The xint bundle

Jean-François Burnol
jfbu (at) free (dot) fr
Package version: 1.09d (2013/10/22)
Documentation generated from the source file
with timestamp "22-10-2013 at 22:40:55 CEST"

Abstract

The **xint** package implements with expandable TeX macros the basic arithmetic operations of addition, subtraction, multiplication and division, applied to arbitrarily long numbers. The **xintfrac** package extends the scope of **xint** to fractional numbers with arbitrarily long numerators and denominators.

xintexpr provides 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, subexpressions, macros expanding to the previous items.

The **xintbinhex** package 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, and **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 Quick introduction

The **xint** bundle consists of three principal components **xint**, **xintfrac** (which loads **xint**), and **xintexpr** (which loads **xintfrac**), and four additional modules. They may be used with Plain TeX, LaTeX or any other macro package based on TeX. The package requires the ε-TeX extensions which in modern distributions are made available by default, except if you invoke TeX under the name `tex` in command line.

The goal is too compute *exactly*, purely by expansion, without count registers nor assignments nor definitions, with arbitrarily big numbers and fractions. As will be commented upon more later, this works fine when the data has dozens of digits, but multiplying out two 1000 digits numbers under this constraint of expandability is expensive; so in many situations the package will be used for fixed point (rounding or truncating each intermediate result) or floating point computations. The "floating point" macros work with a given arbitrary precision (default is 16 digits; from the remark made above, beyond 100 digits things will start becoming too slow if hundreds of computations are needed). The only non-algebraic operation which is currently implemented is the extraction of square roots.

The package macros expand their arguments[1]; as they are themselves completely expandable, this means that one may nest them arbitrarily deep to construct complicated (and still completely expandable) formulas.

But one will presumably prefer to use the `\xintexpr ... \relax` parser as it allows infix notations, function names (corresponding to some of the package macros), comparison operators, boolean operators, 2way and 3way conditionals.

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 little 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\expandafter\expandafter
    \allowsplits #1\relax }% Expands twice before printing.
%% (all macros from the xint bundle expand in two steps to their final
%% output)
```

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).

2 Recent changes

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` 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, and their replacement texts are allowed to close groups as happens with the tabulation in alignments,
- `\xintForpair`, `\xintForthree`, `\xintForfour` are experimental variants of `\xintFor`,

[1] see in section 9 the related explanations.

- \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 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.[2] 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.

Release 1.0 ([2013/03/28]): initial release.

3 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 theoretical upper bound, the true limitation is from the *time* taken by the expansion-compatible algorithms, this 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

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

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.[3, 4]

The LaTeX3 project has implemented expandably floating-point computations with 16 significant figures (l3fp), including special functions such as exp, log, sine and cosine.

The **xint** package can be used for 24, 40, etc... significant figures but one rather quickly (not much beyond 100 figures perhaps) 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.[5]

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.[6] [7]

4 Missing things

'Arbitrary-precision' floating-point operations are currently limited to the basic four operations, the power function with integer exponent, and the extraction of square-roots.

[3]currently (`v1.08`), the only non-elementary operation implemented for floating point numbers is the square-root extraction; furthermore no `NaN`'s nor error traps has been implemented, only the notion of 'scientific notation with a given number of significant figures'.

[4]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.)

[5]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.

[6]I could, naturally, be proven wrong!

[7]The LuaTEX project possibly makes endeavours such as **xint** appear even more insane that they are, in truth.

5 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

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

- the formula may be input without `\xinttheexpr` through suitable nesting of various package macros. Here one could use:

```
\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 25). So, a utility `\xintNewExpr` is provided to do the work of translating an `\xintexpr`-ession with parameters into a chain of macro evaluations.[8]

```
\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. The use even thousands of times of such an `\xintNewExpr`-generated `\formula` has no memory impact.

- count registers and `\numexpr`-essions *must* be prefixed by `\the` (or `\number`) when used inside `\xintexpr`. However, they may be used directly as arguments to most package macros, without being prefixed by `\the`. See Use of count registers. With release `1.09a` this functionality has been added to many macros of the integer only **xint** (with the cost of a small extra overhead; earlier, this overhead was added through the loading of **xintfrac**).

[8] 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.

- like a `\numexpr`, an `\xintexpr` is not directly printable, one uses equivalently `\xintthe \xintexpr` or `\xinttheexpr`. One may for example define:

```
\def\x {\xintexpr \a + \b \relax} \def\y {\xintexpr \x * \a \relax}
```

where `\x` could have been set up equivalently as `\def\x {( \a + \b )}` but the earlier method is better than with parentheses, as it allows `\xintthe\x`.

- sometimes one needs an integer, not a fraction or decimal number. The `round` function rounds to the nearest integer (half-integers are rounded towards $\pm\infty$), and `\xintexpr round(...)\relax` has an alternative syntax as `\xintnumexpr ... \relax`. There is also `\xintthenumexpr`. The rounding is applied to the final result only.

- there is also `\xintboolexpr ... \relax` and `\xinttheboolexpr ... \relax`. Same as regular expression but the final result is converted to `1` if it is not zero. See also `\xintifboolexpr` (subsection 25.9) and the discussion of the `bool` and `togl` functions in section 5. Here is an example of use:

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

This was obtained with the following input:

```
\xintNewBoolExpr \AssertionA[3]{ #1 & (#2|#3) }
\xintNewBoolExpr \AssertionB[3]{ #1 | (#2&#3) }
\xintNewBoolExpr \AssertionC[3]{ xor(#1,#2,#3) }
\begin{tabular}{ccc}
\xintFor #1 in {0,1} \do {%
  \xintFor #2 in {0,1} \do {%
    \xintFor #3 in {0,1} \do {%
    #1 AND (#2 OR #3) is \AssertionA {#1}{#2}{#3}&
    #1 OR (#2 AND #3) is \AssertionB {#1}{#2}{#3}&
    #1 XOR #2 XOR #3  is \AssertionC {#1}{#2}{#3}\\ }}}
\end{tabular}
```

- 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 `/b` even if `b=1` gets printed; this is the exception) which is the default format of the macros of the **xintfrac** package (hence of `\xintexpr`). To get a float format from the `\xintexpr` one needs something more:

`\xintFloat[60]{\xinttheexpr sqrt(2,60)\relax}`:
1.41421356237309504880168872420969807856967187537694807317668e0

The precision used by \xintfloatexpr must be set by \xintDigits, it can not be passed as an option to \xintfloatexpr.

\xintDigits:=48; \xintthefloatexpr 2^100000\relax:
9.99002093014384507944032764330033590980429139054e30102

Floats are quickly indispensable when using the power function (which can only have an integer exponent), as exact results will easily have hundreds of digits.

6 The \xintexpr math parser (II)

An expression is built with infix operators (including comparison and boolean operators) and parentheses, and functions. And there are two special branching constructs. 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. And -2^-10^-5*3 does (-((2^(-10))^(-5)))*3.

The characters used in the syntax should not have been made active. Use \xintexprSafeCatcodes, \xintexprRestoreCatcodes if need be (these commands must be exercised out of expansion only context). Apart from that infix operators may be of catcode letter or other, it does not matter, or even of catcode tabulation, math superscript, or math subscript. This should cause no problem. As an alternative to \xintexprSafeCatcodes one may also use \string inside the expression.

The A/B[N] notation is the output format of most **xintfrac** macros,[9] but for user input in an \xintexpr..\relax such a fraction should be written with the scientific notation AeN/B (possibly within parentheses) or *braces* must be used: {A/B[N]}. The square brackets are *not parsable* if not enclosed in braces together with the fraction.

Braces are also allowed in their usual rôle for arguments to macros (these arguments are thus not seen by the scanner), or to encapsulate *arbitrary* completely expandable material which will not be parsed but completely expanded and *must* return an integer or fraction possibly with decimal mark or in A/B[N] notation, but is not allowed to have the e or E. Braced material is not allowed to expand to some infix operator or parenthesis, it is allowed only in locations where the parser expects to find a number or fraction, possibly with decimal marks, but no e nor E.

One may use sub-\xintexpr-expressions nested within a larger one. It is allowed to alternate \xintfloatexpr-essions with \xintexpr-essions. Do not use \xinttheexpr inside an \xintexpr: this gives a number in A/B[n] format which requires protection by braces. Do not put within braces numbers in scientific notation.

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.

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[10] are the postfix operators: ! for the factorial, ? and : are two-fold way and three-fold way branching constructs. Then the

[9] this format is convenient for nesting macros; when displaying the final result of a computation one has \xintFrac in math mode, or \xintIrr and also \xintPRaw for inline text mode.

[10] in releases earlier than 1.09c, these postfix operators took precedence on a previous function name; the opposite now holds.

e and E of the scientific notation, the power, multiplication/division, addition/subtraction, comparison, and logical operators. At the lowest level: commas then parentheses.

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

- Functions are at the same top level of priority.

functions with one (numeric) argument floor, ceil, 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.

functions with one named 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 25.9)).

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[11] 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.

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

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

`|` logic operators: `\ifsometest10 & \ifsomeothertest10 | \ifsomethirdtest10`, etc... of course `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 then 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).

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`: the last two are integer-only and require the **xintgcd** package. Currently, `and` and `or` are left undefined, and the package uses the vocabulary `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

```
\xintthenumexpr (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 `e` and `E` of the scientific notation. They are treated as infix operators of highest priority. The decimal mark is scanned in a special direct way: in `1.12e3` first `1.12` is formed then only `e` is found. `1e3-1` is `999`.
- The power operator `^`.
- Multiplication and division `*`, `/`.
- Addition and subtraction `+`, `-`.
- Comparison operators `<`, `>`, `=`.
- Conjunction (logical and): `&`.
- Inclusive disjunction (logical or): `|`.
- The comma `,`. One can thus do `\xintthenumexpr 2^3,3^4,5^6\relax`: 8,81,15625.
- The parentheses.

7 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}`: 114738181166266556633273330008454586747025480423426102975889545437359089469703202762226470542663205834690270868221168133415250032403876277616895322211763429587203376221608860691585075716801971671071208769703353650737748777873778498781606749999798366581251723275215497054165956673849115333267485410756076697189062351899583237782636999811095323939932351899922205645878127014958776791431677354372538584459487155941215197416398666125896983737258716757394949435520170950261865801665199030718414432231169678376 96

1234/56789 with 1500 digits after the decimal point:

`\xintTrunc{1500}{1234/56789}\dots`: 0.021729560302171195125816619415731919914067865255595273732589057740055292398175703041081899663667259504481501699272746482593460001056542640300058109845216503196041486907675782281779922168025497895719241402384264558277131134550705242212400288788321682015883357692510873584673088098047157019845392593636091496592649985032312595749176777192766204722745602141259751008117769286305446477310746799556252091073975593865009068657662575498776171441652432689429290883797918610998608885523604923488703798270791878708904893553328989769145433094437302998820194051664935110672841571431087006 2

```
86428709785345753579038194016446847100670904576590536899751712479529
48634418637412174893025057669619116378171829051400799450597827043969
78288048741833805842680800859321347444047262674109422599447076018242
96958918100336332740495518498300727253517406539998943457359699941890
15478349680395851309232421771822007783197450210428075859761573544172
28688654492947577875997112116783179841166423074891264153269119019528
42980154607406363908503407350014967687404250823222807233795277254397
85874024899188223071369455352268925320044374790892602440613499093134
23374245012238285583475673105707091162020813890013911144763950765112
96201729208121291095106446671010230854566905562697001179805948335064
88932715842856891299371357129021465424642096180598355315289932909542
34094631002482875204705136558136258782510697494233038088362182817094
85992005494021729560302171195125816619415731919914067865255595273732
58905774005529239817570304108189966366 7...
```

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

```
\xintTrunc{200}{\xinttheexpr .99^-100\relax}\dots: 2.731999026429026003
84667172125783743550535164293857207083343057250824645551870534304481
43013784806140368055624765019253070342696854891531946166122710159206
719138403488514857479430864709639207317797930 3...
```

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

```
\xintAssign\xintBezout
           {\xintNum{\xinttheexpr 7^200-3^200\relax}}
           {\xintNum{\xinttheexpr 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)+1432
58949362763693185913068326832046547441686338771408915838167247899192
11328201191274624371580391777549768571912876931442406050669914563361
4320567769677489 1×(2^200-1)=1803403947125
```

The Euclide algorithm applied to 179,876,541,573 and 66,172,838,904:[12]

```
\xintTypesetEuclideAlgorithm {179876541573}{66172838904}
```

$$
\begin{aligned}
179876541573 &= 2 \times 66172838904 + 47530863765\\
66172838904 &= 1 \times 47530863765 + 18641975139\\
47530863765 &= 2 \times 18641975139 + 10246913487\\
18641975139 &= 1 \times 10246913487 + 8395061652\\
10246913487 &= 1 \times 8395061652 + 1851851835\\
8395061652 &= 4 \times 1851851835 + 987654312\\
1851851835 &= 1 \times 987654312 + 864197523\\
987654312 &= 1 \times 864197523 + 123456789\\
864197523 &= 7 \times 123456789 + 0
\end{aligned}
$$

$\sum_{n=1}^{500}(4n^2-9)^{-2}$ 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,

[12] this example is computed tremendously faster than the other ones, but we had to limit the space taken by the output.

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]}}
```

Computation of $2^{999,999,999}$ with 24 significant figures:
`\xintFloatPow[24] {2}{999999999}`: 2.306,488,000,584,534,696,558,06×10^{301,029,995}

To see more of **xint** in action, jump to the section 28 describing the commands of the **xintseries** package, especially as illustrated with the traditional computations of π and $\log 2$, or also see the computation of the convergents of e made with the **xintcfrac** package.

Note that 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.

8 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[14] 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.[15] 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.

[13] This number is typeset using the numprint package, with `\npthousandsep {,\hskip 1pt plus .5pt minus .5pt}`. But the breaking accross 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.

[14] this section was written before the **xintfrac** package; the author is not aware of another package allowing expandable computations with arbitrarily big fractions.

[15] the `\ReverseOrder` could be avoided in that circumstance, but it does play a crucial rôle here.

9 Expansions

Except for some specific macros dealing with assignments or typesetting, the bundle macros all work in expansion-only context. Such macros can also be used inside a `\csname...\endcsname`, and in an `\edef`. Furthermore they expand their arguments so that they can be arbitrarily chained.

By convention in this manual *ff*-expansion ("full first") is the process to expand repeatedly the first token seen until hitting against something not further expandable like an unexpandable TEX-primitive or an opening brace { or a (un-active) character. The type of expansion done almost systematically by the package macros to their arguments is usually the *ff*-expansion. However, when the argument is of a type a priori restricted to obey the TEX bound of 2147483647 (in absolute value), then it is fed into a `\numexpr..\relax` and the expansion will be a complete one, not limited to what comes first only.

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

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

The tex run creates a file `myfile-out.tex` containing the decimal representation of the integer quotient `2^{1000}/100!`.

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

expands (in two steps) and tells us that `[2^{1000}/100!]` has 144 digits. This is not so many, let us print them here: 114813249641507505482278393872551066259805517784186172883663478065826541894704737970419535798876630484358265060061503749531707793118627774829601.

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{\expandafter\expandafter\expandafter
                   \allowsplits #1\relax }%
% Expands twice before printing.
```

The `\printnumber` macro is not part of the package and would need additional thinking for more general use.[16] It may be used as `\printnumber {\xintQuo{\xintPow {2}{1000}}{\xintFac{100}}}`, or as `\printnumber\mynumber` if the macro `\mynumber` was previously defined via an `\edef`, as for example:

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

or as `\expandafter\printnumber\expandafter{\mynumber}`, if the macro `\mynumber` is defined by a `\newcommand` or a `\def` (see below item 3 for the underlying expansion issue; adding four `\expandafter`'s to `\printnumber` would allow to use it directly as `\printnumber\mynumber` with a `\mynumber` itself defined via a `\def` or `\newcommand`).

[16] 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).

