.09i of December 2013: an \endlinechar -1 was removed, but without it there is a spurious trailing space token in the outputs of the created macros, and nesting is then impossible.

Contents

38.1 Catcodes, ε-TEX and reload detection

The code for reload detection is copied from Heiko Oberdiek's packages, and adapted here to check for previous loading of the **xintfrac** package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \def\space { }%
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintexpr.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintexpr}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintexpr.sty
      \ifx\w\relax % but xintfrac.sty not yet loaded.
         \y{xintexpr}{now issuing \string\input\space xintfrac.sty}%
         \def\z{\endgroup\input xintfrac.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xintfrac.sty not yet loaded.
            \y{xintexpr}{now issuing \string\RequirePackage{xintfrac}}%
            \def\z{\endgroup\RequirePackage{xintfrac}}%
          \fi
      \else
        \y{xintexpr}{I was already loaded, aborting input}%
        \aftergroup\endinput
      \fi
    \fi
```

```
  \fi
\z%
```

38.2 Confirmation of xintfrac loading

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \ifdefined\PackageInfo
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \else
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
  \fi
  \def\empty {}%
  \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
  \ifx\w\relax % Plain TeX, user gave a file name at the prompt
      \y{xintexpr}{Loading of package xintfrac failed, aborting input}%
      \aftergroup\endinput
  \fi
  \ifx\w\empty % LaTeX, user gave a file name at the prompt
      \y{xintexpr}{Loading of package xintfrac failed, aborting input}%
      \aftergroup\endinput
  \fi
\endgroup%
```

38.3 Catcodes

```
\XINTsetupcatcodes%
```

38.4 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintexpr}%
  [2014/02/13 v1.09kb Expandable expression parser (jfB)]%
```

38.5 Encapsulation in pseudo cs names, helper macros

1.09i uses .= for encapsulation, thus allowing \escapechar to be anything (all previous releases were with ., so \escapechar 46 was forbidden). Besides, the \edef definition has \space already expanded, perhaps this will compensate a tiny bit the time penalty of '.=' viz '.' in unlocking... well not really, I guess. (for no special reason 1.09k uses some \expandafter's rather than \edef+\noexpand's for the definition of \XINT_expr_lock)

```
\def\xint_gob_til_! #1!{}% nota bene: this ! has catcode 11
\expandafter\def\expandafter
\XINT_expr_lock\expandafter#\expandafter1\expandafter !\expandafter
    {\expandafter\expandafter\space\csname .=#1\endcsname }%
\def\XINT_expr_unlock  {\expandafter\XINT_expr_unlock_a\string }%
\def\XINT_expr_unlock_a #1.={}%
\def\XINT_expr_unexpectedtoken {\xintError:ignored }%
\def\XINT_newexpr_setprefix #1>{\noexpand\romannumeral-`0}%
\def\xint_UDxintrelaxfork #1\xint_relax #2#3\krof {#2}%
```

38.6 \xintexpr, \xinttheexpr, \xintthe, ...

\xintthe is defined with a parameter, I guess I wanted to make sure no stray space tokens could cause a problem.

With 1.09i, \xintiexpr replaces \xintnumexpr which is kept for compatibility but will be removed at some point. Should perhaps issue a warning, but well, people can also read the documentation. Also 1.09i removes \xinttheeval.

1.09i has re-organized the material here.

1.09j modifies the mechanism of \XINT_expr_usethe and \XINT_expr_print, etc... in order for \xintexpr-essions to be usable within \edef'initions. I hesitated quite a bit with adding \xint_protect in front of the \.=digits macros, which will in 99.99999% of use cases supposed all have \relax meaning; and it is a bit of a pain, really, it is quite a pain to add these extra tokens only for \edef contexts and for situations which will never occur... well no damn'it let's *NOT* add this extra \xint_protect. Just one before the printing macro (which can not be \protected, else \xintthe could not work).

```
\def\xint_protect {\noexpand\xint_protect\noexpand }% 1.09j
\def\XINT_expr_done      {!\XINT_expr_usethe\xint_protect\XINT_expr_print }%
\let\XINT_iiexpr_done    \XINT_expr_done
\def\XINT_iexpr_done     {!\XINT_expr_usethe\xint_protect\XINT_iexpr_print }%
\def\XINT_flexpr_done    {!\XINT_expr_usethe\xint_protect\XINT_flexpr_print }%
\def\XINT_boolexpr_done {!\XINT_expr_usethe\xint_protect\XINT_boolexpr_print }%
\protected\def\XINT_expr_usethe #1#2#3% modified in 1.09j
   {\xintError:missing_xintthe!\show#3missing xintthe (see log)!}%
\def\xintthe #1{\romannumeral-`0\expandafter\xint_gobble_iii\romannumeral-`0#1}%
\let\XINT_expr_print      \XINT_expr_unlock
\def\XINT_iexpr_print     #1{\xintRound:csv {\XINT_expr_unlock #1}}%
\def\XINT_flexpr_print    #1{\xintFloat:csv {\XINT_expr_unlock #1}}%
\def\XINT_boolexpr_print #1{\xintIsTrue:csv{\XINT_expr_unlock #1}}%
\def\xintexpr       {\romannumeral0\xinteval      }%
\def\xintfloatexpr  {\romannumeral0\xintfloateval }%
\def\xintiiexpr     {\romannumeral0\xintiieval    }%
\def\xinteval
   {\expandafter\XINT_expr_until_end_a  \romannumeral-`0\XINT_expr_getnext  }%
\def\xintfloateval
   {\expandafter\XINT_flexpr_until_end_a\romannumeral-`0\XINT_expr_getnext  }%
\def\xintiieval
   {\expandafter\XINT_iiexpr_until_end_a\romannumeral-`0\XINT_expr_getnext  }%
```

```
\def\xinttheexpr
   {\romannumeral-`0\expandafter\xint_gobble_iii\romannumeral0\xinteval      }%
\def\xintthefloatexpr
   {\romannumeral-`0\expandafter\xint_gobble_iii\romannumeral0\xintfloateval }%
\def\xinttheiiexpr
   {\romannumeral-`0\expandafter\xint_gobble_iii\romannumeral0\xintiieval    }%
\def\xintiexpr      {\romannumeral0\expandafter\expandafter\expandafter
    \XINT_iexpr_done \expandafter\xint_gobble_iv\romannumeral0\xinteval    }%
\def\xinttheiexpr   {\romannumeral-`0\expandafter\expandafter\expandafter
    \XINT_iexpr_print\expandafter\xint_gobble_iv\romannumeral0\xinteval    }%
\def\xintboolexpr      {\romannumeral0\expandafter\expandafter\expandafter
    \XINT_boolexpr_done \expandafter\xint_gobble_iv\romannumeral0\xinteval }%
\def\xinttheboolexpr   {\romannumeral-`0\expandafter\expandafter\expandafter
    \XINT_boolexpr_print\expandafter\xint_gobble_iv\romannumeral0\xinteval }%
\let\xintnumexpr   \xintiexpr    % deprecated
\let\xintthenumexpr\xinttheiexpr % deprecated
```

38.7 \xintifboolexpr, \xintifboolfloatexpr, cshxintifbooliiexpr

1.09c. Does not work with comma separated expressions. I could make use \xintORof:csv (or AND, or XOR) to allow it, but don't know it the overhead is worth it.

1.09i adds \xintifbooliiexpr