Just to show off, let's print 300 digits (after the decimal point) of the decimal expansion of `0.7^{-25}`:[17]

```
\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 uses the macro `\xintTrunc` from package **`xintfrac`** wich extends to fractions the basic arithmetic operations defined for integers by **`xint`**. It also uses `\xinttheexpr` from package **`xintexpr`**, which allows to use standard notations. Note that 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 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 $\times 10^{35}$

Important points, to be noted, related to the expansion of arguments:

1. the macros *ff*-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. Unfortunately, after `\def\x {12}`, one can not use just `-\x` as input to one of the package macros: the rules above explain that the expansion will act only on the minus sign, hence do nothing. The only way is to use the `\xintOpp` macro, which replaces a number with its opposite.

   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...)

3. 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]`.

[17] the `\np` typesetting macro is from the `numprint` package.

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.

Release 1.07 has the \xintNewExpr command which 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.

10 Inputs and outputs

The core bundle constituents are **xint**, **xintfrac**, **xintexpr**, each one loading its predecessor. The base constituent **xint** only deals with integers, of arbitrary sizes, and apart from its macro \xintNum, the input format is rather strict.

> With release 1.09a, arithmetic macros of **xint** parse their arguments automatically through \xintNum. This means also that the arguments may already contain infix algebra with count registers, see Use of count registers.

Then **xintfrac** extends the scope to fractions: numerators and denominators are separated by a forward slash and may contain each an optional fractional part after the decimal mark (which has to be a dot) and a scientific part (with a lower case e).

The numeric arguments to the bundle macros may be of various types, extending in generality:

1. 'short' integers, *i.e.* less than (or equal to) in absolute value 2,147,483,647. I will refer to this as the 'TeX' or '\numexpr' limit. This is the case for arguments which serve to count or index something. It is also the case for the exponent in the power function and for the argument to the factorial function. The bounds have been (arbitrarily) lowered to 999,999,999 and 999,999 respectively for the latter cases.[18] When the argument exceeds the TeX bound (either positively or negatively), an error will originate from a \numexpr expression and it may sometimes be followed by a more specific error 'message' from a package macro.

[18] the float power function limits the exponent to the TeX bound, not 999999999, and it has a variant with no imposed limit on the exponent; but the result of the computation must in all cases be representable with a power of ten exponent obeying the TeX bound.

2. 'long' integers, which are the bread and butter of the package commands. They are signed integers with, for all pratical purposes, an illimited number of digits: most macros only require that the number of digits itself be less than the TEX and `\numexpr` bound of 2,147,483,647. Concretely though, multiplying out two 1000 digits numbers is already a longish operation.

3. 'fractions': they become available after having loaded the **xintfrac** package. A fraction has a numerator, a forward slash and then a denominator. Both can make use of scientific notation (with a lowercase `e`) and the dot as decimal mark. No separator for thousands. Except within `\xintexpr`-essions, spaces should be avoided.

The package macros first *ff*-expand their arguments: the first token of the argument is repeatedly expanded until no more is possible.

For those arguments which are constrained to obey the TEX bounds on numbers, they are systematically inserted inside a `\numexpr...\relax` expression, hence the expansion is then a complete one.

The allowed input formats for 'long numbers' and 'fractions' are:

1. the strict format is for some macros of **xint**. The number 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. There is a macro `\xintNum` which normalizes to this form an input having arbitrarily many minus and plus signs, followed by a string of zeros, then digits:
   `\xintNum {+-+-+----++-++----00000000009876543210}=-9876543210`
   Note that `-0` is not legal input and will confuse **xint** (but not `\xintNum` which even accepts an empty input).

2. the extended integer format is for the arithmetic macros of **xint** which automatically parse their arguments via `\xintNum`, and for the fractions serving as input to the macros of **xintfrac**: they are (or expand to) `A/B` (or just an integer `A`), where `A` and `B` will be automatically given to `\xintNum`. Each of `A` and `B` may be decimal numbers: with a decimal point and digits following it. Here is an example:
   `\xintAdd {+--0367.8920280/-++278.289287}{-109.2882/+270.12898}`
   Incidentally this evaluates to
   `=-129792033529284840/7517400124223726[-1]`
   `=-6489601676464242/3758700062111863 (irreducible)`
   `=-1.72655481129771093694248704898677881556360055242806...`
   where the second line was produced with `\xintIrr` and the next with `\xintTrunc {50}` to get fifty digits of the decimal expansion following the decimal mark. Scientific notation is accepted on input both for the numerators and denominators of fractions, and is produced on output by `\xintFloat`:
   `\xintAdd{10.1e1}{101.010e3}=101111/1[0]`
   This last example shows that fractions with a denominator equal to one, are generally printed as fraction. In math mode `\xintFrac` will remove such dummy denominators, and in inline text mode one has `\xintPRaw`.
   `\xintPRaw{\xintAdd{10.1e1}{101.010e3}}=101111`
   `\xintRaw{1.234e5/6.789e3}=1234/6789[2]`
   `\xintFloat[24]{1/66049}=1.51402746445820527184363e-5`

Of course, even with **xintfrac** is loaded, some macros by their nature can not accept fractions on input. Starting with release `1.05` most of them have also been extended to accept a fraction actually reducing to an integer. For example it used to be the case with the earlier releases that `\xintQuo {100/2}{12/3}` would not work (the macro `\xintQuo` computes a euclidean quotient). It now does, because its arguments are, after simplification, integers.

A number can start directly with a decimal point:

```
\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 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*!

Finally, after the decimal point there may be `eN` where `N` is a positive or negative number (obeying the TeX bounds on integers). This ‘e’ part (which must be in lowercase, except inside `\xintexpr`-essions) may appear both at the numerator and at the denominator.

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

Use of count registers: when an argument to a macro is said in the documentation to have to obey the TeX bound, this means that it is fed to a `\numexpr...\relax`, hence it is subjected to a complete expansion which must delivers 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 dealing with long numbers/fractions for arithmetic operations allow the use of count registers and even infix algebra with them 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*. The fraction symbol should be protected by braces else it will be used inside the `\numexpr` which does a rounded division. 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.

Outputs: 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 \xintRawWithZeros, 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 zeros (*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. Of course, the \xintFrac itself is not accepted as input to the package macros.

Direct user input of things such as 16000/289072[17] or 3[-4] is authorized. 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 zeros; and NO plus sign). This format with a power of ten represented by a number within square brackets is the output format used by (almost all) **xintfrac** macros dealing with fractions. 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 chaining package macros without too much speed impact, as they always output computation results in the A/B[n] form.

IMPORTANT!

All computations done by **xintfrac** on fractions 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 always converted into exact internal representations.

Floating point evaluations are done with special macros containing 'Float' in their names, or inside \xintthefloatexpr-essions.

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 expected to, as a general rule (with possible exceptions related to the allowed use of braces, see the documentation) be completely harmless, and even recommended for making the source more legible.

Syntax such as \xintMul\A\B is accepted and equivalent[19] to \xintMul {\A}{\B}. Or course \xintAdd\xintMul\A\B\C does not work, the product operation must be put within braces: \xintAdd{\xintMul\A\B}\C. It would be nice to have a functional form

[19] see however near the end of this later section for the important difference when used in contexts where TeX expects a number, such as following an \ifcase or an \ifnum.

\add(x,\mul(y,z)) but this is not provided by the package.[20] Arguments must be either within braces or a single control sequence.

Note that - and + may serve only as unary operators, on *explicit* numbers. They can not serve to prefix macros evaluating to such numbers, *except inside an* \xintexpr*-ession.*

11 More on fractions

With package **xintfrac** loaded, the routines \xintAdd, \xintSub, \xintMul, \xintPow, \xintSum, \xintPrd are modified to allow fractions on input,[21] [22] [23] [24] and produce on output a fractional number f=A/B[n] where A and B are integers, with B positive, and n is a signed "small" integer (*i.e* less in absolute value than 2^{31}-9). 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 zeros (*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).[25]

The \xintiAdd, \xintiSub, \xintiMul, \xintiPow, \xintiSum, \xintiPrd, etc... are the original un-modified integer-only versions. They have less parsing overhead.

1.09a: the original now also use \xintNum →

The macro \xintRaw prints the fraction directly from its internal representation in A/B[n] form. To convert the trailing [n] into explicit zeros either at the numerator or the denominator, use \xintRawWithZeros. In both cases the B is printed even if it has value 1. The macro \xintPRaw will not print the [n] if n=0 and will not print the /B if B=1.

Conversely (sort of), the macro \xintREZ puts all powers of ten into the [n] (REZ stands for remove zeros). Here also, the B is printed even if it has value 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.

The macro \xintTrunc{N}{f} prints[26] the decimal expansion of f with N digits after

---

[20] yes it is with the 1.09a \xintexpr, \xintexpr add(x,mul(y,z))\relax.

[21] of course, the power function does not accept a fractional exponent. Or rather, does not expect, and errors will result if one is provided.

[22] macros \xintiAdd, \xintiSub, \xintiMul, \xintiPow, \xintiSum, \xintiPrd are the original ones dealing only with integers. They are available as synonyms, also when **xintfrac** is not loaded.

[23] also \xintCmp, \xintSgn, \xintOpp, \xintAbs, \xintMax, \xintMin are extended to fractions and have their integer-only initial synonyms.

[24] and \xintFac, \xintQuo, \xintRem, \xintDivision, \xintGeq, \xintFDg, \xintLDg, \xintOdd, \xintMON, \xintMMON all accept a fractional input as long as it reduces to an integer.

[25] 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.

[26] 'prints' does not at all mean that this macro is designed for typesetting; I am just using the verb here in analogy to the effect of the functioning of a computing software in console mode. The package does not provide any 'printing' facility, besides its rudimentary \xintFrac and \xintFwOver math-mode only macros. To deal with really long numbers, some macros are necessary as TeX by default will print a long number on a single line extending beyond the page limits. The \printnumber command used in this documentation is just one way to address this problem, some other method should be used if it is important that digits occupy the same width always.

the decimal point.[27] Currently, it does not verify that N is non-negative and strange things could happen with a negative N. A negative f is no problem, needless to say. When the original fraction is negative and its truncation has only zeros, it is printed as -0.0...0, with N zeros following the decimal point:

\xintTrunc {5}{\xintPow {-13}{-9}}=-0.00000

\xintTrunc {20}{\xintPow {-13}{-9}}=-0.00000000009429959537

The output always contains a decimal point (even for N=0) followed by N digits, except when the original fraction was zero. In that case the output is 0, with no decimal point.

\xintTrunc {10}{\xintSum {{1/2}{1/3}{1/5}{-31/30}}}=0

The macro \xintiTrunc{N}{f} is like \xintTrunc{N}{f} followed by multiplication by 10^N. Thus, it outputs an integer in a format acceptable by the integer-only macros. To get the integer part of the decimal expansion of f, use \xintiTrunc{0}{f}:

\xintiTrunc {0}{\xintPow {1.01}{100}}=2

\xintiTrunc {0}{\xintPow{0.123}{-10}}=1261679032

See also the documentations of \xintRound, \xintiRound and \xintFloat.

12 \ifcase, \ifnum, ... constructs

When using things such as \ifcase \xintSgn{\A} one has 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} 0\or OK\else ERROR\fi   ---> gives OK
```

Release 1.07 provides the expandable \xintSgnFork which chooses one of three branches according to whether its argument expands to -1, 0 or 1. This, rather than the corresponding \ifcase, should be used when such a fork is needed as argument to one of the package macros.

Release 1.09a has \xintifSgn which does not restrict its first argument to be -1, 0, 1: the argument, which may be also a count register will be first replaced by its sign. There are also \xintifZero, \xintifNotZero, \xintifGt, \xintifLt, \xintifEq.

13 Multiple outputs

Some macros have an output consisting of more than one number, each one is then 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, ... the next two sections explain ways to deal, expandably or not, with such outputs.

[27] the current release does not provide a macro to get the period of the decimal expansion.

See the subsection 22.57 for a rare example of a bundle macro which may return an empty string, or a number prefixed by a chain of zeros. This is the only situation where a macro from the package **xint** may output something which could require parsing through `\xintNum` before further processing by the other (integer-only) package macros from **xint**.

14 Assignments

It might not be necessary to maintain at all times complete expandability. For example why not allow oneself the two definitions `\edef\A {\xintQuo{100}{3}}` and `\edef\B {\xintRem {100}{3}}`. A special syntax is provided to make these things more efficient, as the package provides `\xintDivision` 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:->11481324964150750548227839387255106625980551 77841861728836634780658265418947047379704195357988766304843582650600 61503749531707793118627774829601` and `\meaning\B: macro:->5493629452133 98322513812878622391280734105004984760505953218996123132766490228838 81328787024445820751296031520410548049646250831385676526243868372056 68069376`.

Another example (which uses a macro 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.

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

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

When one does not know in advance the number of tokens, one can use `\xintAssignArray` or its synonym `\xintDigitsOf`:

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

This defines `\Out` to be macro with one parameter, `\Out{0}` gives the size N of the array and `\Out{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 `\Out` 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\Out
\cnta = 1
\cntb = 0
\loop
\advance \cntb \xintiSqr{\Out{\cnta}}
\ifnum \cnta < \Out{0}
\advance\cnta 1
\repeat
```

```
|2^{100}| (=\xintiPow {2}{100}) has \Out{0} digits and the sum of
their squares is \the\cntb. These digits are, from the least to
the most significant: \cnta = \Out{0}
\loop \Out{\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.

We used a group in order to release the memory taken by the `\Out` array: indeed internally, besides `\Out` itself, additional macros are defined which are `\Out0`, `\Out00`, `\Out1`, `\Out2`, ..., `\OutN`, where `N` is the size of the array (which is the value returned by `\Out{0}`; the digits are parts of the names not arguments).

The command `\xintRelaxArray\Out` sets all these macros to `\relax`, but it was simpler to put everything withing a group.

Needless to say `\xintAssign`, `\xintAssignArray` and `\xintDigitsOf` do not do any check on whether the macros they define are already defined.

In the example above, we deliberately broke all rules of complete expandability, but had we wanted to compute the sum of the digits, not the sum of the squares, we could just have written:

`\xintiSum{\xintiPow{2}{100}}=115`

Indeed, `\xintiSum` is usually used as in
`\xintiSum{{123}{-345}{\xintFac{7}}{\xintiOpp{\xintRem{3347}{591}}}}=4426`
but in the example above each digit of 2^{100} is treated as would have been a summand enclosed within braces, due to the rules of TeX for parsing macro arguments.

Note that `{-\xintRem{3347}{591}}` is not a valid input, because the expansion will apply only to the minus sign and leave unaffected the `\xintRem`. So we used `\xintiOpp` which replaces a number with its opposite.

As a last example with `\xintAssignArray` here is one line extracted from the source code of the **xintgcd** macro `\xintTypesetEuclideAlgorithm`:

`\xintAssignArray\xintEuclideAlgorithm {#1}{#2}\to\U`

This is done inside a group. After this command `\U{1}` contains the number `N` of steps of the algorithm (not to be confused with `\U{0}=2N+4` which is the number of elements in the `\U` array), and the GCD is to be found in `\U{3}`, a convenient location between `\U{2}` and `\U{4}` which are (absolute values of the expansion of) the initial inputs. Then follow `N` quotients and remainders from the first to the last step of the algorithm. The `\xintTypesetEuclideAlgorithm` macro organizes this data for typesetting: this is just an example of one way to do it.

15 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 with 1.06, and \xintApplyUnbraced, new with 1.06b.

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
\xintiSum{\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.

Of course, it would be nicer to do \edef\z{\xintiPow {2}{100}}, and then use \z in place of \xintiPow {2}{100} everywhere as this would spare the CPU some repetitions.

16 A new kind of for loop

As part of the utilities coming with the **xint** package, there is a new kind of for loop, \xintFor. Check it out (subsection 23.11).

17 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:use_xintthe!
```

```
\xintError:bigtroubleahead
\xintError:unknownfunction
```

18 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`.[28]
- 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 `\xintthenumexpr 4/2\relax`.