```
\def\xintifboolexpr #1%
              {\romannumeral0\xintifnotzero {\xinttheexpr #1\relax}}%
\def\xintifboolfloatexpr #1%
              {\romannumeral0\xintifnotzero {\xintthefloatexpr #1\relax}}%
\def\xintifbooliiexpr #1%
              {\romannumeral0\xintifnotzero {\xinttheiiexpr #1\relax}}%
```

38.8 \XINT_get_next: looking for a number

June 14: 1.08b adds a second \romannumeral-`0 to \XINT_expr_getnext in an attempt to solve a problem with space tokens stopping the \romannumeral and thus preventing expansion of the following token. For example: 1+ \the\cnta caused a problem, as `\the' was not expanded. I did not define \XINT_expr_getnext as a macro with parameter (which would have cured preventively this), precisely to try to recognize brace pairs. The second \romannumeral-`0 is added for the same reason in other places.

The get-next scans forward to find a number: after expansion of what comes next, an opening parenthesis signals a parenthesized sub-expression, a ! with catcode 11 signals there was there an \xintexpr.. \relax sub-expression (now evaluated), a minus is a prefix operator, a plus is silently ignored, a digit or decimal point signals to start gathering a number, braced material {...} is allowed and will be directly fed into a \csname..\endcsname for complete expansion which must delivers a (fractional) number, possibly ending in [n]; explicit square brackets must be enclosed into such braces. Once a number issues from the previous procedures, it is a locked into a \csname...\endcsname, and the flow then proceeds with \XINT_expr_getop which will scan for an infix or postfix operator following the number.

A special r\^ole is played by underscores _ for use with \xintNewExpr to input macro parameters.

Release 1.09a implements functions; the idea is that a letter (actually, anything not otherwise recognized!) triggers the function name gatherer, the comma is promoted to a binary operator of priority intermediate between parentheses and infix operators. The code had some other revisions in order for all the _getnext and _getop macros to now be shared by \xintexpr and \xintfloatexpr.

1.09i now allows direct insertion of \count's, \dimen's and \skip's which will be unpacked using \number.

1.09i speeds up a bit the recognition of a braced thing: the case of a single braced control sequence makes a third expansion mandatory, let's do it immediately and not wait. So macros got shuffled and modified a bit.

\XINT_expr_unpackvariable does not insert a [0] for compatibility with \xintiiexpr. A [0] would have made a bit faster \xintexpr macros when dealing with an unpacked count control sequence, as without it the \xintnum will be used in the parsing by xintfrac macros when the number is used. But [0] is not accepted by most macros ultimately called by \xintiiexpr.

```
\def\XINT_expr_getnext
{%
    \expandafter\XINT_expr_getnext_checkforbraced_a
    \romannumeral-`0\romannumeral-`0%
}%
\def\XINT_expr_getnext_checkforbraced_a #1% was done later in <1.09i
{%
    \expandafter\XINT_expr_getnext_checkforbraced_b\expandafter
    {\romannumeral-`0#1}%
}%
\def\XINT_expr_getnext_checkforbraced_b #1%
{%
    \XINT_expr_getnext_checkforbraced_c #1\xint_relax\Z {#1}%
}%
\def\XINT_expr_getnext_checkforbraced_c #1#2%
{%
    \xint_UDxintrelaxfork
              #1\XINT_expr_getnext_wasemptyorspace
              #2\XINT_expr_getnext_gotonetoken_wehope
    \xint_relax\XINT_expr_getnext_gotbracedstuff
    \krof
}% doubly braced things are not acceptable, will cause errors.
\def\XINT_expr_getnext_wasemptyorspace #1{\XINT_expr_getnext }%
\def\XINT_expr_getnext_gotbracedstuff #1\xint_relax\Z #2%
{%
    \expandafter\XINT_expr_getop\csname .=#2\endcsname
}%
\def\XINT_expr_getnext_gotonetoken_wehope\Z #1%
{% screens out sub-expressions and \count or \dimen registers/variables
```

```
    \xint_gob_til_! #1\XINT_expr_subexpr !% recall this ! has catcode 11
    \ifcat\relax#1% \count or \numexpr etc... token or count, dimen, skip cs
       \expandafter\XINT_expr_countdimenetc_fork
    \else
       \expandafter\expandafter\expandafter
       \XINT_expr_getnext_onetoken_fork\expandafter\string
    \fi
    #1%
}%
\def\XINT_expr_subexpr !#1\fi !{\expandafter\XINT_expr_getop\xint_gobble_iii }%
\def\XINT_expr_countdimenetc_fork #1%
{%
    \ifx\count#1\else\ifx#1\dimen\else\ifx#1\numexpr\else\ifx#1\dimexpr\else
    \ifx\skip#1\else\ifx\glueexpr#1\else
      \XINT_expr_unpackvariable
    \fi\fi\fi\fi\fi\fi
    \expandafter\XINT_expr_getnext\number #1%
}%
\def\XINT_expr_unpackvariable\fi\fi\fi\fi\fi\fi\expandafter\XINT_expr_getnext
                \number #1{\fi\fi\fi\fi\fi\fi
    \expandafter\XINT_expr_getop\csname .=\number#1\endcsname }%
```

1.09a: In order to have this code shared by \xintexpr and \xintfloatexpr, I have moved to the until macros the responsability to choose expr or floatexpr, hence here, the opening parenthesis for example can not be triggered directly as it would not know in which context it works. Hence the \xint_c_xviii ({}. And also the mechanism of \xintNewExpr has been modified to allow use of #.

1.09i also has \xintiiexpr.

```
\begingroup
\lccode`*=`#
\lowercase{\endgroup
\def\XINT_expr_sixwayfork #1(-.+*#2#3\krof {#2}%
\def\XINT_expr_getnext_onetoken_fork #1%
{% The * is in truth catcode 12 #. For (hacking) use with \xintNewExpr.
    \XINT_expr_sixwayfork
        #1-.+*{\xint_c_xviii ({}}% back to until for oparen triggering
        (#1.+*{-}%
        (-#1+*{\XINT_expr_scandec_II .}%
        (-.#1*{\XINT_expr_getnext }%
         (-.+#1{\XINT_expr_scandec_II }%
         (-.+*{\XINT_expr_scan_dec_or_func #1}%
    \krof
}}%
```

38.9 `\XINT_expr_scan_dec_or_func`: collecting an integer or decimal number or hexa-decimal number or function name

`\XINT_expr_scanfunc_b` rewritten in 1.09i. And 1.09k adds hexadecimal numbers to the syntax, with " as prefix, and possibly a fractional part. Naturally to postfix with an E in scientific notation, one would need to surround the hexadecimal number in parentheses to avoid ambiguities; or rather, just use a lowercase e. By the way, if I allowed only lowercase e for scientific notation I could possibly fuse together the scanning in the dec and hexa cases; up to some loss of syntax control in the dec case.