19 Package namespace

Inner macros of **xint**, **xintfrac**, **xintexpr**, **xintbinhex**, **xintgcd**, **xintseries**, and **xintcfrac** all begin either with `\XINT_` or with `\xint_`.[29] 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.

20 Loading and usage

```
Usage with LaTeX: \usepackage{xint}
```

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

[29] 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 starting with `\XINTinFloat..` or `\XINTinfloat...`

```
                  \usepackage{xintfrac}   % (loads xint)
                  \usepackage{xintexpr}   % (loads xintfrac)

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

Usage with TeX:   \input xint.sty\relax
                  \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 mecanism 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.

Furthermore, **xintfrac**, **xintbinhex**, and **xintgcd** check for the previous loading of **xint**, and will try to load it if this was not already done. Similarly **xintseries**, **xintcfrac** and **xintexpr** do the necessary loading of **xintfrac**. Each package will refuse to be loaded twice.

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 actual use of the macros, note that when feeding them with negative numbers the minus sign must have category code other, as is standard. Similarly the slash used for inputting fractions must be of category other, as usual. And the square brackets also must be of category code other, if used on input. The 'e' of the scientific notation must be of category code letter. All of that is relaxed when inside an `\xintexpr`-ession (but arguments to macros which have been inserted in the expression must obey the rules, as it is the macro and not the parser which will get the tokens). In an `\xintexpr`-ession, the scientific 'e' may be 'E'.

The components of the **xint** bundle presuppose that the usual `\space` and `\empty` macros are pre-defined, which is the case in Plain TEX as well as in LATEX.

Lastly, the macros `\xintRelaxArray` (of **xint**) and `\xintTypesetEuclideAlgorithm` and `\xintTypesetBezoutAlgorithm` (of **xintgcd**) use `\loop`, both Plain and LATEX incarnations are compatible. `\xintTypesetBezoutAlgorithm` also uses the `\endgraf` macro.

21 Installation

```
Run tex or latex on xint.dtx.

This will extract the style files xint.sty, xintfrac.sty, xintexpr.sty,
xintbinhex.sty, xintgcd.sty, xintseries.sty, xintcfrac.sty (and xint.ins).

Files with the same names and in the same repertory will be overwritten.
The tex (not latex) run will stop with the complaint that it does not
understand \NeedsTeXFormat, but the style files will already have been
extracted by that time.

Alternatively, run tex or latex on xint.ins if available.

To get xint.pdf run pdflatex thrice on xint.dtx

          xint.sty |
      xintfrac.sty |
      xintexpr.sty |
    xintbinhex.sty | --> TDS:tex/generic/xint/
       xintgcd.sty |
    xintseries.sty |
     xintcfrac.sty |
          xint.dtx   --> TDS:source/generic/xint/
          xint.pdf   --> TDS:doc/generic/xint/

It may be necessary to then refresh the TeX installation filename
database.
```

22 Commands of the xint package

In the description of the macros {N} (or also {M}) stands (except if mentioned otherwise) for a (long) number within braces or for a control sequence possibly within braces and *ff*-expanding to such a number (without the braces!), or for material within braces which *ff*-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 zeros, followed by digits.

The letter 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 in section 10.

Some of these macros are extended by **xintfrac** to accept fractions on input, and, generally, to output a fraction. This will be mentioned and the original integer only macro \xintAbc remains then available under the name \xintiAbc. Even the original integer-only macros may now accept fractions on input as long as they are integers in disguise; they still produce on output integers without any forward slash mark nor trailing [n]. On the other hand macros such as \xintAdd will output fractions A/B[n], with B present even if its value is one. To remove this unit denominator and convert the [n] part into explicit zeros, one has \xintNum (if one is certain to deal with an integer; see also \xintPRaw). This is mandatory when the computation result is fetched into a context where TeX expects a number (assuming it does not exceed 2^31). See the also the **xintfrac** documentation for more information on how macros of **xint** are modified after loading **xintfrac** (or **xintexpr**).

Package **xint** also provides some general macro programming or token manipulation utilities (expandable as well as non-expandable), which are described in the next section (section 23).

Contents

22.1 \xintRev

\xintRev{N} will revert the order of the digits of the number, keeping the optional sign.

Leading zeros 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

22.2 \xintLen

\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.

22.3 \xintDigitsOf

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.

22.4 \xintNum

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

22.5 \xintSgn

\xintSgn{N} returns 1 if the number is positive, 0 if it is zero and -1 if it is negative. Extended by **xintfrac** to fractions.

22.6 \xintOpp

\xintOpp{N} returns the opposite -N of the number N. Extended by **xintfrac** to fractions.

22.7 \xintAbs

\xintAbs{N} returns the absolute value of the number. Extended by xintfrac to fractions.

22.8 \xintAdd

\xintAdd{N}{M} returns the sum of the two numbers. Extended by xintfrac to fractions.

22.9 \xintSub

\xintSub{N}{M} returns the difference N-M. Extended by xintfrac to fractions.

22.10 \xintCmp

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

22.11 \xintEq

New with release 1.09a.

\xintEq{N}{M} returns 1 if N=M, 0 otherwise. Extended by xintfrac to fractions.

22.12 \xintGt

New with release 1.09a.

\xintGt{N}{M} returns 1 if N>M, 0 otherwise. Extended by xintfrac to fractions.

22.13 \xintLt

New with release 1.09a.

\xintLt{N}{M} returns 1 if N<M, 0 otherwise. Extended by xintfrac to fractions.

22.14 \xintIsZero

New with release 1.09a.

\xintIsZero{N} returns 1 if N=0, 0 otherwise. Extended by xintfrac to fractions.

22.15 \xintNot

New with release 1.09c.

\xintNot is a synonym for \xintIsZero.

22.16 \xintIsNotZero

New with release 1.09a.

\xintIsNotZero{N} returns 1 if N<>0, 0 otherwise. Extended by xintfrac to fractions.

22.17 \xintIsOne

New with release 1.09a.

\xintIsOne{N} returns 1 if N=1, 0 otherwise. Extended by xintfrac to fractions.

22.18 \xintAND

New with release 1.09a.

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

22.19 \xintOR

New with release 1.09a.

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

22.20 \xintXOR

New with release 1.09a.

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

22.21 \xintANDof

New with release 1.09a.

\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 *ff*-expanded first (each item also is *ff*-expanded). Extended by xintfrac to fractions.

22.22 \xintORof

New with release 1.09a.

\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 *ff*-expanded first. Extended by xintfrac to fractions.

22.23 \xintXORof

New with release 1.09a.

\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 *ff*-expanded first. Extended by xintfrac to fractions.

22.24 \xintGeq

\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 (starting with release 1.07). Please note that the macro compares *absolute values*.

22.25 \xintMax

\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.

22.26 \xintMaxof

New with release 1.09a.

\xintMaxof{{a}{b}{c}...} returns the maximum. The list argument may be a macro, it is *ff*-expanded first. Extended by **xintfrac** to fractions.

22.27 \xintMin

\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.

22.28 \xintMinof

New with release 1.09a.

\xintMinof{{a}{b}{c}...} returns the minimum. The list argument may be a macro, it is *ff*-expanded first. Extended by **xintfrac** to fractions.

22.29 \xintSum

\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.

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

\xintiSum{1234567890}=45

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

22.30 \xintMul

Modified in release 1.03.

\xintMul{N}{M} returns the product of the two numbers. Starting with release 1.03 of **xint**, the macro checks the lengths of the two numbers and then activates its algorithm with the best (or at least, hoped-so) choice of which one to put first. This makes the macro a bit slower for numbers up to 50 digits, but may give substantial speed gain when one of the number has 100 digits or more. Extended by **xintfrac** to fractions.

22.31 \xintSqr

\xintSqr{N} returns the square. Extended by **xintfrac** to fractions.

22.32 \xintPrd

\xintPrd{⟨*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 product of all these numbers is returned.

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

\xintiPrd{123456789123456789}=131681894400

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

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

=\xintiPrd {{\xintiPow {2}{200}}{\xintiPow {3}{100}}{\xintiPow {7}{100}}}
=26787279316615775757662795170075484023247402663740153489744596148154264129654994900004440072407657271300001653120764065456211801435719940159033435392440282124389668222489278629880843827161333 76

Extended by xintfrac to fractions.

With xintexpr, the above would be coded simply as

\xintNum {\xinttheexpr 2^200*3^100*7^100\relax }

(\xintNum to print an integer, not a fraction).

22.33 \xintPow

\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>999999999, then an error is raised. 2^999999999 has 301,029,996 digits; each exact multiplication of two one thousand digits numbers already takes a few seconds, so needless to say this bound is completely irrealistic. Already 2^9999 has 3,010 digits,[30] so I should perhaps lower the bound to 99999.

Extended by xintfrac to fractions (\xintPow) and also to floats (\xintFloatPow). Of course, 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}{9999}=9.975e3009 and \xintFloatPow[4]{2}{999999999}=2.306e301029995.

22.34 \xintSgnFork

New with release 1.07. See also \xintifSgn.

\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 should be prefixed by \the and a \numexpr...\relax also should 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.

[30] on my laptop \xintiPow{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 circa one hundredth of a second (1.08b). This is done without log/exp which are not (yet?) implemented in xintfrac. The LaTeX3 l3fp does this with log/exp and is ten times faster (16 figures only).

22.35 \xintifSgn

New with release 1.09a.

Same as \xintSgnFork except that the first argument may expand to an integer (or a fraction if xintfrac is loaded), it is its sign which decides which of the three branches is taken. This first argument may be a count register, with no \the or \number prefix.

22.36 \xintifZero

New with release 1.09a.

\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.

22.37 \xintifNotZero

New with release 1.09a.

\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.

22.38 \xintifTrue

New with release 1.09c.

\xintifTrue{⟨N⟩}{⟨YES⟩}{⟨NO⟩} is a synonym for \xintifNotZero.

22.39 \xintifEq

New with release 1.09a.

\xintifEq{⟨A⟩}{⟨B⟩}{⟨YES⟩}{⟨NO⟩} checks equality of its two first arguments (which may be macros but must expand to numbers or fractions, if xintfrac is loaded) and does the YES or the NO branch.

22.40 \xintifGt

New with release 1.09a.

\xintifGt{⟨A⟩}{⟨B⟩}{⟨YES⟩}{⟨NO⟩} checks if $A > B$ and in that case executes the YES branch.

22.41 \xintifLt

New with release 1.09a.

\xintifLt{⟨A⟩}{⟨B⟩}{⟨YES⟩}{⟨NO⟩} checks if $A < B$ and in that case executes the YES branch.

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

22.42 \xintFac

\xintFac{x} returns the factorial. It is an error if the argument is negative or at least 10^6. It is not recommended to launch the computation of things such as 100000!, if you need your computer for other tasks. Note that the argument is of the x type, it must obey the TeX bounds, but on the other hand may involve count registers and even arithmetic operations as it will be completely expanded inside a \numexpr.

With xintfrac loaded, the macro also accepts a fraction as argument, as long as this fraction turns out to be an integer: \xintFac {66/3}=1124000727777607680000.

22.43 \xintDivision

\xintDivision{N}{M} returns {quotient Q}{remainder R}. This is euclidean division: N = QM + R, $0 \leq$ 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 (even if N vanishes) and returns {0}{0}.

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.

22.44 \xintQuo

\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.

22.45 \xintRem

\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.

22.46 \xintFDg

\xintFDg{N} returns the first digit (most significant) of the decimal expansion.

22.47 \xintLDg

\xintLDg{N} returns the least significant digit. When the number is positive, this is the same as the remainder in the euclidean division by ten.

22.48 \xintMON, \xintMMON

New in version 1.03.

\xintMON{N} returns (-1)^N and \xintMMON{N} returns (-1)^{N-1}.

\xintMON {-280914019374101929}=-1, \xintMMON {-280914019374101929}=1

22.49 \xintOdd

\xintOdd{N} is 1 if the number is odd and 0 otherwise.

22.50 \xintiSqrt, \xintiSquareRoot

New with 1.08.

\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

\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.

22.51 \xintInc, \xintDec

New with 1.08.

\xintInc{N} is N+1 and \xintDec{N} is N-1. These macros remain integer-only, even with xintfrac loaded.

22.52 \xintDouble, \xintHalf

New with 1.08.

\xintDouble{N} returns 2N and \xintHalf{N} is N/2 rounded towards zero. These macros remain integer-only, even with xintfrac loaded.

22.53 \xintDSL

\xintDSL{N} is decimal shift left, *i.e.* multiplication by ten.

22.54 \xintDSR

\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.

22.55 \xintDSH

\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 of course), and for a non-negative N this is thus the same as the quotient from the euclidean division by 10^x.

22.56 \xintDSHr, \xintDSx

New in release 1.01.

\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.

\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
```

22.57 \xintDecSplit

This has been modified in release 1.01.

\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 zeros 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 zeros 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:->.
```

22.58 \xintDecSplitL

\xintDecSplitL{x}{N} returns the first piece after the action of \xintDecSplit.

22.59 \xintDecSplitR

\xintDecSplitR{x}{N} returns the second piece after the action of \xintDecSplit.

23 Commands (utilities) of the xint package

The completely expandable utilities come first, up to and including \xintSeq (which is listed here because it generates sequences of short integers using \numexpr, thus does not make use of the big integers macros of xint).

Contents

23.1 \xintReverseOrder

`\xintReverseOrder{⟨list⟩}` does not do any expansion of its argument and just reverses the order of the tokens in the ⟨*list*⟩.[31] Brace pairs encountered are removed once and the enclosed material does not get reverted. Spaces are gobbled.

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

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

23.2 \xintRevWithBraces

New in release 1.06.

`\xintRevWithBraces{⟨list⟩}` first does the expansion of its argument (which thus may be macro), 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 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 }
```

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

23.3 \xintLength

`\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 *ff*-expands its argument.

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

23.4 \xintCSVtoList

New with release 1.06.

`\xintCSVtoList{a,b,c...,z}` returns `{a}{b}{c}...{z}`. A *list* is in this manual the word we use to describe a succession or tokens where braced tokens count as one thing.

[31] the argument is not a token list variable, just a ⟨*list*⟩ of tokens.

The argument to \xintCSVtoList may be a macro which is first expanded fully. This means that the first item before the comma, if it is itself a macro, will be expanded which may or may not be a good thing. A space at the start of the first item will stop the expansion and be gobbled.

Contiguous spaces, tab characters, or other blanc spaces (empty lines not allowed) are collapsed by TEX into single spaces. *No attempt* is made to get rid of such spaces either before or after the commas, as priority has been given to the speed of the conversion (but without impacting the input stack size).

```
\xintCSVtoList {1,2,a , b ,c d,x,y }->{1}{2}{a }{ b }{c d}{x}{y }
\def\y{a,b,c,d,e} \xintCSVtoList\y->{a}{b}{c}{d}{e}
```

The macro \xintCSVtoListNoExpand does the same job without the initial expansion.

```
\xintCSVtoListNoExpand{\a,\b,\c,\d,\e}->{\a}{\b}{\c}{\d}{\e}
```

23.5 \xintNthElt

New in release 1.06. With 1.09b negative indices count from the tail.

\xintNthElt{x}{⟨*list*⟩} gets (expandably) the xth element of the ⟨*list*⟩, which may be a macro: the list argument is first expanded. The seeked element is returned with one pair of braces removed (if initially present).

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

\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}

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.

23.6 \xintListWithSep

New with release 1.04.

\xintListWithSep{sep}{⟨*list*⟩} inserts the given separator sep in-between all elements of the given list: this separator may be a macro but will not be expanded. The second argument also may be itself a macro: it is expanded as usual, *i.e.* fully for what comes first. Applying \xintListWithSep removes one level of top braces to each list constituent. 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 removing one-level of brace pairs from each ot the top-level braced material 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

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

23.7 \xintApply

New with release 1.04.

\xintApply{\macro}{⟨*list*⟩} expandably applies the one parameter command \macro to each item in the ⟨*list*⟩ given as second argument and return a new list with these outputs: each item is given one after the other as parameter to \macro which is expanded (as usual, *i.e.* fully for what comes first), and the result is braced. On output, a new list with these braced results (if \macro is defined to start with a space, the space will be gobbled and the \macro will not be executed; \macro is allowed to have its own arguments, the list items will serve as last arguments to the macro.).

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 is not necessarily compatible with expansion only contexts: \xintApply will try to expand it, but 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
```

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

23.8 \xintApplyUnbraced

New in release 1.06b.

\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.

```
\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
```

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

23.9 \xintSeq

New with release 1.09c.

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

The current implementation is only for (short) integers; possibly, a future variant could allow big integers and fractions, although one already has similar functionality using \xintApply with an affine transformation to post-process an integer sequence.

\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

\xintiSum{\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 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.

IMPORTANT!

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 \xintiSum{\xintSeq [1]{0}{5000}} returns the correct value 12502500.

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

23.10 \xintApplyInline

1.09a, enhanced in 1.09c to be usable within alignments, and corrected in 1.09d for a problem related to spaces at the very end of the list parameter.

\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.

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`.

23.11 `\xintFor`, `\xintFor*`

New with `1.09c`. The macro `\xintFor*` is corrected in `1.09d` to fix a bug when a space token was at the very end of the list. The new version *ff*-expands the (unbraced) items.

`\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`, `#3`, `#4` can be used to represent the items for up to four levels of nested loops. Here is an example:

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

The use of either `#1`, `#2`, `#3`, or `#4` to denote the item is mandatory, but one does not have to use necessarily `#1` as the first one. Notice that contrarily to what happens in loops where the item is represented by a macro, here it is truly exactly like in a macro definition. This may avoid the user quite a few troubles with `\expandafter`s or other `\edef`/`\noexpand`s which one encounters at times when trying to do things with the `\@for` loop of LaTeX. For example if above, rather than the package's `\xintiPrd` one had a macro which does not expand its arguments, or perhaps it does, but not the fourth one, etc. . .

Allowing `#5`, etc. . . , would have meant more lines of code and also some more tokens inside the already existing code, I decided to postpone it to later, if people are interested (on the basis that someone will actually read these lines, one day; I mean someone besides me). One may naturally put multiple `\xintFor` loops one after the other inside a primary one. The replacement text can do quite arbitrary things in-between such sub-loops (if any).

The non-starred variant `\xintFor` deals with comma separated values (no effort is done to remove the spaces before and after the commas) and the comma separated list may be a macro which is just expanded once.

The starred variant `\xintFor*` deals with token lists and *ff*-expands each *unbraced* list item. This makes it easy to concatenate various list macros `\x`, `\y`, ... as if `\x` expands to `123` and `\y` expands to `456` then `{\x\y}` as argument to `\xintFor*` has the same effect as `{123456}`. Spaces at the start, end, or in-between items are gobbled (but naturally not the spaces which may be inside *braced* items).

The `\xintFor` loops may be used inside alignments or other contexts with the replacement text closing 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 #2 in {A,B,C} \do {%
      #2:\xintFor* #1 in {abcde} \do {&($ #1 \to #2 $)}\\ }%
\end{tabular}
```

It is not an expandable macro and has some strong cousinage to \xintApplyInline. When inserted inside a macro for later execution the # characters must be doubled. 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.

The advantages of using macro parameters rather than macros for the items reveals itself in certain circumstances which may concern more the macro programmer than the general LaTeX (or TeX) user. On the other hand the capacity of \xintFor to survive in contexts such as alignments could prove of more general interest.

23.12 \xintForpair, \xintForthree, \xintForfour

New in 1.09c and in experimental status.

This is experimental and subjected to change. The syntax is illustrated in this example:

```
\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 {%
        $\left(\begin{tabular}{cc}
          #1 & #3\\
          #4 & #2\\
        \end{tabular}\right)$&}\\\noalign{\vskip1\jot}}%
\end{tabular}
```

$$\begin{pmatrix} \text{A} & \text{X} \\ \text{x} & \text{a} \end{pmatrix} \begin{pmatrix} \text{A} & \text{Y} \\ \text{y} & \text{a} \end{pmatrix} \begin{pmatrix} \text{A} & \text{Z} \\ \text{z} & \text{a} \end{pmatrix}$$

$$\begin{pmatrix} \text{B} & \text{X} \\ \text{x} & \text{b} \end{pmatrix} \begin{pmatrix} \text{B} & \text{Y} \\ \text{y} & \text{b} \end{pmatrix} \begin{pmatrix} \text{B} & \text{Z} \\ \text{z} & \text{b} \end{pmatrix}$$

$$\begin{pmatrix} \text{C} & \text{X} \\ \text{x} & \text{c} \end{pmatrix} \begin{pmatrix} \text{C} & \text{Y} \\ \text{y} & \text{c} \end{pmatrix} \begin{pmatrix} \text{C} & \text{Z} \\ \text{z} & \text{c} \end{pmatrix}$$

Only #1#2, #2#3, #3#4 are accepted. One can nest with \xintFor, for disjoint sets of macro parameters of course. There is also \xintForthree (with #1#2#3 or #2#3#4) and \xintForfour (only with #1#2#3#4).

These three macros `\xintForpair`, `\xintForthree` and `\xintForfour` are to be considered in experimental status, and may be removed or substantially modified at some later stage. Actually they may be more of interest for some programming tasks, where having macro parameters rather than macros may be very helpful in certain circumstances, than for use by a general audience.

23.13 \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 be the complete expansions of the successive braced things found on the left of `\to`.

A 'full' expansion is first applied first 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 `\edef` as the complete expansion of the material between `\xintAssign` and `\to`.

```
\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}}`)

This macro uses various `\edef`'s, thus is incompatible with expansion-only contexts.

23.14 \xintAssignArray

Changed in release 1.06 to let the defined macro pass its argument through a `\numexpr...\relax`.

`\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 completely expanded `x`th 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.

23.15 \xintRelaxArray

`\xintRelaxArray\myArray` sets to `\relax` all macros which were defined by the previous `\xintAssignArray` with `\myArray` as array name.

24 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.

f stands for an integer or a fraction (see section 10 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.

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), \xintPRaw which does not print the [n] if n=0 or the B if B=1. Use \xintNum (or \xintPRaw if simplification is not needed) for fractions a priori known to simplify to integers: \xintNum {\xintAdd {2}{3}} gives 5 whereas \xintAdd {2}{3} returns 5/1[0]. Some macros (among them \xintiTrunc, \xintiRound, and \xintFac) already produce integers on output.

Contents

24.1 \xintLen

The original macro is extended to accept a fraction on input.

\xintLen {201710/298219}=11, \xintLen {1234/1}=4, \xintLen {1234}=4

24.2 \xintRaw

New with release 1.04.

MODIFIED IN 1.07.

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]

24.3 \xintPRaw

New in 1.09b.

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.

24.4 \xintNumerator

This returns the numerator corresponding to the internal representation of a fraction, with positive powers of ten converted into zeros 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.

24.5 \xintDenominator

This returns the denominator corresponding to the internal representation of the fraction:[32]

\xintDenominator {178000/25600000[17]}=25600000

[32] recall that the [] construct excludes presence of a decimal point.

`\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`.

24.6 \xintRawWithZeros

New name in 1.07 (former name `\xintRaw`).

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`

24.7 \xintREZ

This command normalizes a fraction by removing the powers of ten from its numerator and denominator:

`\xintREZ {178000/25600000[17]}=178/256[15]`
`\xintREZ {1780000000000e30/2560000000000e15}=178/256[15]`

As shown by the example, it does not otherwise simplify the fraction.

24.8 \xintFrac

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.)

24.9 \xintSignedFrac

New with release 1.04.

This is as `\xintFrac` except that a negative fraction has the sign put in front, not in the numerator.

`\[\xintFrac {-355/113}=\xintSignedFrac {-355/113}\]`

$$\frac{-355}{113} = -\frac{355}{113}$$

24.10 \xintFwOver

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). $\xint-FwOver {178.000/25600000}$ 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.

24.11 \xintSignedFwOver

New with release 1.04.

This is as \xintFwOver except that a negative fraction has the sign put in front, not in the numerator.

```
\[\xintFwOver {-355/113}=\xintSignedFwOver {-355/113}\]
```

$$\frac{-355}{113} = -\frac{355}{113}$$

24.12 \xintIrr

MODIFIED IN 1.08.

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}}.

24.13 \xintJrr

MODIFIED IN 1.08.

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}/\xintiPrdExpr {\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.

24.14 \xintTrunc

\xintTrunc{x}{f} returns the start of the decimal expansion of the fraction f, with x digits after the decimal point. The argument x should be non-negative. When x=0, the integer part of f results, with an ending decimal point. Only when f evaluates to zero does \xintTrunc not print a decimal point. When f is not zero, the sign is maintained in the output, also when the digits are all zero.

\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. The identity \xintTrunc {x}{-f}=-\xintTrunc {x}{f} holds.[33]

24.15 \xintiTrunc

\xintiTrunc{x}{f} returns the integer equal to 10^x times what \xintTrunc{x}{f} would return.

\xintiTrunc {16}{-803.2028/20905.298}=-384210165289200
\xintiTrunc {10}{\xintPow {-11}{-11}}=0
\xintiTrunc {12}{\xintPow {-11}{-11}}=-3

Differences between \xintTrunc{0}{f} and \xintiTrunc{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 zeros.)

24.16 \xintRound

New with release 1.04.

\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:

-0.000000000003504938994813924976040033131625985563 70...

24.17 \xintiRound

New with release 1.04.

\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

[33] Recall that -\macro is not valid as argument to any package macro, one must use \xintOpp{\macro} or \xintiOpp{\macro}, except inside \xinttheexpr...\relax.

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 zeros.)

24.18 \xintFloor

New with release 1.09a.

\xintFloor {f} returns the largest relative integer N with N ≤ f.

\xintFloor {-2.13}=-3, \xintFloor {-2}=-2, \xintFloor {2.13}=2

24.19 \xintCeil

New with release 1.09a.

\xintCeil {f} returns the smallest relative integer N with N > f.

\xintCeil {-2.13}=-2, \xintCeil {-2}=-2, \xintCeil {2.13}=3

24.20 \xintE

New with 1.07.

\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

24.21 \xintDigits, \xinttheDigits

New with release 1.07.

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.

24.22 \xintFloat

New with release 1.07.

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 zeros 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.

24.23 \xintAdd

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.

24.24 \xintFloatAdd

New with release 1.07.

\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.

24.25 \xintSub

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.

24.26 \xintFloatSub

New with release 1.07.

\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.

24.27 \xintMul

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 \xintiMul.

24.28 \xintFloatMul

New with release 1.07.

\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.

24.29 \xintSqr

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 is available as \xintiSqr.

24.30 \xintDiv

\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].

24.31 \xintFloatDiv

New with release 1.07.

\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.

24.32 \xintFac

Modified in 1.08b (to allow fractions on input).

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.

24.33 \xintPow

\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 999999999; future releases will presumably lower this limit as even much much smaller values already create gigantic numerators and denominators which can not be computed exactly in a reasonable time. Indeed 2^999999999 has 301029996 digits.

24.34 \xintFloatPow

New with 1.07.

\xintFloatPow [P]{f}{x} uses either the optional argument P or the value of \xintDigits. It computes a floating approximation to f^x.

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
```

24.35 \xintFloatPower

New with 1.07.

\xintFloatPower{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 an 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.

24.36 \xintFloatSqrt

New with 1.08.

\xintFloatSqrt[P]{f} computes a floating point approximation of $\sqrt{f}$, 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

24.37 \xintSum

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 is available \xintiSum.

24.38 \xintPrd

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 is available as \xintiPrd.

24.39 \xintCmp

Rewritten in 1.08a.

The macro is extended to fractions. Its output is still either -1, 0, or 1 with no forward slash nor trailing [n]. The original, which skips the overhead of the fraction format parsing, is available as \xintiCmp.

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}.

Note that since release 1.08a using this macro on inputs with large powers of tens does not take a quasi-infinite time, contrarily to the earlier, somewhat dumb version (the earlier version indirectly led to the creation of giant chains of zeros in certain circumstances, causing a serious efficiency impact).

24.40 \xintIsOne

See \xintIsOne (subsection 22.17).

24.41 \xintGeq

Rewritten in 1.08a.

The macro is extended to fractions. The original, which skips the overhead of the fraction format parsing, is available as \xintiGeq. 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|}

Same improvements in 1.08a as for \xintCmp.

24.42 \xintMax

Rewritten in 1.08a.

The macro is extended to fractions. But now \xintMax {2}{3} returns 3/1[0]. The original is available as \xintiMax.

24.43 \xintMaxof

See \xintMaxof (subsection 22.26).

24.44 \xintMin

Rewritten in 1.08a.

The macro is extended to fractions. The original is available as \xintiMin.

24.45 \xintMinof

See \xintMinof (subsection 22.28).

24.46 \xintAbs

The macro is extended to fractions. The original is available as \xintiAbs. Note that \xintAbs {-2}=2/1[0] whereas \xintiAbs {-2}=2.

24.47 \xintSgn

The macro is extended to fractions. Its output is still either -1, 0, or 1 with no forward slash nor trailing [n]. The original, which skips the overhead of the fraction format parsing, is available as \xintiSgn.

24.48 \xintOpp

The macro is extended to fractions. The original is available as \xintiOpp. Note that \xintOpp {3} now outputs -3/1[0].

24.49 \xintDivision, \xintQuo, \xintRem, \xintFDg, \xintLDg, \xintMON, \xintMMON, \xintOdd

These macros are extended to accept a fraction on input if this fraction in fact reduces to an integer (if not an \xintError:NotAnInteger will be raised). As usual, the 'i' variants all exist, they accept on input only integers in the strict format and have less overhead. There is no difference in the output, the difference is only in the accepted format for the inputs.

24.50 \xintNum

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 zeros to get an explicit integer.

```
\xintNum {1e80}
100000000000000000000000000000000000000000000000000000000000000000000000000000000
```

25 Expandable expressions with the xintexpr package

The **xintexpr** package was first released with version 1.07 of the **xint** bundle. Loading this package automatically loads **xintfrac**, hence also **xint**.

> Release 1.09a has extended the scope of \xintexpr-essions with infix comparison operators (<, >, =), logical operators (&, |), functions (round, sqrt, max, all, etc...) and conditional branching (if and ?, ifsgn and :, the function forms evaluate the skipped branches, the ? and : operators do not).
>
> Refer to the first pages of this manual (section 5 and section 6) for the current situation. Apart from some adjustments in the description of \xintNewExpr which now works with #, and removal of obsolete material, the documentation in this section is close to its earlier state describing 1.08b and is lacking in examples illustrating all the new functionality with 1.09a.

Contents

25.1 The \xintexpr expressions

See section 5 for up-to-date information →

An xintexpression is a construct `\xintexpr`⟨*expandable_expression*⟩`\relax` where the expandable expression is read and expanded from left to right, and whose constituents should be (they are uncovered by iterated left to right expansion of the contents during the scanning):

- integers or decimal numbers, such as `123.345`, or numbers in scientific notation `6.02e23` or `6.02E23` (or anything expanding to these things; a decimal number may start directly with a decimal point),
- fractions `A/B`, or `a.b/c.d` or `a.beN/c.deM`, if they are to be treated as one entity should then be parenthesized, *e.g.* disambiguating `A/B^2` from `(A/B)^2`,
- the standard binary operators, `+`, `-`, `*`, `/`, and `^` (the `**` notation for exponentiation is not recognized and will give an error),
- opening and closing parentheses, with arbitrary level of nesting,
- `+` and `-` as prefix operators,
- `!` as postfix factorial operator (applied to a non-negative integer),
- and sub-expressions `\xintexpr`⟨*stuff*⟩`\relax` (they do not need to be put within parentheses).
- braced material `{...}` which is only allowed to arise when the parser is starting to fetch an operand; the material will be completely expanded and *must* deliver some number `A`, or fraction `A/B`, possibly with decimal mark or ending `[n]`, but without the `e`, `E` of the scientific notation. Conversely fractions in `A/B[n]` format with the ending `[n]` *must* be enclosed in such braces. Braces also appear in the completely other rôle of feeding macros with their parameters, they will then not be seen by the parser at all as they are managed by the macro.