```
\def\XINT_expr_scan_dec_or_func #1% this #1 has necessarily here catcode 12
{%
    \ifnum \xint_c_ix<1#1
        \expandafter\XINT_expr_scandec_I
    \else
        \if #1"\expandafter\expandafter\expandafter\XINT_expr_scanhex_I
        \else % We assume we are dealing with a function name!!
              \expandafter\expandafter\expandafter\XINT_expr_scanfunc
        \fi
    \fi #1%
}%
\def\XINT_expr_scanfunc
{%
    \expandafter\XINT_expr_func\romannumeral-`0\XINT_expr_scanfunc_c
}%
\def\XINT_expr_scanfunc_c #1%
{%
    \expandafter #1\romannumeral-`0\expandafter
    \XINT_expr_scanfunc_a\romannumeral-`0\romannumeral-`0%
}%
\def\XINT_expr_scanfunc_a #1% please no braced things here!
{%
    \ifcat #1\relax % missing opening parenthesis, probably
        \expandafter\XINT_expr_scanfunc_panic
    \else
        \xint_afterfi{\expandafter\XINT_expr_scanfunc_b \string #1}%
    \fi
}%
\def\xint_UDparenfork #1()#2#3\krof {#2}%
\def\XINT_expr_scanfunc_b #1%
{%
    \xint_UDparenfork
        #1){(}% and then \XINT_expr_func
        (#1{(}% and then \XINT_expr_func (this is for bool/toggle names)
         (){\XINT_expr_scanfunc_c #1}%
    \krof
}%
\def\XINT_expr_scanfunc_panic {\xintError:bigtroubleahead(0\relax }%
```

```
\def\XINT_expr_func #1(% common to expr and flexpr and iiexpr
{%
    \xint_c_xviii @{#1}% functions have the highest priority.
}%
```

Scanning for a number of fraction. Once gathered, lock it and do _getop. 1.09i modifies \XINT_expr_scanintpart_a (splits _aa) and also \XINT_expr_scanfracpart_a in order for the tacit multiplication of \count's and \dimen's to be compatible with escape-char=a digit.

1.09j further extends for recognition of an \xint..expr and then insertion of a * (which is done in \XINT_expr_getop_a).

```
\def\XINT_expr_scandec_I
{%
    \expandafter\XINT_expr_getop\romannumeral-‘0\expandafter
    \XINT_expr_lock\romannumeral-‘0\XINT_expr_scanintpart_b
}%
\def\XINT_expr_scandec_II
{%
    \expandafter\XINT_expr_getop\romannumeral-‘0\expandafter
    \XINT_expr_lock\romannumeral-‘0\XINT_expr_scanfracpart_b
}%
\def\XINT_expr_scanintpart_a #1% Please no braced material: 123{FORBIDDEN}
{%  careful that ! has catcode letter here
    \ifcat #1\relax\else
           \ifx !#1\else
        \expandafter\expandafter\expandafter
        \xint_thirdofthree
    \fi\fi
    \xint_firstoftwo !% this stops the scan
    {\expandafter\XINT_expr_scanintpart_aa\string }#1%
}%
\def\XINT_expr_scanintpart_aa #1%
{%
    \ifnum \xint_c_ix<1#1
       \expandafter\XINT_expr_scanintpart_b
    \else
       \if .#1%
           \expandafter\expandafter\expandafter
           \XINT_expr_scandec_transition
       \else % gather what we got so far, leave catcode 12 #1 in stream
           \expandafter\expandafter\expandafter !% ! of catcode 11, space needed
       \fi
    \fi
    #1%
}%
\def\XINT_expr_scanintpart_b #1%
{%
    \expandafter #1\romannumeral-‘0\expandafter
    \XINT_expr_scanintpart_a\romannumeral-‘0\romannumeral-‘0%
```

```
}%
\def\XINT_expr_scandec_transition .%
{%
    \expandafter.\romannumeral-`0\expandafter
    \XINT_expr_scanfracpart_a\romannumeral-`0\romannumeral-`0%
}%
\def\XINT_expr_scanfracpart_a #1%
{%
    \ifcat #1\relax\else
           \ifx !#1\else
        \expandafter\expandafter\expandafter
        \xint_thirdofthree
    \fi\fi
    \xint_firstoftwo !% this stops the scan
    {\expandafter\XINT_expr_scanfracpart_aa\string }#1%
}%
\def\XINT_expr_scanfracpart_aa #1%
{%
    \ifnum \xint_c_ix<1#1
       \expandafter\XINT_expr_scanfracpart_b
    \else
       \expandafter !%
    \fi
    #1%
}%
\def\XINT_expr_scanfracpart_b #1%
{%
    \expandafter #1\romannumeral-`0\expandafter
    \XINT_expr_scanfracpart_a\romannumeral-`0\romannumeral-`0%
}%
```

1.09k [2014/01/21]: added scanning for an hexadecimal number, possibly with a "hexa-decimal" part, only with uppercase ABCDEF (xintbinhex.sty works with ABCDEF, as tex itself requires uppercase letters after ", thus at least I feel comfortable with not bothering allowing abcdef... which would be possible but would complicate things; although perhaps there could be some use for lowercase. If needed, can be implemented, but I will probably long be dead when an archivist droid will be the first around circa 2500 AD to read these lines).

For compatibility with \xintiiexpr, the [] thing is incorporated only if there the parser encounters a . indicating a fractional part (this fractional part may be empty). Thus for (infinitesimally) faster further processing by \xintexpr, "ABC.+ etc... is better than "ABC+ etc... on the other hand the initial processing with a . followed by an empty fractional part adds its bit of overhead... The . is not allowed in \xintiiexpr, as it will provoke insertion of [0] which is incompatible with it.

```
\def\XINT_expr_scanhex_I #1%
{%
    \expandafter\XINT_expr_getop\romannumeral-`0\expandafter
    \XINT_expr_lock\expandafter\XINT_expr_inhex
```

```
    \romannumeral-`0\XINT_expr_scanhexI_a
}%
\def\XINT_expr_inhex #1.#2#3;% expanded inside \csname..\endcsname
{%
    \if#2I\xintHexToDec{#1}%
    \else
      \xintiiMul{\xintiiPow{625}{\xintLength{#3}}}{\xintHexToDec{#1#3}}%
      [\the\numexpr-4*\xintLength{#3}]%
    \fi
}%
\def\XINT_expr_scanhexI_a #1%
{%
    \ifcat #1\relax\else
           \ifx !#1\else
        \expandafter\expandafter\expandafter
        \xint_thirdofthree
    \fi\fi
    \xint_firstoftwo {.I;!}%
    {\expandafter\XINT_expr_scanhexI_aa\string }#1%
}%
\def\XINT_expr_scanhexI_aa #1%
{%
    \if\ifnum`#1>`/
       \ifnum`#1>`9
       \ifnum`#1>`@
       \ifnum`#1>`F
       0\else1\fi\else0\fi\else1\fi\else0\fi 1%
       \expandafter\XINT_expr_scanhexI_b
    \else
       \if .#1%
           \expandafter\xint_firstoftwo
       \else % gather what we got so far, leave catcode 12 #1 in stream
           \expandafter\xint_secondoftwo
       \fi
       {\expandafter\XINT_expr_scanhex_transition}%
       {\xint_afterfi {.I;!}}%
    \fi
    #1%
}%
\def\XINT_expr_scanhexI_b #1%
{%
    \expandafter #1\romannumeral-`0\expandafter
    \XINT_expr_scanhexI_a\romannumeral-`0\romannumeral-`0%
}%
\def\XINT_expr_scanhex_transition .%
{%
    \expandafter.\expandafter.\romannumeral-`0\expandafter
    \XINT_expr_scanhexII_a\romannumeral-`0\romannumeral-`0%
}%
```