Such an expression, like a `\numexpr` expression, is not directly printable, nor can it be directly used as argument to the other package macros. For this one uses one of the two equivalent forms:

- `\xinttheexpr`⟨*expandable_expression*⟩`\relax`, or
- `\xintthe\xintexpr`⟨*expandable_expression*⟩`\relax`.

As with other package macros the computations are done *exactly*, and with no simplification of the result. The output format can be coded inside the expression through the use of one of the functions `round`, `trunc`, `float`, `reduce`.[34]

```
\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
```

[34] 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.

```
\xinttheexpr 1.99^-2 - 2.01^-2 \relax=800/1599920001[4]
\xinttheexpr round(1.99^-2 - 2.01^-2, 10)\relax=0.0050002500
```

- `\xintexpr`-essions evaluate through expansion to arbitrarily big fractions, and are prefixed by `\xintthe` for printing (or use `\xinttheexpr`).
- the standard operations of addition, subtraction, multiplication, division, power, are written in infix form,
- recognized numbers on input are either integers, decimal numbers, or numbers written in scientific notation, (or anything expanding to the previous things),
- macros encountered on the way must be fully expandable,
- fractions on input with the ending `[n]` part, or macros expanding to such some `A/B[n]` must be enclosed in (exactly one) pair of braces,
- 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.
- 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` as it would have to be put within some braces, and it is simpler to write it directly as `\xintexpr...\relax`,
- as usual no simplification is done on the output and is the responsability of post-processing,
- very long output will need special macros to break across lines, like the `\printnumber` macro used in this documentation,
- use of +, *, ... inside parameters to macros is out of the scope of the `\xintexpr` parser,
- finally each **xintexpr**ession is completely expandable and obtains its result in two expansion steps.

With defined macros destined to be re-used within another one, one has the choice between parentheses or `\xintexpr...\relax`: `\def\x {(\a+\b)}` or `\def\x {\xintexpr \a+\b\relax}`. The latter is better as it allows `\xintthe`.

25.2 \numexpr expressions, count and dimension registers

They can not be used directly but must be prefixed by \the or \number for the count registers and by \number for the dimension registers. The dimension is then converted to its value in scalable points sp, which are 1/65536th of a point.

One may thus compute exactly and expandably with dimensions even exceeding temporarily the TeX limits and then convert back approximately to points by division by 65536 and rounding to 4,5 or 6 decimal digits after the decimal point.

25.3 Catcodes and spaces

25.3.1 \xintexprSafeCatcodes

New with release 1.09a.

Active characters will interfere with \xintexpr-essions. One may prefix them with \string or use the command \xintexprSafeCatcodes before the \xintexpr-essions. This (locally) sets the catcodes of the characters acting as operators to safe values. The command \xintNewExpr does it by itself, in a group.

25.3.2 \xintexprRestoreCatcodes

New with release 1.09a.

Restores the catcodes to the earlier state.

Spaces inside an \xinttheexpr...\relax should mostly be innocuous (if the expression contains macros, then it is the macro which is responsible for grabbing its arguments, so spaces within the arguments are presumably to be avoided, as a general rule.).

\xintexpr and \xintthexpr are very agnostic regarding catcodes: digits, binary operators, minus and plus signs as prefixes, parentheses, decimal point, may be indifferently of catcode letter or other or subscript or superscript, ..., it does not matter. The characters +, -, *, /, ^ or ! should not be active as everything is expanded along the way. If one of them (especially ! which is made active by Babel for certain languages) is active, it should be prefixed with \string. In the case of the factorial, the macro \xintFac may be used rather than the postfix !, preferably within braces as this will avoid the subsequent slow scan digit by digit of its expansion (other macros from the xintfrac package generally *must* be used within a brace pair, as they expand to fractions A/B[n] with the trailing [n]; the \xintFac produces an integer with no [n] and braces are only optional, but preferable, as the scanner will get the job done faster.)

Sub-material within braces is treated technically in a different manner, and depending on the macros used therein may be more sensitive to the catcode of the five operations. Digits, slash, square brackets, sign, produced on output by an \xinttheexpr are all of catcode 12. For the output of \xintthefloatexpr digits, decimal dot, signs are of catcode 12, and the ‘e’ is of catcode 11.

Note that if some macro is inserted in the expression it 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 as flexible within the macro arguments.

25.4 Expandability

As is the case with all other package macros \xintexpr expands in two steps to its final (non-printable) result; and similarly for \xinttheexpr.

As explained above the expressions should contain only expandable material, except that braces are allowed when they enclose either a fraction (or decimal number) or something arbitrarily complicated but expanding (in a manner compatible to an expansion only context) to such a fraction or decimal number.

25.5 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. But, if the package is used for computing plots[35], this may cause a problem.

There is a solution.[36]

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 was necessary to do before the availability of the xintexpr package.

25.6 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* despite the bracket notation, and n=0 if the macro to be defined has no parameter,[37]
- the placeholders #1, #2, ..., #n are used inside ⟨*stuff*⟩ in their usual rôle.

---

[35] 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.

[36] which convinced me that I could stick with the parser implementation despite its potential impact on the hash-table.

[37] there is some use for \xintNewExpr[0] compared to an \edef as \xintNewExpr has some built-in catcode protection.

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, it transcends TeX groups or LaTeX environments. The protection against active characters is done automatically.

1.09a: and→ many others...

It will be a completely expandable macro entirely built-up using `\xintAdd`, `\xintSub`, `\xintMul`, `\xintDiv`, `\xintPow`, `\xintOpp` and `\xintFac` and corresponding to the formula as written with the infix operators.

> 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.

25.6.1 Use of conditional operators

The `1.09a` conditional operators `?` and `:` can not be parsed by `\xintNewExpr` when they contain macro parameters within their scope, and not only numerical data. However using the functions `if` and, respectively `ifsgn`, the parsing should succeed. Moreover 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.

25.6.2 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}}
```

25.7 `\xintnumexpr`, `\xintthenumexpr`

Equivalent to doing `\xintexpr round(...)\relax`. Thus, only the final result is rounded to an integer. The rounding is towards $+\infty$ for positive numbers and towards $-\infty$ for negative ones. Can be used on comma separated lists of expressions.

25.8 \xintboolexpr, \xinttheboolexpr

New in 1.09c.

Equivalent to doing \xintexpr ...\relax and returning 1 if the result does not vanish, and 0 is the result is zero (as is the case with \xintexpr, this can be used on comma separated lists of expressions, and will then return a comma separated list of 0's and 1's)).

25.9 \xintifboolexpr

New in 1.09c.

\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. The <expr> can be a pure logic expression using various & and |, with parentheses, the logic functions all, any, xor, the bool or togl operators, but it is not limited to them: the most general computation can be done, as we have here just a wrapper which tests if the outcome of the computation vanishes or not.

This will crash if used on an expression which is a comma separated list: the expression must return a single number/fraction.

25.10 \xintifboolfloatexpr

New in 1.09c.

\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. This will crash if used on an expression which is a comma separated list.

25.11 \xintfloatexpr, \xintthefloatexpr

\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`!

Note that using `\xintthefloatexpr` only pays off compared to using `\xinttheexpr` and then `\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
```

Note that 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)`.

25.12 \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`.

25.13 \xintNewNumExpr

New in `1.09c`.

Like `\xintNewExpr` but using `\xintthenumexpr`.

25.14 \xintNewBoolExpr

New in `1.09c`.

Like `\xintNewExpr` but using `\xinttheboolexpr`.

25.15 Technicalities and experimental status

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 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 token doing the actual printing and finally a token `\.A/B[n]`. Using `\xinttheexpr` means zapping the first two things, the third one will then recover `A/B[n]` from the undefined control sequence `\.A/B[n]`.

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.

> This implementation and user interface are still to be considered *experimental*.

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 absolutely mandatory (contrarily to a `\numexpr`).

25.16 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. 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.

26 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 *ff*-expanded. It then may start with an optional minus sign (unique, of category code other), followed with optional leading zeros (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 zeros are allowed, and stripped. The hexadecimal letters on output are of category code letter, and uppercased.

Contents

26.1 \xintDecToHex

Converts from decimal to hexadecimal.

\xintDecToHex{2718281828459045235360287471352662497757247093699959574
966967627724076630353547594571382178525166427427466391932003}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F4
6DCE46C6032936BF37DAC918814C63

26.2 \xintDecToBin

Converts from decimal to binary.

\xintDecToBin{2718281828459045235360287471352662497757247093699959574
966967627724076630353547594571382178525166427427466391932003}
->1000110101001001110010111110001100110100101001001101010010111000000
1010001111101111101000010101000000101111001000101001110001111000001
0110001011111000100000110110001000111000100100010111010111011100101
0110101011101100000101110110011100011010010011100101111010001101101 1
1001110010001101100011000000011001010010011011010111111001101111011
010110010010001100010000000101001100011000 11

26.3 \xintHexToDec

Converts from hexadecimal to decimal.

\xintHexToDec{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B576
0BB38D272F46DCE46C6032936BF37DAC918814C63}
->2718281828459045235360287471352662497757247093699959574966967627 72
4076630353547594571382178525166427427466391932003

26.4 \xintBinToDec

Converts from binary to decimal.

\xintBinToDec{1000110101001001110010111110001100110100101001001101010
0101110000001010001111101111101000010101000000101111001000101001110001 11
1100000101100010111110001000001101100010001110001001000101110101110111 1
0010101101010111011000001011101100111000110100100111001011110100011011 0
1110011100100011011000110000000110010100100110110101111110011011111011 0
101100100100011000100000001010011000110 0011}
->2718281828459045235360287471352662497757247093699959574966967627 72
4076630353547594571382178525166427427466391932003

26.5 \xintBinToHex

Converts from binary to hexadecimal.

```
\xintBinToHex{1000110101001001110010111110001100110100101001001101010
01011100000101000111110111110100001010100000010111100100010100111000111
11000001011000101111100010000011011000100011100010010001011101011101111
00101011010101110110000010111011001110001101001001110010111101000110110
11100111001000110110001100000001100101001001101101011111100110111110110
101100100100011000100000010100110001100011}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F4
6DCE46C6032936BF37DAC918814C63
```

26.6 \xintHexToBin

Converts from hexadecimal to binary.

```
\xintHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B576
0BB38D272F46DCE46C6032936BF37DAC918814C63}
->1000110101001001110010111110001100110100101001001101010010111000001
1010001111101111101000010101000000101111001000101001110001111100000 1
01100010111110001000001101100010001110001001000101110101110111100101
01101010111011000001011101100111000110100100111001011110100011011011
10011100100011011000110000000110010100100110110101111110011011111011
0101100100100011000100000010100110001100011
```

26.7 \xintCHexToBin

Also converts from hexadecimal to binary. Faster on inputs with at least one hundred hexadecimal digits.

```
\xintCHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B57
60BB38D272F46DCE46C6032936BF37DAC918814C63}
->1000110101001001110010111110001100110100101001001101010010111000001
10100011111011111010000101010000001011110010001010011100011111000001
01100010111110001000001101100010001110001001000101110101110111100101
01101010111011000001011101100111000110100100111001011110100011011011
10011100100011011000110000000110010100100110110101111110011011111011
0101100100100011000100000010100110001100011
```

27 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

27.1 \xintGCD

\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

27.2 \xintGCDof

New with release 1.09a.

\xintGCDof{{a}{b}{c}...} computes the greatest common divisor of all integers a, b, ... The list argument may be a macro, it is *ff*-expanded first and must contain at least one item.

27.3 \xintLCM

New with release 1.09a.

\xintGCD{N}{M} computes the least common multiple. It is 0 if one of the two integers vanishes.

27.4 \xintLCMof

New with release 1.09a.

\xintLCMof{{a}{b}{c}...} computes the least common multiple of all integers a, b, ... The list argument may be a macro, it is *ff*-expanded first and must contain at least one item.

27.5 \xintBezout

\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:->{10000}{1113}{-131}{-1177}{1}.

\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.

27.6 \xintEuclideAlgorithm

\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:->{5}{10000}{1}{1113}{8}{1096}{1}{17}{64}{8}{2}{1}{8}{0}.

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.

27.7 \xintBezoutAlgorithm

\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:->{5}{10000}{0}{1}{1}{1113}{1}{0}{8}{1096}{8}{1}{1}{17}{9}{1}{64}{8}{584}{65}{2}{1}{1177}{131}{8}{0}{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.

27.8 \xintTypesetEuclideAlgorithm

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}
```

$$
\begin{aligned}
123456789012345 &= 12 \times 9876543210321 + 4938270488493\\
9876543210321 &= 2 \times 4938270488493 + 2233335\\
4938270488493 &= 2211164 \times 2233335 + 536553\\
2233335 &= 4 \times 536553 + 87123\\
536553 &= 6 \times 87123 + 13815\\
87123 &= 6 \times 13815 + 4233\\
13815 &= 3 \times 4233 + 1116\\
4233 &= 3 \times 1116 + 885\\
1116 &= 1 \times 885 + 231\\
885 &= 3 \times 231 + 192\\
231 &= 1 \times 192 + 39\\
192 &= 4 \times 39 + 36\\
39 &= 1 \times 36 + 3\\
36 &= 12 \times 3 + 0
\end{aligned}
$$

27.9 \xintTypesetBezoutAlgorithm

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}
```

$$
\begin{aligned}
10000 &= 8 \times 1113 + 1096\\
8 &= 8 \times 1 + 0\\
1 &= 8 \times 0 + 1
\end{aligned}
$$

$$\begin{aligned}1113 &= 1 \times 1096 + 17\\ 9 &= 1 \times 8 + 1\\ 1 &= 1 \times 1 + 0\\ 1096 &= 64 \times 17 + 8\\ 584 &= 64 \times 9 + 8\\ 65 &= 64 \times 1 + 1\\ 17 &= 2 \times 8 + 1\\ 1177 &= 2 \times 584 + 9\\ 131 &= 2 \times 65 + 1\\ 8 &= 8 \times 1 + 0\\ 10000 &= 8 \times 1177 + 584\\ 1113 &= 8 \times 131 + 65\end{aligned}$$

$$131 \times 10000 - 1177 \times 1113 = -1$$

28 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 *ff*-expanded, they may be count registers, etc...

This package was first released with version `1.03` of the **xint** bundle.

Contents

28.1 \xintSeries

`\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 fully expandable command, taking on input an explicit number `n` and producing some fraction `\coeff{n}`; it is expanded at the time it is needed.

```
\def\coeff #1{\xintiMON{#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...	11. 0.736544011544...	21. 0.716390450794...
2. 0.500000000000...	12. 0.653210678210...	22. 0.670935905339...
3. 0.833333333333...	13. 0.730133755133...	23. 0.714414166209...
4. 0.583333333333...	14. 0.658705183705...	24. 0.672747499542...
5. 0.783333333333...	15. 0.725371850371...	25. 0.712747499542...
6. 0.616666666666...	16. 0.662871850371...	26. 0.674285961081...
7. 0.759523809523...	17. 0.721695379783...	27. 0.711322998118...
8. 0.634523809523...	18. 0.666139824228...	28. 0.675608712404...
9. 0.745634920634...	19. 0.718771403175...	29. 0.710091471024...
10. 0.645634920634...	20. 0.668771403175...	30. 0.676758137691...

28.2 \xintiSeries

`\xintiSeries{A}{B}{\coeff}` computes $\sum_{n=A}^{n=B}$ `\coeff{n}` where now `\coeff{n}` *must* expand to a (possibly long) integer, as is acceptable on input by the integer-only `\xintiAdd`.

```
\def\coeff #1{\xintiTrunc {40}{\xintMON{#1}/#1.5}}%
```

```
% better:
\def\coeff #1{\xintiTrunc {40}
   {\the\numexpr 2*\xintiMON{#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 `\xintiMON` 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]}%
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12}
   = \xintTrunc {50}{\xintSeries {0}{50}{\exactcoeff}}\dots\]
```

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} \approx 1.5805993064935250412367895069567264144804$$

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} = 1.58059930649352504123678950695672641448068680288367\dots$$

This shows indeed that our sum of truncated terms estimated wrongly the 39th and 40th digits of the exact result[38] and that the sum of rounded terms fared a bit better.

[38] 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.

28.3 \xintRationalSeries

New with release 1.04.

\xintRationalSeries{A}{B}{f}{\ratio} evaluates $\sum_{n=A}^{n=B} F(n)$, 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
```

$\sum_{n=0}^{0} \frac{2^n}{n!} = 1.000000000000\cdots = 1 = 1$

$\sum_{n=0}^{1} \frac{2^n}{n!} = 3.000000000000\cdots = 3 = 3$

$\sum_{n=0}^{2} \frac{2^n}{n!} = 5.000000000000\cdots = \frac{10}{2} = 5$

$\sum_{n=0}^{3} \frac{2^n}{n!} = 6.333333333333\cdots = \frac{38}{6} = \frac{19}{3}$

$\sum_{n=0}^{4} \frac{2^n}{n!} = 7.000000000000\cdots = \frac{168}{24} = 7$

$\sum_{n=0}^{5} \frac{2^n}{n!} = 7.266666666666\cdots = \frac{872}{120} = \frac{109}{15}$

$\sum_{n=0}^{6} \frac{2^n}{n!} = 7.355555555555\cdots = \frac{5296}{720} = \frac{331}{45}$

$\sum_{n=0}^{7} \frac{2^n}{n!} = 7.380952380952\cdots = \frac{37200}{5040} = \frac{155}{21}$

$\sum_{n=0}^{8} \frac{2^n}{n!} = 7.387301587301\cdots = \frac{297856}{40320} = \frac{2327}{315}$

$\sum_{n=0}^{9} \frac{2^n}{n!} = 7.388712522045\cdots = \frac{2681216}{362880} = \frac{20947}{2835}$

$\sum_{n=0}^{10} \frac{2^n}{n!} = 7.388994708994\cdots = \frac{26813184}{3628800} = \frac{34913}{4725}$

$\sum_{n=0}^{11} \frac{2^n}{n!} = 7.389046015712\cdots = \frac{294947072}{39916800} = \frac{164591}{22275}$

$\sum_{n=0}^{12} \frac{2^n}{n!} = 7.389054566832\cdots = \frac{3539368960}{479001600} = \frac{691283}{93555}$

$\sum_{n=0}^{13} \frac{2^n}{n!} = 7.389055882389\cdots = \frac{46011804672}{6227020800} = \frac{14977801}{2027025}$

$\sum_{n=0}^{14} \frac{2^n}{n!} = 7.389056070325\cdots = \frac{644165281792}{87178291200} = \frac{314533829}{42567525}$

$\sum_{n=0}^{15} \frac{2^n}{n!} = 7.389056095384\cdots = \frac{9662479259648}{1307674368000} = \frac{4718007451}{638512875}$

$\sum_{n=0}^{16} \frac{2^n}{n!} = 7.389056098516\cdots = \frac{154599668219904}{20922789888000} = \frac{1572669151}{212837625}$

$\sum_{n=0}^{17} \frac{2^n}{n!} = 7.389056098884\cdots = \frac{2628194359869440}{355687428096000} = \frac{16041225341}{2170943775}$

$\sum_{n=0}^{18} \frac{2^n}{n!} = 7.389056098925\cdots = \frac{47307498477912064}{6402373705728000} = \frac{103122162907}{13956067125}$

$\sum_{n=0}^{19} \frac{2^n}{n!} = 7.389056098930\cdots = \frac{898842471080853504}{121645100408832000} = \frac{4571749222213}{618718975875}$

$\sum_{n=0}^{20} \frac{2^n}{n!} = 7.389056098930\cdots = \frac{17976849421618118656}{2432902008176640000} = \frac{68576238333199}{9280784638125}$

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
```

$\sum_{n=1}^{1} \frac{1^n}{n!} / \sum_{n=0}^{1} \frac{1^n}{n!} = 0.50000000\dots$
$\sum_{n=2}^{3} \frac{2^n}{n!} / \sum_{n=0}^{3} \frac{2^n}{n!} = 0.52631578\dots$
$\sum_{n=3}^{5} \frac{3^n}{n!} / \sum_{n=0}^{5} \frac{3^n}{n!} = 0.53804347\dots$
$\sum_{n=4}^{7} \frac{4^n}{n!} / \sum_{n=0}^{7} \frac{4^n}{n!} = 0.54317053\dots$
$\sum_{n=5}^{9} \frac{5^n}{n!} / \sum_{n=0}^{9} \frac{5^n}{n!} = 0.54502576\dots$
$\sum_{n=6}^{11} \frac{6^n}{n!} / \sum_{n=0}^{11} \frac{6^n}{n!} = 0.54518217\dots$
$\sum_{n=7}^{13} \frac{7^n}{n!} / \sum_{n=0}^{13} \frac{7^n}{n!} = 0.54445274\dots$
$\sum_{n=8}^{15} \frac{8^n}{n!} / \sum_{n=0}^{15} \frac{8^n}{n!} = 0.54327992\dots$
$\sum_{n=9}^{17} \frac{9^n}{n!} / \sum_{n=0}^{17} \frac{9^n}{n!} = 0.54191055\dots$
$\sum_{n=10}^{19} \frac{10^n}{n!} / \sum_{n=0}^{19} \frac{10^n}{n!} = 0.54048295\dots$
$\sum_{n=11}^{21} \frac{11^n}{n!} / \sum_{n=0}^{21} \frac{11^n}{n!} = 0.53907332\dots$
$\sum_{n=12}^{23} \frac{12^n}{n!} / \sum_{n=0}^{23} \frac{12^n}{n!} = 0.53772178\dots$
$\sum_{n=13}^{25} \frac{13^n}{n!} / \sum_{n=0}^{25} \frac{13^n}{n!} = 0.53644744\dots$
$\sum_{n=14}^{27} \frac{14^n}{n!} / \sum_{n=0}^{27} \frac{14^n}{n!} = 0.53525726\dots$
$\sum_{n=15}^{29} \frac{15^n}{n!} / \sum_{n=0}^{29} \frac{15^n}{n!} = 0.53415135\dots$
$\sum_{n=16}^{31} \frac{16^n}{n!} / \sum_{n=0}^{31} \frac{16^n}{n!} = 0.53312615\dots$
$\sum_{n=17}^{33} \frac{17^n}{n!} / \sum_{n=0}^{33} \frac{17^n}{n!} = 0.53217628\dots$
$\sum_{n=18}^{35} \frac{18^n}{n!} / \sum_{n=0}^{35} \frac{18^n}{n!} = 0.53129566\dots$
$\sum_{n=19}^{37} \frac{19^n}{n!} / \sum_{n=0}^{37} \frac{19^n}{n!} = 0.53047810\dots$
$\sum_{n=20}^{39} \frac{20^n}{n!} / \sum_{n=0}^{39} \frac{20^n}{n!} = 0.52971771\dots$

28.4 `\xintRationalSeriesX`

New with release `1.04`.

`\xintRationalSeriesX{A}{B}{\first}{\ratio}{\g}` is a parametrized version of `\xintRationalSeries` where `\first` is turned into a one parameter macro with `\first {\g}` giving `F(A,\g)` and `\ratio` is a two parameters macro such that `\ratio{n}{\g}` gives `F(n,\g)/F(n-1,\g)`. 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-parameters 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]))=4355349527343049937531284783056957554465259984189164206 5630853442715414147101380720658820298104601315534223370128916508905 6830056936564478988779520000000000/395940866122425193243875570782668457763038822400000000000000000000[-90]` (length of numerator: 155)

`E(L(12[-2]))=443453770054417465442109252347264824711893599160411729 6038825841980841532261080707075058900962803059710371332802034641237 1558877141883806589829590141346329464027599993974220093034636265326 43`

54170486398431674455531227136795459841404436480000000000/395940866122 4251932438755707826684577630388224000000000000000000000[-180] (length of numerator: 245)

E(L(123[-3]))=444641592651941777154254148848854866198954971552616390074295913531792113850864779762350800814416981762774148663052493217566759754097977420731516373336789722730765496139079185229545102248282391199621029237793811740122110919735433161132757168955864017710881850585395079859843831617966207195391567803471832147436302936555630100480000000000/395940866122425193243875570782668457763038822400000000000000000000[-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))=5181385161173226049160748331648333448838405901330061681251253466743091335325539480471366915857159004497689259144894523418643519242240000000000/45337120162108979178809662782137765289223265381758152546654836095087089601022689942796465342115407786358809263904208715776000000000000000000[0] (length of numerator: 141; length of denominator: 141)

E(L(1/71))=164799489177219556498025955806107098256158101756209369864657152282149780083067798039175325186850716609293467854603842163754716919123274624394132188208895310089982001627351524910000588238596565380887916286153347403881434316800000000000/162510607383091507102283159265830434485606350979982865517923046004017115844425486049111273926394712850261667426510159483544917475146636033045963798199826115486814955381536472641379276308916890414267771321449447424000000000000000000000[0] (length of numerator: 232; length of denominator: 232)

E(L(1/712))=209623173880163120675481637897216200283968902248203238943136902264182865559717266406341976325767001357109452980607391271438079195073959301528254006087908156888129567520269011715459969154688799089625738271433856535377918700884980798641197021855117078629780316835353043067415753497212012899985019017494798220551782400000000/2093291722337673799732719862311619975662927884547744846526034295741465968177583093786412050480958301357075221213896546903011983961080605724903426024563430558292203346913309844190901402018394162270065876670575550330002721292096217682473000829618103432600036119035084894266166648343032219206471638591733760000000000000000000[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.[39] 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

[39] 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`.

28.5 `\xintPowerSeries`

`\xintPowerSeries{A}{B}{\coeff}{f}` evaluates the sum $\sum_{n=A}^{n=B}$ `\coeff{n}` $\cdot$ f^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. [40]

> 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]}
\[ \sum_{n=0}^{n=20} \Bigl(\frac 5{17}\Bigr)^n
 =\xintFrac{\xintIrr{\xintPowerSeries {0}{20}{\geom}{\f}}}
 =\xintFrac{\xinttheexpr (17^21-5^21)/12/17^20\relax}\]
```

$$\sum_{n=0}^{n=20}\Bigl(\frac{5}{17}\Bigr)^n=\frac{5757661159377657976885341}{4064231406647572522401601}=\frac{69091933912531895722624092}{48770776879770870268819212}$$

[40] 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
\[\mathrm{Arctg}(\frac15)\approx
           \frac15\sum_{n=0}^{15} \frac{(-1)^n}{(2n+1)25^n}
= \xintFrac{\xintIrr {\xintDiv
           {\xintPowerSeries {0}{15}{\coeffarctg}{\f}}{5}}}\]
```

$$\mathrm{Arctg}(\frac{1}{5}) \approx \frac{1}{5}\sum_{n=0}^{15} \frac{(-1)^n}{(2n+1)25^n} = \frac{165918726519122955895391793269168}{840539304153062403202056884765625}$$

28.6 \xintPowerSeriesX

New with release 1.04.

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…

28.7 \xintFxPtPowerSeries

\xintFxPtPowerSeries{A}{B}{\coeff}{f}{D} computes $\sum_{n=A}^{n=B}$ \coeff{n} · 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:

$$\texttt{\textbackslash xintPowerSeries \{0\}\{19\}\{\textbackslash coeffexp\}\{\textbackslash f\}} = \frac{38682746160036397317757}{63777066403145711616000}$$
$$= 0.606530659712633423603799152126\ldots$$

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.

28.8 \xintFxPtPowerSeriesX

New with release `1.04`.

`\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.

28.9 \xintFloatPowerSeries

New with 1.08a.

\xintFloatPowerSeries[P]{A}{B}{\coeff}{f} computes $\sum_{n=A}^{n=B}$ \coeff{n} · 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

28.10 \xintFloatPowerSeriesX

New with 1.08a.

\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

28.11 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 zeros or nines in the exact decimal expansion (in the present case of $\log 2$, strings of zeros 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...
$\approx$ 0.69314718055994530941723212145817656807550013436025525412068000711...
$\approx$ 0.6931471805599453094172321214581765680755001343602552541206800094933723...

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 zeros or nines encountered in the decimal expansion may falsify the ending digits, nines may be zeros (and the last non-nine one should be increased) and zeros 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).
% Alternatives:
% \def\coeffarctg #1{1/\the\numexpr\xintiMON{#1}*(2*#1+1)\relax }%
% The [0] can *not* be used above, as the denominator is signed.
% \def\coeffarctg #1{\xintiMON{#1}/\the\numexpr 2*#1+1\relax [0]}%
\def\xa {1/25[0]}%      1/5^2, the [0] for faster parsing
\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} % must be lowercase to stop \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? Copy the \Machin code to a Plain TeX or LaTeX document loading **xintseries**, and compile:

```
\newwrite\outfile
\immediate\openout\outfile \jobname-out\relax
\immediate\write\outfile {\Machin {1000}}
\immediate\closeout\outfile
```

This will create a file with the correct first 1000 digits of π after the decimal point. On my laptop (a 2012 model) this took about 42 seconds last time I tried (and for 200 digits it is less than 1 second). 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.

29 Commands of the xintcfrac package

This package was first included in release 1.04 of the **xint** bundle.

Contents

29.1 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. $2.718281828458563411277850606202\cdots = \frac{1084483}{398959}$
18. $2.718281828459065114074529546648\cdots = \frac{13580623}{4996032}$
19. $2.718281828459028013207065591026\cdots = \frac{14665106}{5394991}$
20. $2.718281828459045851404621084949\cdots = \frac{28245729}{10391023}$
21. $2.718281828459045213521983758221\cdots = \frac{410105312}{150869313}$
22. $2.718281828459045254624795027092\cdots = \frac{438351041}{161260336}$
23. $2.718281828459045234757560631479\cdots = \frac{848456353}{312129649}$
24. $2.718281828459045235379013372772\cdots = \frac{14013652689}{5155334720}$
25. $2.718281828459045235343535532787\cdots = \frac{14862109042}{5467464369}$

26. $2.71828182845904523536075323018\cdots = \frac{28875761731}{10622799089}$

27. $2.71828182845904523536027459394\cdots = \frac{534625820200}{196677847971}$

28. $2.71828182845904523536029912091\cdots = \frac{563501581931}{207300647060}$

29. $2.71828182845904523536028717990\cdots = \frac{1098127402131}{403978495031}$

30. $2.71828182845904523536028747861\cdots = \frac{22526049624551}{8286870547680}$

31. $2.71828182845904523536028746472\cdots = \frac{23624177026682}{8690849042711}$

32. $2.71828182845904523536028747150\cdots = \frac{46150226651233}{16977719590391}$

33. $2.71828182845904523536028747134\cdots = \frac{1038929163353808}{382200680031313}$

34. $2.71828182845904523536028747135\cdots = \frac{1085079390005041}{399178399621704}$

35. $2.71828182845904523536028747135\cdots = \frac{2124008553358849}{781379079653017}$

36. $2.71828182845904523536028747135\cdots = \frac{5206128467061741}{1915227631129411}$

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).

```
\edef\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 = 568964038871896267597523892315807875293889017667917446057232024547192296961118230175243860174995310817731367012417086097496343293829 06

Denominator = 331123817669737619306256360816356753365468823729314438156205615463246659728581865461337692063148916019550614570592553376611426452172 23

Expansion = 1.718281828459045235360287471352662497757247093699959574966967627724076630353547594571382178525166427427466391932003059921817413596629043572900334295260595630738132328627943490763233829880753195251019011573834187930702154089149934884167509244761460668082264800168477411 8...