```
\def\XINT_expr_scanhexII_a #1%
{%
    \ifcat #1\relax\else
           \ifx !#1\else
       \expandafter\expandafter\expandafter
       \xint_thirdofthree
    \fi\fi
    \xint_firstoftwo {;!}% this stops the scan
    {\expandafter\XINT_expr_scanhexII_aa\string }#1%
}%
\def\XINT_expr_scanhexII_aa #1%
{%
    \if\ifnum`#1>`/
      \ifnum`#1>`9
      \ifnum`#1>`@
      \ifnum`#1>`F
      0\else1\fi\else0\fi\else1\fi\else0\fi 1%
      \expandafter\XINT_expr_scanhexII_b
    \else
      \xint_afterfi {;!}%
    \fi
    #1%
}%
\def\XINT_expr_scanhexII_b #1%
{%
    \expandafter #1\romannumeral-`0\expandafter
    \XINT_expr_scanhexII_a\romannumeral-`0\romannumeral-`0%
}%
```

38.10 \XINT_expr_getop: looking for an operator

June 14 (1.08b): I add here a second \romannumeral-`0, because \XINT_expr_getnext and others try to expand the next token but without grabbing it.

This finds the next infix operator or closing parenthesis or postfix exclamation mark ! or expression end. It then leaves in the token flow <precedence> <operator> <locked number>. The <precedence> is generally a character command which thus stops expansion and gives back control to an \XINT_expr_until_<op> command; or it is the minus sign which will be converted by a suitable \XINT_expr_checkifprefix_<p> into an operator with a given inherited precedence. Earlier releases than 1.09c used tricks for the postfix !, ?, :, with <precedence> being in fact a macro to act immediately, and then re-activate \XINT_expr_getop.

In versions earlier than 1.09a the <operator> was already made in to a control sequence; but now it is a left as a token and will be (generally) converted by the until macro which knows if it is in a \xintexpr or an \xintfloatexpr. (or an \xintiiexpr, since 1.09i)

1.09i allows \count's, \dimen's, \skip's with tacit multiplication.

1.09j extends the mechanism of tacit multiplication to the case of a sub xintexpression in its various variants. Careful that our ! has catcode 11 so \ifx! would be a disaster...

1.09k extends tacit multiplication to the case of an encountered opening parenthesis.

```
\def\XINT_expr_getop #1% this #1 is the current locked computed value
{% full expansion of next token, first swallowing a possible space
    \expandafter\XINT_expr_getop_a\expandafter #1%
    \romannumeral-‘0\romannumeral-‘0%
}%
\def\XINT_expr_getop_a #1#2%
{% if a control sequence is found, must be either \relax or register|variable
    \ifcat #2\relax\expandafter\xint_firstoftwo
             \else \expandafter\xint_secondoftwo
    \fi
    {\ifx #2\relax\expandafter\xint_firstofthree
             \else\expandafter\xint_secondofthree % tacit multiplication
     \fi }%
    {\ifx !#2\expandafter\xint_secondofthree      % tacit multiplication
        \else            % 1.09k adds tacit multiplication in front of (
          \if (#2\expandafter\expandafter\expandafter\xint_secondofthree
          \else
            \expandafter\expandafter\expandafter\xint_thirdofthree
          \fi
     \fi }%
    {\XINT_expr_foundend #1}%
    {\XINT_expr_foundop *#1#2}%
    {\XINT_expr_foundop #2#1}%
}%
\def\XINT_expr_foundend {\xint_c_ \relax }% \relax is a place holder here.
\def\XINT_expr_foundop #1% then becomes <prec> <op> and is followed by <\.=f>
{% 1.09a: no control sequence \XINT_expr_op_#1, code common to expr/flexpr
    \ifcsname XINT_expr_precedence_#1\endcsname
        \expandafter\xint_afterfi\expandafter
        {\csname XINT_expr_precedence_#1\endcsname #1}%
    \else
        \XINT_expr_unexpectedtoken
        \expandafter\XINT_expr_getop
    \fi
}%
```

38.11 Parentheses

1.09a removes some doubling of \romannumeral-‘\0 from 1.08b which served no useful purpose here (I think...).

```
\def\XINT_tmpa #1#2#3#4#5%
{%
    \def#1##1%
    {%
        \xint_UDsignfork
```

```
                     ##1{\expandafter#1\romannumeral-`0#3}%
                       -{#2##1}%
        \krof
    }%
    \def#2##1##2%
    {%
        \ifcase ##1\expandafter #4%
        \or\xint_afterfi{%
                          \XINT_expr_extra_closing_paren
                          \expandafter #1\romannumeral-`0\XINT_expr_getop
                        }%
        \else
 \xint_afterfi{\expandafter#1\romannumeral-`0\csname XINT_#5_op_##2\endcsname }%
        \fi
    }%
}%
\xintFor #1 in {expr,flexpr,iiexpr} \do {%
\expandafter\XINT_tmpa
    \csname XINT_#1_until_end_a\expandafter\endcsname
    \csname XINT_#1_until_end_b\expandafter\endcsname
    \csname XINT_#1_op_-vi\expandafter\endcsname
    \csname XINT_#1_done\endcsname
    {#1}%
}%
\def\XINT_expr_extra_closing_paren {\xintError:removed }%
\def\XINT_tmpa #1#2#3#4#5#6%
{%
    \def #1{\expandafter #3\romannumeral-`0\XINT_expr_getnext }%
    \let #2#1%
    \def #3##1{\xint_UDsignfork
                ##1{\expandafter #3\romannumeral-`0#5}%
                  -{#4##1}%
               \krof }%
    \def #4##1##2%
    {%
        \ifcase ##1\expandafter \XINT_expr_missing_cparen
             \or   \expandafter \XINT_expr_getop
             \else \xint_afterfi
        {\expandafter #3\romannumeral-`0\csname XINT_#6_op_##2\endcsname }%
        \fi
    }%
}%
\xintFor #1 in {expr,flexpr,iiexpr} \do {%
\expandafter\XINT_tmpa
    \csname XINT_#1_op_(\expandafter\endcsname
    \csname XINT_#1_oparen\expandafter\endcsname
    \csname XINT_#1_until_)_a\expandafter\endcsname
    \csname XINT_#1_until_)_b\expandafter\endcsname
    \csname XINT_#1_op_-vi\endcsname
```

```
    {#1}%
}%
\def\XINT_expr_missing_cparen {\xintError:inserted \xint_c_ \XINT_expr_done }%
\expandafter\let\csname XINT_expr_precedence_)\endcsname   \xint_c_i
\expandafter\let\csname XINT_flexpr_precedence_)\endcsname \xint_c_i
\expandafter\let\csname XINT_iiexpr_precedence_)\endcsname \xint_c_i
\expandafter\let\csname XINT_expr_op_)\endcsname  \XINT_expr_getop
\expandafter\let\csname XINT_flexpr_op_)\endcsname\XINT_expr_getop
\expandafter\let\csname XINT_iiexpr_op_)\endcsname\XINT_expr_getop
```

38.12 The \XINT_expr_until_<op> macros for boolean operators, comparison operators, arithmetic operators, scientfic notation.

Extended in 1.09a with comparison and boolean operators. 1.09i adds \xintiiexpr and incorporates optional part [\XINTdigits] for a tiny bit faster float operations now already equipped with their optional argument

.