29.2 \xintCFrac

`\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 (or macro expanding in two steps to one such). 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.

29.3 \xintGCFrac

`\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.

29.4 \xintGCtoGCx

New with release `1.05`.

`\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:

```
$$\xintGCtoGCx {+\cfrac}{\\}{a+b/...}\endcfrac$$
$$\xintGCtoGCx {+\cfrac\xintFwOver}{\\\xintFwOver}{a+b/...}\endcfrac$$
```

29.5 \xintFtoCs

`\xintFtoCs{f}` returns the comma separated list of the coefficients of the simple continued fraction of `f`.

```
\[ \xintSignedFrac{-5262046/89233} = [\xintFtoCs{-5262046/89233}]\]
```

$$-\frac{5262046}{89233} = [-59, 33, 27, 100]$$

29.6 \xintFtoCx

`\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`.

29.7 \xintFtoGC

`\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

29.8 \xintFtoCC

`\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

```
\[\xintFrac{566827/208524} = \xintGCFrac{\xintFtoCC{566827/208524}}\]
```

$$\frac{566827}{208524} = 3 - \cfrac{1}{4 - \cfrac{1}{2 + \cfrac{1}{5 - \cfrac{1}{2 + \cfrac{1}{7 - \cfrac{1}{2 + \cfrac{1}{9 - \cfrac{1}{2 + \cfrac{1}{11}}}}}}}}}$$

29.9 \xintFtoCv

`\xintFtoCv{f}` returns the list of the (braced) convergents of `f`, with no separator. To be treated with `\xintAssignArray` or `\xintListWithSep`.

```
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintFtoCv{5211/3748}}}\]
```

$$1 \to \frac{3}{2} \to \frac{4}{3} \to \frac{7}{5} \to \frac{25}{18} \to \frac{32}{23} \to \frac{57}{41} \to \frac{317}{228} \to \frac{374}{269} \to \frac{691}{497} \to \frac{5211}{3748}$$

29.10 \xintFtoCCv

`\xintFtoCCv{f}` returns the list of the (braced) centered convergents of `f`, with no separator. To be treated with `\xintAssignArray` or `\xintListWithSep`.

```
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintFtoCCv{5211/3748}}}\]
```

$$1 \to \frac{4}{3} \to \frac{7}{5} \to \frac{32}{23} \to \frac{57}{41} \to \frac{374}{269} \to \frac{691}{497} \to \frac{5211}{3748}$$

29.11 \xintCstoF

`\xintCstoF{a,b,c,d,...,z}` computes the fraction corresponding to the coefficients, which may be fractions or even macros expanding to such fractions (in two steps). 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).

29.12 \xintCstoCv

`\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
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintCstoCv
{\xintPow {-.3}{-5},7.3/4.57,\xintCstoF{3/4,9,-1/3}}}}\]
```

$$\frac{-100000}{243}\to\frac{-72888949}{177390}\to\frac{-2700356878}{6567804}$$

29.13 \xintCstoGC

`\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.

```
\[\xintGCFrac {\xintCstoGC {-1,1/2,-1/3,1/4,-1/5}}
=\xintSignedFrac {\xintCstoF {-1,1/2,-1/3,1/4,-1/5}}\]
```

$$-1+\cfrac{1}{\frac{1}{2}+\cfrac{1}{\frac{-1}{3}+\cfrac{1}{\frac{1}{4}+\cfrac{1}{\frac{-1}{5}}}}}=-\frac{145}{83}$$

29.14 \xintGCtoF

`\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.

29.15 \xintGCtoCv

`\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.

```
\[\xintListWithSep{,}{\xintApply\xintFrac
                 {\xintGCtoCv{3+{-2}/{7/2}+{3/4}/12+{-56}/3}}}\]
\[\xintListWithSep{,}{\xintApply\xintFrac{\xintApply\xintIrr
                 {\xintGCtoCv{3+{-2}/{7/2}+{3/4}/12+{-56}/3}}}}\]
```

$$3, \frac{17}{7}, \frac{834}{342}, \frac{1306}{542}$$

$$3, \frac{17}{7}, \frac{139}{57}, \frac{653}{271}$$

29.16 \xintCntoF

\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]
```

29.17 \xintGCntoF

\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
\[\xintGCFrac{\xintGCntoGC {6}{\coeffA}{\coeffB}}
 = \xintFrac{\xintGCntoF {6}{\coeffA}{\coeffB}}\]
```

29.18 \xintCntoCs

\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
\[\xintFrac{\xintCntoF {5}{\macro}}=\xintCFrac{\xintCntoF {5}{\macro}}\]
```

$$\frac{72625}{49902} = 1 + \cfrac{1}{2 + \cfrac{1}{5 + \cfrac{1}{10 + \cfrac{1}{17 + \cfrac{1}{26}}}}}$$

29.19 \xintCntoGC

`\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}
\[\xintGCFrac{\xintCntoGC {5}{\macro}}\]
```

$$1 + \cfrac{1}{\frac{-2}{2} + \cfrac{1}{\frac{3}{5} + \cfrac{1}{\frac{-4}{10} + \cfrac{1}{\frac{5}{17} + \cfrac{1}{\frac{-6}{26}}}}}}$$

29.20 \xintGCntoGC

`\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.

```
\def\an #1{\the\numexpr #1*#1*#1+1\relax}%
\def\bn #1{\the\numexpr \xintiMON{#1}*(#1+1)\relax}%
$\xintGCntoGC {5}{\an}{\bn}=\xintGCFrac {\xintGCntoGC {5}{\an}{\bn}}
        = \displaystyle\xintFrac {\xintGCntoF  {5}{\an}{\bn}}$\par
```

$$1 + 1/2 + -2/9 + 3/28 + -4/65 + 5/126 = 1 + \cfrac{1}{2 - \cfrac{2}{9 + \cfrac{3}{28 - \cfrac{4}{65 + \cfrac{5}{126}}}}} = \frac{5797655}{3712466}$$

29.21 \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.

29.22 \xintGCtoGC

`\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.

```
\edef\x {\xintGCtoGC
  {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {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.

30 Package xint implementation

The commenting of the macros is currently (2013/10/22) very sparse.

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

30.1 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 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 and the module (not very probable anyhow...).

```
\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{xint}{\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{xint}{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
         \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   % `
        \catcode94=11   % ^
      \catcode36=3   % $
      \catcode91=12  % [
      \catcode93=12  % ]
      \catcode33=11  % !
      \catcode64=11  % @
      \catcode38=12  % &
      \catcode124=12 % |
      \catcode63=11  % ?
      }%
      \XINT_setcatcodes
  }%
\ChangeCatcodesIfInputNotAborted
\def\XINTsetupcatcodes {% for use by other modules
      \edef\XINT_restorecatcodes_endinput
      {%
           \XINT_storecatcodes\noexpand\endinput %
      }%
      \XINT_setcatcodes
}%
```

30.2 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`. No `firstoftwo` etc.. yet here.

```
\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 {xint}%
  [2013/10/22 v1.09d Expandable operations on long numbers (jfB)]%
```

30.3 Token management, constants

```
\def\xint_gobble_     {}%
\def\xint_gobble_i    #1{}%
\def\xint_gobble_ii   #1#2{}%
\def\xint_gobble_iii  #1#2#3{}%
\def\xint_gobble_iv   #1#2#3#4{}%
\def\xint_gobble_v    #1#2#3#4#5{}%
\def\xint_gobble_vi   #1#2#3#4#5#6{}%
\def\xint_gobble_vii  #1#2#3#4#5#6#7{}%
\def\xint_gobble_viii #1#2#3#4#5#6#7#8{}%
\def\xint_firstofone  #1{#1}%
\xint_firstofone{\let\XINT_sptoken= } % 1.09d, 2013/10/22
\def\xint_firstoftwo  #1#2{#1}%
\def\xint_secondoftwo #1#2{#2}%
\def\xint_firstoftwo_andstop  #1#2{ #1}%
\def\xint_secondoftwo_andstop #1#2{ #2}%
\def\xint_exchangetwo_keepbraces_andstop #1#2{ {#2}{#1}}%
\def\xint_firstofthree  #1#2#3{#1}%
\def\xint_secondofthree #1#2#3{#2}%
\def\xint_thirdofthree  #1#2#3{#3}%
\def\xint_minus_andstop { -}%
\def\xint_gob_til_R     #1\R {}%
\def\xint_gob_til_W     #1\W {}%
\def\xint_gob_til_Z     #1\Z {}%
\def\xint_gob_til_zero  #10{}%
\def\xint_gob_til_one   #11{}%
\def\xint_gob_til_G     #1G{}%
\def\xint_gob_til_minus #1-{}%
\def\xint_gob_til_zeros_iii #1000{}%
\def\xint_gob_til_zeros_iv  #10000{}%
\def\xint_gob_til_relax      #1\relax {}%
\def\xint_gob_til_xint_undef #1\xint_undef {}%
\def\xint_gob_til_xint_relax #1\xint_relax {}%
\def\xint_UDzerofork      #10\dummy  #2#3\krof {#2}%
\def\xint_UDsignfork      #1-\dummy  #2#3\krof {#2}%
\def\xint_UDwfork         #1\W\dummy #2#3\krof {#2}%
\def\xint_UDzerosfork     #100\dummy #2#3\krof {#2}%
```

```
\def\xint_UDonezerofork   #110\dummy #2#3\krof {#2}%
\def\xint_UDzerominusfork #10-\dummy #2#3\krof {#2}%
\def\xint_UDsignsfork     #1--\dummy #2#3\krof {#2}%
\def\xint_afterfi #1#2\fi {\fi #1}%
\let\xint_relax\relax
\def\xint_braced_xint_relax {\xint_relax }%
\chardef\xint_c_     0
\chardef\xint_c_i    1
\chardef\xint_c_ii   2
\chardef\xint_c_iii  3
\chardef\xint_c_iv   4
\chardef\xint_c_v    5
\chardef\xint_c_viii 8
\chardef\xint_c_ix   9
\chardef\xint_c_x   10
\newcount\xint_c_x^viii  \xint_c_x^viii 100000000
\newtoks\XINT_toks
```

30.4 \xintRev, \xintReverseOrder

\xintRev: fait l'expansion avec \romannumeral-'0, vérifie le signe.
\xintReverseOrder: ne fait PAS l'expansion, ne regarde PAS le signe.

```
\def\xintRev {\romannumeral0\xintrev }%
\def\xintrev #1%
{%
    \expandafter\XINT_rev_fork
    \romannumeral-'0#1\xint_relax % empty #1 ok
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
    \xint_relax
}%
\def\XINT_rev_fork #1%
{%
    \xint_UDsignfork
      #1\dummy {\expandafter\xint_minus_andstop
                 \romannumeral0\XINT_rord_main {}}%
       -\dummy {\XINT_rord_main {}#1}%
    \krof
}%
\def\XINT_Rev          {\romannumeral0\XINT_rev }%
\def\xintReverseOrder {\romannumeral0\XINT_rev }%
\def\XINT_rev #1%
{%
    \XINT_rord_main {}#1%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
}%
```

```
\def\XINT_rord_main #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint_undef #9\XINT_rord_cleanup\xint_undef
    \XINT_rord_main {#9#8#7#6#5#4#3#2#1}%
}%
\def\XINT_rord_cleanup\xint_undef\XINT_rord_main #1#2\xint_relax
{%
    \expandafter\space\xint_gob_til_xint_relax #1%
}%
```

30.5 \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.

```
\def\xintRevWithBraces         {\romannumeral0\xintrevwithbraces }%
\def\xintRevWithBracesNoExpand {\romannumeral0\xintrevwithbracesnoexpand }%
\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\Z
}%
\def\xintrevwithbracesnoexpand #1%
{%
    \XINT_revwbr_loop {}%
    #1\xint_relax\xint_relax\xint_relax\xint_relax
      \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\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}%
}%
\def\XINT_revwbr_finish_a\xint_relax\XINT_revwbr_loop #1#2\Z
{%
    \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
}%
```

30.6 \xintLen, \xintLength

\xintLen -> fait l'expansion, ne compte PAS le signe.
\xintLength -> ne fait PAS l'expansion, compte le signe.
1.06: improved code is roughly 20% faster than the one from earlier versions.
1.09a, \xintnum inserted

```
\def\xintiLen {\romannumeral0\xintilen }%
\def\xintilen #1%
{%
    \expandafter\XINT_length_fork
    \romannumeral0\xintnum{#1}\xint_relax\xint_relax\xint_relax\xint_relax
                  \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\let\xintLen\xintiLen \let\xintlen\xintilen
\def\XINT_Len #1%
{%
    \romannumeral0\XINT_length_fork
    #1\xint_relax\xint_relax\xint_relax\xint_relax
      \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\def\XINT_length_fork #1%
{%
    \expandafter\XINT_length_loop
    \xint_UDsignfork
      #1\dummy {{0}}%
       -\dummy {{0}#1}%
    \krof
}%
\def\XINT_Length {\romannumeral0\XINT_length }%
\def\XINT_length #1%
{%
    \XINT_length_loop
    {0}#1\xint_relax\xint_relax\xint_relax\xint_relax
         \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\let\xintLength\XINT_Length
\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\Z
{%
    \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
}%
\def\XINT_length_finish_c #1#2\Z #3%
   {\expandafter\space\the\numexpr #3-#1\relax}%
```

30.7 \xintCSVtoList

\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). Blanks either before or after the separator will be collapsed into one space and the is no attempt to get rid of those. First included in release 1.06.

```
\def\xintCSVtoList {\romannumeral0\xintcsvtolist }%
\def\xintCSVtoListNoExpand {\romannumeral0\xintcsvtolistnoexpand }%
\def\xintcsvtolist #1%
{%
    \expandafter\XINT_csvtol_loop_a\expandafter
    {\expandafter}\romannumeral-`0#1%
        ,\xint_undef,\xint_undef,\xint_undef,\xint_undef
        ,\xint_undef,\xint_undef,\xint_undef,\xint_undef,\Z
}%
\def\xintcsvtolistnoexpand #1%
{%
    \XINT_csvtol_loop_a
    {}#1,\xint_undef,\xint_undef,\xint_undef,\xint_undef
        ,\xint_undef,\xint_undef,\xint_undef,\xint_undef,\Z
}%
\def\XINT_csvtol_loop_a #1#2,#3,#4,#5,#6,#7,#8,#9,%
{%
    \xint_gob_til_xint_undef #9\XINT_csvtol_finish_a\xint_undef
    \XINT_csvtol_loop_b {#1}{{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}}%
```

```
}%
\def\XINT_csvtol_loop_b #1#2{\XINT_csvtol_loop_a {#1#2}}%
\def\XINT_csvtol_finish_a\xint_undef\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
}%
\def\XINT_csvtol_finish_dviii #1#2#3#4#5#6#7#8#9{ #9}%
\def\XINT_csvtol_finish_dvii  #1#2#3#4#5#6#7#8#9{ #9{#1}}%
\def\XINT_csvtol_finish_dvi   #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}}%
\def\XINT_csvtol_finish_dv    #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}}%
\def\XINT_csvtol_finish_div   #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}}%
\def\XINT_csvtol_finish_diii  #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}{#5}}%
\def\XINT_csvtol_finish_dii   #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}{#5}{#6}}%
\def\XINT_csvtol_finish_di    #1#2#3#4#5#6#7#8#9%
                                             { #9{#1}{#2}{#3}{#4}{#5}{#6}{#7}}%
```

30.8 \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).

```
\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\Z }%
\long\def\xintlistwithsepnoexpand #1#2{\XINT_lws_start {#1}#2\Z }%
\long\def\XINT_lws_start #1#2%
```

```
{%
    \xint_gob_til_Z #2\XINT_lws_dont\Z
    \XINT_lws_loop_a {#2}{#1}%
}%
\long\def\XINT_lws_dont\Z\XINT_lws_loop_a #1#2{ }%
\long\def\XINT_lws_loop_a #1#2#3%
{%
    \xint_gob_til_Z #3\XINT_lws_end\Z
    \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\Z\XINT_lws_loop_b #1#2#3{ #1}%
```