```
\def\XINT_tmpb #1#2#3#4#5#6%#7%
{%
    \expandafter\XINT_tmpc
    \csname XINT_#1_op_#3\expandafter\endcsname
    \csname XINT_#1_until_#3_a\expandafter\endcsname
    \csname XINT_#1_until_#3_b\expandafter\endcsname
    \csname XINT_#1_op_-#5\expandafter\endcsname
    \csname xint_c_#4\expandafter\endcsname
    \csname #2#6\expandafter\endcsname
    \csname XINT_expr_precedence_#3\endcsname {#1}%{#7}%
}%
\def\XINT_tmpc #1#2#3#4#5#6#7#8#9%
{%
    \def #1##1% \XINT_expr_op_<op>
    {% keep value, get next number and operator, then do until
        \expandafter #2\expandafter ##1%
        \romannumeral-`0\expandafter\XINT_expr_getnext
    }%
    \def #2##1##2% \XINT_expr_until_<op>_a
    {\xint_UDsignfork
        ##2{\expandafter #2\expandafter ##1\romannumeral-`0#4}%
          -{#3##1##2}%
     \krof }%
    \def #3##1##2##3##4% \XINT_expr_until_<op>_b
    {%  either execute next operation now, or first do next (possibly unary)
      \ifnum ##2>#5%
        \xint_afterfi {\expandafter #2\expandafter ##1\romannumeral-`0%
                       \csname XINT_#8_op_##3\endcsname {##4}}%
      \else
        \xint_afterfi
        {\expandafter ##2\expandafter ##3%
```

```
        \csname .=#6#9{\XINT_expr_unlock ##1}{\XINT_expr_unlock ##4}\endcsname }%
        \fi
      }%
      \let #7#5%
}%
\def\XINT_tmpa #1{\XINT_tmpb {expr}{xint}#1{}}%
\xintApplyInline {\XINT_tmpa }{%
  {|{iii}{vi}{OR}}%
  {&{iv}{vi}{AND}}%
  {<{v}{vi}{Lt}}%
  {>{v}{vi}{Gt}}%
  {={v}{vi}{Eq}}%
  {+{vi}{vi}{Add}}%
  {-{vi}{vi}{Sub}}%
  {*{vii}{vii}{Mul}}%
  {/{vii}{vii}{Div}}%
  {^{viii}{viii}{Pow}}%
  {e{ix}{ix}{fE}}%
  {E{ix}{ix}{fE}}%
}%
\def\XINT_tmpa #1{\XINT_tmpb {flexpr}{xint}#1{}}%
\xintApplyInline {\XINT_tmpa }{%
  {|{iii}{vi}{OR}}%
  {&{iv}{vi}{AND}}%
  {<{v}{vi}{Lt}}%
  {>{v}{vi}{Gt}}%
  {={v}{vi}{Eq}}%
}%
\def\XINT_tmpa #1{\XINT_tmpb {flexpr}{XINTinFloat}#1{[\XINTdigits]}}%
\xintApplyInline {\XINT_tmpa }{%
  {+{vi}{vi}{Add}}%
  {-{vi}{vi}{Sub}}%
  {*{vii}{vii}{Mul}}%
  {/{vii}{vii}{Div}}%
  {^{viii}{viii}{Power}}%
  {e{ix}{ix}{fE}}%
  {E{ix}{ix}{fE}}%
}%
\def\XINT_tmpa #1{\XINT_tmpb {iiexpr}{xint}#1{}}%
\xintApplyInline {\XINT_tmpa }{%
  {|{iii}{vi}{OR}}%
  {&{iv}{vi}{AND}}%
  {<{v}{vi}{Lt}}%
  {>{v}{vi}{Gt}}%
  {={v}{vi}{Eq}}%
  {+{vi}{vi}{iiAdd}}%
  {-{vi}{vi}{iiSub}}%
  {*{vii}{vii}{iiMul}}%
  {/{vii}{vii}{iiQuo}}%
```

```
  {^{viii}{viii}{iiPow}}%
  {e{ix}{ix}{iE}}%
  {E{ix}{ix}{iE}}%
}%
```

38.13 The comma as binary operator

New with 1.09a.

```
\def\XINT_tmpa #1#2#3#4#5#6%
{%
    \def #1##1% \XINT_expr_op_,_a
    {%
        \expandafter #2\expandafter ##1\romannumeral-`0\XINT_expr_getnext
    }%
    \def #2##1##2% \XINT_expr_until_,_a
    {\xint_UDsignfork
        ##2{\expandafter #2\expandafter ##1\romannumeral-`0#4}%
          -{#3##1##2}%
     \krof }%
    \def #3##1##2##3##4% \XINT_expr_until_,_b
    {%
      \ifnum ##2>\xint_c_ii
        \xint_afterfi {\expandafter #2\expandafter ##1\romannumeral-`0%
                       \csname XINT_#6_op_##3\endcsname {##4}}%
      \else
        \xint_afterfi
        {\expandafter ##2\expandafter ##3%
         \csname .=\XINT_expr_unlock ##1,\XINT_expr_unlock ##4\endcsname }%
      \fi
    }%
    \let #5\xint_c_ii
}%
\xintFor #1 in {expr,flexpr,iiexpr} \do {%
\expandafter\XINT_tmpa
    \csname XINT_#1_op_,\expandafter\endcsname
    \csname XINT_#1_until_,_a\expandafter\endcsname
    \csname XINT_#1_until_,_b\expandafter\endcsname
    \csname XINT_#1_op_-vi\expandafter\endcsname
    \csname XINT_expr_precedence_,\endcsname {#1}%
}%
```

38.14 \XINT_expr_op_-<level>: minus as prefix inherits its precedence level

1.09i: \xintiiexpr must use \xintiiOpp (or at least \xintiOpp, but that would be a waste; however impacts round and trunc as I allow them).

```
\def\XINT_tmpa #1#2#3%
```

```
{%
    \expandafter\XINT_tmpb
    \csname XINT_#1_op_-#3\expandafter\endcsname
    \csname XINT_#1_until_-#3_a\expandafter\endcsname
    \csname XINT_#1_until_-#3_b\expandafter\endcsname
    \csname xint_c_#3\endcsname {#1}#2%
}%
\def\XINT_tmpb #1#2#3#4#5#6%
{%
    \def #1% \XINT_expr_op_-<level>
    {%  get next number+operator then switch to _until macro
        \expandafter #2\romannumeral-`0\XINT_expr_getnext
    }%
    \def #2##1% \XINT_expr_until_-<l>_a
    {\xint_UDsignfork
        ##1{\expandafter #2\romannumeral-`0#1}%
          -{#3##1}%
     \krof }%
    \def #3##1##2##3% \XINT_expr_until_-<l>_b
    {%  _until tests precedence level with next op, executes now or postpones
        \ifnum ##1>#4%
         \xint_afterfi {\expandafter #2\romannumeral-`0%
                        \csname XINT_#5_op_##2\endcsname {##3}}%
        \else
         \xint_afterfi {\expandafter ##1\expandafter ##2%
                        \csname .=#6{\XINT_expr_unlock ##3}\endcsname }%
        \fi
    }%
}%
\xintApplyInline{\XINT_tmpa {expr}\xintOpp}{{vi}{vii}{viii}{ix}}%
\xintApplyInline{\XINT_tmpa {flexpr}\xintOpp}{{vi}{vii}{viii}{ix}}%
\xintApplyInline{\XINT_tmpa {iiexpr}\xintiiOpp}{{vi}{vii}{viii}{ix}}%
```

38.15 ? as two-way conditional

New with 1.09a. Modified in 1.09c to have less precedence than functions. Code is cleaner as it does not play tricks with _precedence. There is no associated until macro, because action is immediate once activated (only a previously scanned function can delay activation).

```
\let\XINT_expr_precedence_? \xint_c_x
\def \XINT_expr_op_? #1#2#3%
{%
        \xintifZero{\XINT_expr_unlock  #1}%
                   {\XINT_expr_getnext #3}%
                   {\XINT_expr_getnext #2}%
}%
\let\XINT_flexpr_op_?\XINT_expr_op_?
\let\XINT_iiexpr_op_?\XINT_expr_op_?
```

38.16 : as three-way conditional

New with 1.09a. Modified in 1.09c to have less precedence than functions.

```
\let\XINT_expr_precedence_: \xint_c_x
\def \XINT_expr_op_: #1#2#3#4%
{%
        \xintifSgn {\XINT_expr_unlock  #1}%
                   {\XINT_expr_getnext #2}%
                   {\XINT_expr_getnext #3}%
                   {\XINT_expr_getnext #4}%
}%
\let\XINT_flexpr_op_:\XINT_expr_op_:
\let\XINT_iiexpr_op_:\XINT_expr_op_:
```

38.17 ! as postfix factorial operator

The factorial is currently the exact one, there is no float version. Starting with 1.09c, it has lower priority than functions, it is not executed immediately anymore. The code is cleaner and does not abuse _precedence, but does assign it a true level. There is no until macro, because the factorial acts on what precedes it.

```
\let\XINT_expr_precedence_! \xint_c_x
\def\XINT_expr_op_! #1{\expandafter\XINT_expr_getop
       \csname .=\xintFac{\XINT_expr_unlock #1}\endcsname }%
\let\XINT_flexpr_op_!\XINT_expr_op_!
\def\XINT_iiexpr_op_! #1{\expandafter\XINT_expr_getop
       \csname .=\xintiFac{\XINT_expr_unlock #1}\endcsname }%
```

38.18 Functions

New with 1.09a. Names of ..Float..:csv macros have been changed in 1.09h

```
\def\XINT_tmpa #1#2#3#4{%
    \def #1##1%
    {%
        \ifcsname XINT_expr_onlitteral_##1\endcsname
           \expandafter\XINT_expr_funcoflitteral
        \else
           \expandafter #2%
        \fi {##1}%
    }%
   \def #2##1%
   {%
        \ifcsname XINT_#4_func_##1\endcsname
        \xint_afterfi
            {\expandafter\expandafter\csname XINT_#4_func_##1\endcsname}%
        \else \csname xintError:unknown ‘##1\string’\endcsname
```

```
            \xint_afterfi{\expandafter\XINT_expr_func_unknown}%
        \fi
        \romannumeral-`0#3%
    }%
}%
\xintFor #1 in {expr,flexpr,iiexpr} \do {%
     \expandafter\XINT_tmpa
                 \csname XINT_#1_op_@\expandafter\endcsname
                 \csname XINT_#1_op_@@\expandafter\endcsname
                 \csname XINT_#1_oparen\endcsname {#1}%
}%
\def\XINT_expr_funcoflitteral #1%
{%
    \expandafter\expandafter\csname XINT_expr_onlitteral_#1\endcsname
    \romannumeral-`0\XINT_expr_scanfunc
}%
\def\XINT_expr_onlitteral_bool #1#2#3{\expandafter\XINT_expr_getop
    \csname .=\xintBool{#3}\endcsname }%
\def\XINT_expr_onlitteral_togl #1#2#3{\expandafter\XINT_expr_getop
    \csname .=\xintToggle{#3}\endcsname }%
\def\XINT_expr_func_unknown #1#2#3%  1.09i removes [0], because \xintiiexpr
   {\expandafter #1\expandafter #2\csname .=0\endcsname }%
\def\XINT_expr_func_reduce #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintIrr {\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_flexpr_func_reduce\XINT_expr_func_reduce
% \XINT_iiexpr_func_reduce not defined
\def\XINT_expr_func_frac #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintTFrac {\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_flexpr_func_frac #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
    .=\XINTinFloatFrac [\XINTdigits]{\XINT_expr_unlock #3}\endcsname
}%
% \XINT_iiexpr_func_frac not defined
\def\XINT_expr_func_sqr #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintSqr {\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_flexpr_func_sqr #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
             .=\XINTinFloatMul [\XINTdigits]%
```

```
            {\XINT_expr_unlock #3}{\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_iiexpr_func_sqr #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintiiSqr {\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_expr_func_abs #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintAbs {\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_flexpr_func_abs\XINT_expr_func_abs
\def\XINT_iiexpr_func_abs #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintiiAbs {\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_expr_func_sgn #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintSgn {\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_flexpr_func_sgn\XINT_expr_func_sgn
\def\XINT_iiexpr_func_sgn #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintiiSgn {\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_expr_func_floor #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintFloor {\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_flexpr_func_floor\XINT_expr_func_floor
\let\XINT_iiexpr_func_floor\XINT_expr_func_floor
\def\XINT_expr_func_ceil #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintCeil {\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_flexpr_func_ceil\XINT_expr_func_ceil
\let\XINT_iiexpr_func_ceil\XINT_expr_func_ceil
\def\XINT_expr_twoargs #1,#2,{{#1}{#2}}%
\def\XINT_expr_func_quo #1#2#3%
{%
    \expandafter #1\expandafter #2\csname .=%
    \expandafter\expandafter\expandafter\xintQuo
    \expandafter\XINT_expr_twoargs
```

```
    \romannumeral-`0\XINT_expr_unlock #3,\endcsname
}%
\let\XINT_flexpr_func_quo\XINT_expr_func_quo
\def\XINT_iiexpr_func_quo #1#2#3%
{%
    \expandafter #1\expandafter #2\csname .=%
    \expandafter\expandafter\expandafter\xintiiQuo
    \expandafter\XINT_expr_twoargs
    \romannumeral-`0\XINT_expr_unlock #3,\endcsname
}%
\def\XINT_expr_func_rem #1#2#3%
{%
    \expandafter #1\expandafter #2\csname .=%
    \expandafter\expandafter\expandafter\xintRem
    \expandafter\XINT_expr_twoargs
    \romannumeral-`0\XINT_expr_unlock #3,\endcsname
}%
\let\XINT_flexpr_func_rem\XINT_expr_func_rem
\def\XINT_iiexpr_func_rem #1#2#3%
{%
    \expandafter #1\expandafter #2\csname .=%
    \expandafter\expandafter\expandafter\xintiiRem
    \expandafter\XINT_expr_twoargs
    \romannumeral-`0\XINT_expr_unlock #3,\endcsname
}%
\def\XINT_expr_oneortwo #1#2#3,#4,#5.%
{%
    \if\relax#5\relax\expandafter\xint_firstoftwo\else
                     \expandafter\xint_secondoftwo\fi
    {#1{0}}{#2{\xintNum {#4}}}{#3}%
}%
\def\XINT_expr_func_round #1#2#3%
{%
    \expandafter #1\expandafter #2\csname .=%
    \expandafter\XINT_expr_oneortwo
    \expandafter\xintiRound\expandafter\xintRound
    \romannumeral-`0\XINT_expr_unlock #3,,.\endcsname
}%
\let\XINT_flexpr_func_round\XINT_expr_func_round
\def\XINT_iiexpr_oneortwo #1#2,#3,#4.%
{%
    \if\relax#4\relax\expandafter\xint_firstoftwo\else
                     \expandafter\xint_secondoftwo\fi
    {#1{0}}{#1{#3}}{#2}%
}%
\def\XINT_iiexpr_func_round #1#2#3%
{%
    \expandafter #1\expandafter #2\csname .=%
    \expandafter\XINT_iiexpr_oneortwo\expandafter\xintiRound
```

```
    \romannumeral-`0\XINT_expr_unlock #3,,.\endcsname
}%
\def\XINT_expr_func_trunc #1#2#3%
{%
    \expandafter #1\expandafter #2\csname .=%
    \expandafter\XINT_expr_oneortwo
    \expandafter\xintiTrunc\expandafter\xintTrunc
    \romannumeral-`0\XINT_expr_unlock #3,,.\endcsname
}%
\let\XINT_flexpr_func_trunc\XINT_expr_func_trunc
\def\XINT_iiexpr_func_trunc #1#2#3%
{%
    \expandafter #1\expandafter #2\csname .=%
    \expandafter\XINT_iiexpr_oneortwo\expandafter\xintiTrunc
    \romannumeral-`0\XINT_expr_unlock #3,,.\endcsname
}%
\def\XINT_expr_argandopt #1,#2,#3.%
{%
    \if\relax#3\relax\expandafter\xint_firstoftwo\else
                     \expandafter\xint_secondoftwo\fi
    {[\XINTdigits]}{[\xintNum {#2}]}{#1}%
}%
\def\XINT_expr_func_float #1#2#3%
{%
    \expandafter #1\expandafter #2\csname .=%
    \expandafter\XINTinFloat
    \romannumeral-`0\expandafter\XINT_expr_argandopt
    \romannumeral-`0\XINT_expr_unlock #3,,.\endcsname
}%
\let\XINT_flexpr_func_float\XINT_expr_func_float
% \XINT_iiexpr_func_float not defined
\def\XINT_expr_func_sqrt #1#2#3%
{%
    \expandafter #1\expandafter #2\csname .=%
    \expandafter\XINTinFloatSqrt
    \romannumeral-`0\expandafter\XINT_expr_argandopt
    \romannumeral-`0\XINT_expr_unlock #3,,.\endcsname
}%
\let\XINT_flexpr_func_sqrt\XINT_expr_func_sqrt
\def\XINT_iiexpr_func_sqrt #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintiSqrt {\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_expr_func_gcd #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintGCDof:csv{\XINT_expr_unlock #3}\endcsname
}%
```

```
\let\XINT_flexpr_func_gcd\XINT_expr_func_gcd
\let\XINT_iiexpr_func_gcd\XINT_expr_func_gcd
\def\XINT_expr_func_lcm #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintLCMof:csv{\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_flexpr_func_lcm\XINT_expr_func_lcm
\let\XINT_iiexpr_func_lcm\XINT_expr_func_lcm
\def\XINT_expr_func_max #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintMaxof:csv{\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_iiexpr_func_max #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintiMaxof:csv{\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_flexpr_func_max #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\XINTinFloatMaxof:csv{\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_expr_func_min #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintMinof:csv{\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_iiexpr_func_min #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintiMinof:csv{\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_flexpr_func_min #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\XINTinFloatMinof:csv{\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_expr_func_sum #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintSum:csv{\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_flexpr_func_sum #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\XINTinFloatSum:csv{\XINT_expr_unlock #3}\endcsname
}%
```

```
\def\XINT_iiexpr_func_sum #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintiiSum:csv{\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_expr_func_prd #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintPrd:csv{\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_flexpr_func_prd #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\XINTinFloatPrd:csv{\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_iiexpr_func_prd #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintiiPrd:csv{\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_expr_func_add\XINT_expr_func_sum
\let\XINT_expr_func_mul\XINT_expr_func_prd
\let\XINT_flexpr_func_add\XINT_flexpr_func_sum
\let\XINT_flexpr_func_mul\XINT_flexpr_func_prd
\let\XINT_iiexpr_func_add\XINT_iiexpr_func_sum
\let\XINT_iiexpr_func_mul\XINT_iiexpr_func_prd
\def\XINT_expr_func_? #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintIsNotZero {\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_flexpr_func_? \XINT_expr_func_?
\let\XINT_iiexpr_func_? \XINT_expr_func_?
\def\XINT_expr_func_! #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintIsZero {\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_flexpr_func_! \XINT_expr_func_!
\let\XINT_iiexpr_func_! \XINT_expr_func_!
\def\XINT_expr_func_not #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintIsZero {\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_flexpr_func_not \XINT_expr_func_not
\let\XINT_iiexpr_func_not \XINT_expr_func_not
\def\XINT_expr_func_all #1#2#3%
{%
```

```
    \expandafter #1\expandafter #2\csname
         .=\xintANDof:csv{\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_flexpr_func_all\XINT_expr_func_all
\let\XINT_iiexpr_func_all\XINT_expr_func_all
\def\XINT_expr_func_any #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintORof:csv{\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_flexpr_func_any\XINT_expr_func_any
\let\XINT_iiexpr_func_any\XINT_expr_func_any
\def\XINT_expr_func_xor #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\xintXORof:csv{\XINT_expr_unlock #3}\endcsname
}%
\let\XINT_flexpr_func_xor\XINT_expr_func_xor
\let\XINT_iiexpr_func_xor\XINT_expr_func_xor
\def\xintifNotZero:: #1,#2,#3,{\xintifNotZero{#1}{#2}{#3}}%
\def\XINT_expr_func_if #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\expandafter\xintifNotZero::
               \romannumeral-`0\XINT_expr_unlock #3,\endcsname
}%
\let\XINT_flexpr_func_if\XINT_expr_func_if
\let\XINT_iiexpr_func_if\XINT_expr_func_if
\def\xintifSgn:: #1,#2,#3,#4,{\xintifSgn{#1}{#2}{#3}{#4}}%
\def\XINT_expr_func_ifsgn #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
         .=\expandafter\xintifSgn::
               \romannumeral-`0\XINT_expr_unlock #3,\endcsname
}%
\let\XINT_flexpr_func_ifsgn\XINT_expr_func_ifsgn
\let\XINT_iiexpr_func_ifsgn\XINT_expr_func_ifsgn
```

38.19 \xintNewExpr, \xintNewFloatExpr...

Rewritten in 1.09a. Now, the parameters of the formula are entered in the usual way by the user, with # not _. And _ is assigned to make macros not expand. This way, : is freed, as we now need it for the ternary operator. (on numeric data; if use with macro parameters, should be coded with the functionn ifsgn , rather)

Code unified in 1.09c, and \xintNewNumExpr, \xintNewBoolExpr added. 1.09i renames \xintNewNumExpr to \xintNewIExpr, and defines \xintNewIIExpr.

```
\def\XINT_newexpr_print #1{\ifnum\xintNthElt{0}{#1}>1
                              \expandafter\xint_firstoftwo
```

```
                           \else
                            \expandafter\xint_secondoftwo
                           \fi
                           {_xintListWithSep,{#1}}{\xint_firstofone#1}}%
\xintForpair #1#2 in {(fl,Float),(i,iRound0),(bool,IsTrue)}\do {%
    \expandafter\def\csname XINT_new#1expr_print\endcsname
                      ##1{\ifnum\xintNthElt{0}{##1}>1
                            \expandafter\xint_firstoftwo
                           \else
                            \expandafter\xint_secondoftwo
                           \fi
                           {_xintListWithSep,{\xintApply{_xint#2}{##1}}}
                           {_xint#2##1}}}%
\toks0 {}%
\xintFor #1 in {Bool,Toggle,Floor,Ceil,iRound,Round,iTrunc,Trunc,TFrac,%
        Lt,Gt,Eq,AND,OR,IsNotZero,IsZero,ifNotZero,ifSgn,%
        Irr,Num,Abs,Sgn,Opp,Quo,Rem,Add,Sub,Mul,Sqr,Div,Pow,Fac,fE,iSqrt,%
        iiAdd,iiSub,iiMul,iiSqr,iiPow,iiQuo,iiRem,iiSgn,iiAbs,iiOpp,iE}\do
 {\toks0
  \expandafter{\the\toks0\expandafter\def\csname xint#1\endcsname {_xint#1}}}%
\xintFor #1 in {,Sqrt,Add,Sub,Mul,Div,Power,fE,Frac}\do
  {\toks0
   \expandafter{\the\toks0\expandafter\def\csname XINTinFloat#1\endcsname
              {_XINTinFloat#1}}}%
\xintFor #1 in {GCDof,LCMof,Maxof,Minof,ANDof,ORof,XORof,Sum,Prd,%
                iMaxof,iMinof,iiSum,iiPrd}\do
 {\toks0
  \expandafter{\the\toks0\expandafter\def\csname xint#1:csv\endcsname
                ####1{_xint#1{\xintCSVtoListNonStripped {####1}}}}}%
\xintFor #1 in {Maxof,Minof,Sum,Prd}\do
 {\toks0
  \expandafter{\the\toks0\expandafter\def\csname XINTinFloat#1:csv\endcsname
                ####1{_XINTinFloat#1{\xintCSVtoListNonStripped {####1}}}}}%
\expandafter\def\expandafter\XINT_expr_protect\expandafter{\the\toks0
    \def\XINTdigits {_XINTdigits}%
    \def\XINT_expr_print ##1{\expandafter\XINT_newexpr_print\expandafter
            {\romannumeral0\xintcsvtolistnonstripped{\XINT_expr_unlock ##1}}}%
    \def\XINT_flexpr_print ##1{\expandafter\XINT_newflexpr_print\expandafter
            {\romannumeral0\xintcsvtolistnonstripped{\XINT_expr_unlock ##1}}}%
    \def\XINT_iexpr_print ##1{\expandafter\XINT_newiexpr_print\expandafter
            {\romannumeral0\xintcsvtolistnonstripped{\XINT_expr_unlock ##1}}}%
    \def\XINT_boolexpr_print ##1{\expandafter\XINT_newboolexpr_print\expandafter
            {\romannumeral0\xintcsvtolistnonstripped{\XINT_expr_unlock ##1}}}%
}%
\toks0 {}%
\def\xintNewExpr      {\xint_NewExpr\xinttheexpr      }%
\def\xintNewFloatExpr {\xint_NewExpr\xintthefloatexpr }%
\def\xintNewIExpr     {\xint_NewExpr\xinttheiexpr     }%
\let\xintNewNumExpr\xintNewIExpr
```

```
\def\xintNewIIExpr    {\xint_NewExpr\xinttheiiexpr    }%
\def\xintNewBoolExpr  {\xint_NewExpr\xinttheboolexpr  }%
```

1.09i has added \escapechar 92, as \meaning is used in \XINT_NewExpr, and a non existent escape-char would be a problem with \scantokens. Also \catcode32 is set to 10 in \xintexprSafeCatcodes for being extra-safe.

```
\def\xint_NewExpr #1#2[#3]%
{%
 \begingroup
    \ifcase #3\relax
        \toks0 {\xdef #2}%
    \or \toks0 {\xdef #2##1}%
    \or \toks0 {\xdef #2##1##2}%
    \or \toks0 {\xdef #2##1##2##3}%
    \or \toks0 {\xdef #2##1##2##3##4}%
    \or \toks0 {\xdef #2##1##2##3##4##5}%
    \or \toks0 {\xdef #2##1##2##3##4##5##6}%
    \or \toks0 {\xdef #2##1##2##3##4##5##6##7}%
    \or \toks0 {\xdef #2##1##2##3##4##5##6##7##8}%
    \or \toks0 {\xdef #2##1##2##3##4##5##6##7##8##9}%
    \fi
    \xintexprSafeCatcodes
    \escapechar92
    \XINT_NewExpr #1%
}%
\catcode`* 13
\def\XINT_NewExpr #1#2%
{%
    \def\XINT_tmpa ##1##2##3##4##5##6##7##8##9{#2}%
    \XINT_expr_protect
    \lccode`*=`_ \lowercase {\def*}{!noexpand!}%
    \catcode`_ 13 \catcode`: 11
    \endlinechar -1 % 1.09i, 2013/12/18 not sure why I had that? removed.
    % 2014/02/13: you idiot, if not then spurious extra ending space
    % token makes impossible nesting of created macros!
    \everyeof {\noexpand }%
    \edef\XINT_tmpb ##1##2##3##4##5##6##7##8##9%
        {\scantokens
         \expandafter{\romannumeral-`0#1%
                     \XINT_tmpa {####1}{####2}{####3}%
                                {####4}{####5}{####6}%
                                {####7}{####8}{####9}%
                     \relax}}%
     \lccode`*=`\$ \lowercase {\def*}{####}%
     \catcode`\$ 13 \catcode`! 0 \catcode`_ 11 %
     \the\toks0
     {\scantokens\expandafter{\expandafter
                 \XINT_newexpr_setprefix\meaning\XINT_tmpb}}%
  \endgroup
```

```
}%
\let\xintexprRestoreCatcodes\empty
\def\xintexprSafeCatcodes
{% for end user.
    \edef\xintexprRestoreCatcodes  {%
        \catcode34=\the\catcode34   % "
        \catcode63=\the\catcode63   % ?
        \catcode124=\the\catcode124 % |
        \catcode38=\the\catcode38   % &
        \catcode33=\the\catcode33   % !
        \catcode93=\the\catcode93   % ]
        \catcode91=\the\catcode91   % [
        \catcode94=\the\catcode94   % ^
        \catcode95=\the\catcode95   % _
        \catcode47=\the\catcode47   % /
        \catcode41=\the\catcode41   % )
        \catcode40=\the\catcode40   % (
        \catcode42=\the\catcode42   % *
        \catcode43=\the\catcode43   % +
        \catcode62=\the\catcode62   % >
        \catcode60=\the\catcode60   % <
        \catcode58=\the\catcode58   % :
        \catcode46=\the\catcode46   % .
        \catcode45=\the\catcode45   % -
        \catcode44=\the\catcode44   % ,
        \catcode61=\the\catcode61   % =
        \catcode32=\the\catcode32\relax % space
    }% it's hard to know where to stop...
        \catcode34=12  % "
        \catcode63=12  % ?
        \catcode124=12 % |
        \catcode38=4   % &
        \catcode33=12  % !
        \catcode93=12  % ]
        \catcode91=12  % [
        \catcode94=7   % ^
        \catcode95=8   % _
        \catcode47=12  % /
        \catcode41=12  % )
        \catcode40=12  % (
        \catcode42=12  % *
        \catcode43=12  % +
        \catcode62=12  % >
        \catcode60=12  % <
        \catcode58=12  % :
        \catcode46=12  % .
        \catcode45=12  % -
        \catcode44=12  % ,
        \catcode61=12  % =
```

```
        \catcode32=10   % space
}%
\let\XINT_tmpa\relax \let\XINT_tmpb\relax \let\XINT_tmpc\relax
\XINT_restorecatcodes_endinput%
```

xinttools:	1069.
xint:	3515.
xintbinhex:	640.
xintgcd:	463.
xintfrac:	2578.
xintseries:	417.
xintcfrac:	944.
xintexpr:	1153.

Total number of code lines: 10779. Each package starts with circa 80 lines dealing with catcodes, package identification and reloading management, also for Plain TeX. Version 1.09kb of 2014/02/13.