30.9 `\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.

```
\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}%
}%
\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
}%
\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
}%
\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\Z
}%
\def\XINT_nthelt_loop_a #1%
{%
    \ifnum #1>\xint_c_viii
        \expandafter\XINT_nthelt_loop_b
    \else
        \expandafter\XINT_nthelt_getit
    \fi
    {#1}%
}%
\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\relax}%
}%
\def\XINT_nthelt_silentend #1\Z { }%
\def\XINT_nthelt_getit #1%
{%
    \expandafter\expandafter\expandafter\XINT_nthelt_finish
    \csname xint_gobble_\romannumeral\numexpr#1-1\endcsname
}%
\def\XINT_nthelt_finish #1#2\Z
{%
    \xint_UDwfork
      #1\dummy { }%
      \W\dummy { #1}%
    \krof
}%
```

30.10 \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.

```
\def\xintApply         {\romannumeral0\xintapply }%
\def\xintApplyNoExpand {\romannumeral0\xintapplynoexpand }%
\def\xintapply #1#2%
{%
    \expandafter\XINT_apply\expandafter {\romannumeral-`0#2}%
    {#1}%
}%
\def\XINT_apply #1#2{\XINT_apply_loop_a {}{#2}#1\Z }%
\def\xintapplynoexpand #1#2{\XINT_apply_loop_a {}{#1}#2\Z }%
\def\XINT_apply_loop_a #1#2#3%
{%
    \xint_gob_til_Z #3\XINT_apply_end\Z
    \expandafter
    \XINT_apply_loop_b
    \expandafter {\romannumeral-`0#2{#3}}{#1}{#2}%
}%
\def\XINT_apply_loop_b #1#2{\XINT_apply_loop_a {#2{#1}}}%
\def\XINT_apply_end\Z\expandafter\XINT_apply_loop_b\expandafter #1#2#3{ #2}%
```

30.11 `\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.

```
\def\xintApplyUnbraced {\romannumeral0\xintapplyunbraced }%
\def\xintApplyUnbracedNoExpand {\romannumeral0\xintapplyunbracednoexpand }%
\def\xintapplyunbraced #1#2%
{%
    \expandafter\XINT_applyunbr\expandafter {\romannumeral-`0#2}%
    {#1}%
}%
\def\XINT_applyunbr #1#2{\XINT_applyunbr_loop_a {}{#2}#1\Z }%
\def\xintapplyunbracednoexpand #1#2%
   {\XINT_applyunbr_loop_a {}{#1}#2\Z }%
\def\XINT_applyunbr_loop_a #1#2#3%
{%
    \xint_gob_til_Z #3\XINT_applyunbr_end\Z
    \expandafter\XINT_applyunbr_loop_b
    \expandafter {\romannumeral-`0#2{#3}}{#1}{#2}%
}%
\def\XINT_applyunbr_loop_b #1#2{\XINT_applyunbr_loop_a {#2#1}}%
\def\XINT_applyunbr_end\Z
    \expandafter\XINT_applyunbr_loop_b\expandafter #1#2#3{ #2}%
```

30.12 \xintSeq

1.09c. Without the optional argument puts stress on the input stack, should not be used to generated thousands of terms then.

```
\def\xintSeq {\romannumeral0\xintseq }%
\def\xintseq #1{\XINT_seq_chkopt  #1\Z }%
\def\XINT_seq_chkopt #1%
{%
    \ifx [#1\expandafter\XINT_seq_opt
       \else\expandafter\XINT_seq_noopt
    \fi  #1%
}%
\def\XINT_seq_noopt #1\Z #2%
{%
    \expandafter\XINT_seq\expandafter
       {\the\numexpr#1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_seq #1#2%
{%
   \ifcase\xintiSgn{\the\numexpr #2-#1\relax}
      \expandafter\xint_firstoftwo_andstop
   \or
      \expandafter\XINT_seq_p
   \else
      \expandafter\XINT_seq_n
   \fi
   {#2}{#1}%
}%
\def\XINT_seq_p #1#2%
{%
    \ifnum #1>#2
      \xint_afterfi{\expandafter\XINT_seq_p}%
    \else
      \expandafter\XINT_seq_e
    \fi
    \expandafter{\the\numexpr #1-1}{#2}{#1}%
}%
\def\XINT_seq_n #1#2%
{%
    \ifnum #1<#2
      \xint_afterfi{\expandafter\XINT_seq_n}%
    \else
      \expandafter\XINT_seq_e
    \fi
     \expandafter{\the\numexpr #1+1}{#2}{#1}%
}%
\def\XINT_seq_e #1#2#3{ }%
\def\XINT_seq_opt [\Z #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\xintiSgn{\the\numexpr #2-#1\relax}
      \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\XINT_Sgn {#3}
        \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\XINT_Sgn {#3}
        \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}}}%
```

30.13 \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, does survive to the case when the expansion of \macro will close a group, as happens with & in alignments. It uses catcode 3 z as list terminator.

1.09d: the same bug with a terminating space token which was discovered in \xintFor* also was in \xintApplyInline. I modify it according to a similar scheme. The new version will thus expand unbraced item elements. This is in fact convenient to insert lists in others.

```
\catcode`z 3%
\def\XINT_xflet #1%
{%
    \expandafter\futurelet\expandafter\XINT_token
    \expandafter#1\romannumeral-`0\romannumeral-`0%
}%
\def\xintApplyInline #1#2%
{%
  \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 {\futurelet\XINT_token\XINT_inline_c }%
\def\XINT_inline_b
```

```
{%
  \ifx\XINT_token\XINT_sptoken
      \xint_afterfi{\XINT_xflet\XINT_inline_b }%
  \else
      \xint_afterfi
          {\ifx\XINT_token z\expandafter\xint_gobble_i
            \else\expandafter\XINT_inline_d\fi }%
  \fi
}%
\def\XINT_inline_d #1%
{%
  \def\XINT_item{{#1}}\XINT_xflet\XINT_inline_e
}%
\def\XINT_inline_e
{%
  \ifx\XINT_token\XINT_sptoken
      \xint_afterfi{\XINT_xflet\XINT_inline_e }%
  \else
      \xint_afterfi
          {\ifx\XINT_token z\expandafter\XINT_inline_w
            \else  \expandafter\XINT_inline_f\fi }%
  \fi
}%
\def\XINT_inline_f
{%
  \expandafter\XINT_inline_g\expandafter{\XINT_inline_macro {##1}}%
}%
\def\XINT_inline_g #1%
{%
   \expandafter\XINT_inline_macro\XINT_item
   \def\XINT_inline_macro ##1{#1}\XINT_inline_d
}%
\def\XINT_inline_w #1%
{%
   \expandafter\XINT_inline_macro\XINT_item
}%
```

30.14 \xintFor, \xintFor*

1.09c: 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.

The #1 will be the macro character #. The \romannumeral#2 in \XINT_for(x) will swallow a space token from blanks before the ‘in’. Blanks after the ‘in’ disappear as #3 is not delimited.

1.09d: \xintFor* crashed when a space token was at the very end of the list. Indeed 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. Now, we conscientiously clean out of the way space tokens. And with the new code, the macro ff-expands each item which is not braced. This way, it is very easy to simulate concatenation of lists or the fact to insert one within the other without having to waste time doing it really. If the list contains two consecutive space tokens and then an unbraced token item \x, this \x will not be expanded. But if we expanded it we would have the risk to again have one or more space token and there could be nothing up to the z. But then the z would be picked next time, spaces discarded, and a crash. And for some reasons I don't want to do an \ifx to compare with {z}. I could use the technique of my completely expandable macros with a gob_til_z. I chose to do it this way, which is guaranteed not to crash at the z, with the feature that unbraced items consecutive to two or more space tokens (a surely rare case, which would require some devilish soul wanting to stress test my package as of course consecutive blanks only give _one_ space token) will not get expanded. [2013/10/22]

```
\def\xintFor {\futurelet\XINT_token\XINT_for_ifstar }%
\def\XINT_for_ifstar {\ifx\XINT_token*\expandafter\XINT_forx
                                 \else\expandafter\XINT_for \fi }%
\def\XINT_for #1#2in#3#4#5%
{%
    \XINT_toks \expandafter{\csname XINT_for_d\romannumeral#2\endcsname {#5}}%
    \expandafter\XINT_for_b #3,z,% THIS z HAS CATCODE 3.
}%
\def\XINT_forx *#1#2in#3#4#5%
{%
    \XINT_toks \expandafter{\csname XINT_forx_d\romannumeral#2\endcsname {#5}}%
    \XINT_xflet\XINT_forx_b #3z% THIS z HAS CATCODE 3.
}%
\def\XINT_for_b {\futurelet\XINT_token\XINT_for_c }%
\def\XINT_for_c
{%
  \ifx\XINT_token z\expandafter\xint_gobble_iv\fi
  \the\XINT_toks
}%
\def\XINT_for_di #1#2,%
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {#2}{####2}{####3}{####4}}%
  \XINT_toks {\XINT_x\XINT_for_di {#1}}%
  \futurelet\XINT_token\XINT_for_e
}%
\def\XINT_for_dii #1#2,%
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {####1}{#2}{####3}{####4}}%
  \XINT_toks {\XINT_x \XINT_for_dii {#1}}%
  \futurelet\XINT_token\XINT_for_e
```

```
}%
\def\XINT_for_diii #1#2,%
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {####1}{####2}{#2}{####4}}%
  \XINT_toks {\XINT_x \XINT_for_diii {#1}}%
  \futurelet\XINT_token\XINT_for_e
}%
\def\XINT_for_div #1#2,%
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {####1}{####2}{####3}{#2}}%
  \XINT_toks {\XINT_x \XINT_for_div {#1}}%
  \futurelet\XINT_token\XINT_for_e
}%
\def\XINT_for_e
{%
    \ifx\XINT_token z\xint_afterfi{\expandafter\XINT_x \xint_gobble_iv}\fi
    \the\XINT_toks
}%
\def\XINT_forx_b
{%
  \ifx\XINT_token\XINT_sptoken
      \xint_afterfi{\XINT_xflet\XINT_forx_b }%
  \else
      \xint_afterfi
     {\ifx\XINT_token z\expandafter\xint_gobble_iii\fi
      \the\XINT_toks }%
  \fi
}%
\def\XINT_forx_di #1#2%
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {#2}{####2}{####3}{####4}}%
  \XINT_toks {\XINT_x \XINT_forx_di {#1}}%
  \XINT_xflet\XINT_forx_e
}%
\def\XINT_forx_dii #1#2%
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {####1}{#2}{####3}{####4}}%
  \XINT_toks {\XINT_x \XINT_forx_dii {#1}}%
  \XINT_xflet\XINT_forx_e
}%
\def\XINT_forx_diii #1#2%
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {####1}{####2}{#2}{####4}}%
  \XINT_toks {\XINT_x \XINT_forx_diii {#1}}%
```

```
  \XINT_xflet\XINT_forx_e
}%
\def\XINT_forx_div #1#2%
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {####1}{####2}{####3}{#2}}%
  \XINT_toks {\XINT_x \XINT_forx_div {#1}}%
  \XINT_xflet\XINT_forx_e
}%
\def\XINT_forx_e
{%
  \ifx\XINT_token\XINT_sptoken
      \xint_afterfi{\XINT_xflet\XINT_forx_e}%
  \else
      \xint_afterfi
     {\ifx\XINT_token z\xint_afterfi{\expandafter\XINT_x \xint_gobble_iii}\fi
      \the\XINT_toks }%
  \fi
}%
```

30.15 \xintForpair, \xintForthree, \xintForfour

1.09c: experimental status. Particularly I don't know yet if {a}{b} is better for the user or worse than (a,b). I prefer the former of course. I am not very motivated to deal with spaces in the (a,b) approach which is the one (currently) followed here.

```
\def\xintForpair #1#2#3#4in#5#6#7%
{%
    \XINT_toks \expandafter{%
    \csname XINT_forii_d\romannumeral#2\endcsname {#7}}%
    \expandafter\XINT_forii_b #5,z,% THIS z HAS CATCODE 3
}%
\def\XINT_forii_b {\futurelet\XINT_token\XINT_forii_c }%
\def\XINT_forii_c
{%
  \ifx\XINT_token z\expandafter\xint_gobble_iv\fi
  \the\XINT_toks
}%
\def\XINT_forii_di #1(#2,#3),%
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {#2}{#3}{####3}{####4}}%
  \XINT_toks {\XINT_x\XINT_forii_di {#1}}%
  \futurelet\XINT_token\XINT_for_e
}%
\def\XINT_forii_dii #1(#2,#3),%
{%
  \def\XINT_y ##1##2##3##4{#1}%
```

```
  \def\XINT_x {\XINT_y {####1}{#2}{#3}{####4}}%
  \XINT_toks {\XINT_x \XINT_forii_dii {#1}}%
  \futurelet\XINT_token\XINT_for_e
}%
\def\XINT_forii_diii #1(#2,#3),%
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {####1}{####2}{#2}{#3}}%
  \XINT_toks {\XINT_x \XINT_forii_diii {#1}}%
  \futurelet\XINT_token\XINT_for_e
}%
\def\xintForthree #1#2#3in#4#5#6%
{%
    \XINT_toks \expandafter{%
    \csname XINT_foriii_d\romannumeral#2\endcsname {#6}}%
    \expandafter\XINT_foriii_b #4,z,%
}%
\def\XINT_foriii_b {\futurelet\XINT_token\XINT_foriii_c }%
\def\XINT_foriii_c
{%
  \ifx\XINT_token z\expandafter\xint_gobble_iv\fi
  \the\XINT_toks
}%
\def\XINT_foriii_di #1(#2,#3,#4),%
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {#2}{#3}{#4}{####4}}%
  \XINT_toks {\XINT_x\XINT_foriii_di {#1}}%
  \futurelet\XINT_token\XINT_for_e
}%
\def\XINT_foriii_dii #1(#2,#3,#4),%
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {####1}{#2}{#3}{#4}}%
  \XINT_toks {\XINT_x \XINT_foriii_dii {#1}}%
  \futurelet\XINT_token\XINT_for_e
}%
\def\xintForfour #1#2#3in#4#5#6%
{%
    \XINT_toks {\XINT_foriv_di {#6}}%
    \expandafter\XINT_foriv_b #4,z,%
}%
\def\XINT_foriv_b {\futurelet\XINT_token\XINT_foriv_c }%
\def\XINT_foriv_c
{%
  \ifx\XINT_token z\expandafter\xint_gobble_iv\fi
  \the\XINT_toks
}%
\def\XINT_foriv_di #1(#2,#3,#4,#5),%
```

```
{%
  \def\XINT_y ##1##2##3##4{#1}%
  \def\XINT_x {\XINT_y {#2}{#3}{#4}{#5}}%
  \XINT_toks {\XINT_x\XINT_foriv_di {#1}}%
  \futurelet\XINT_token\XINT_for_e
}%
\catcode`z 11
```

30.16 \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.

```
\def\xintAssign #1\to
{%
    \expandafter\XINT_assign_a\romannumeral-`0#1{}\to
}%
\def\XINT_assign_a #1% attention to the # at the beginning of next line
#{%
    \def\xint_temp {#1}%
    \ifx\empty\xint_temp
        \expandafter\XINT_assign_b
    \else
        \expandafter\XINT_assign_B
    \fi
}%
\def\XINT_assign_b #1#2\to #3%
{%
    \edef #3{#1}\def\xint_temp {#2}%
    \ifx\empty\xint_temp
      \else
      \xint_afterfi{\XINT_assign_a #2\to }%
    \fi
}%
\def\XINT_assign_B #1\to #2%
```

```
{%
    \edef #2{\xint_temp}%
}%
\def\xintRelaxArray #1%
{%
    \edef\XINT_restoreescapechar {\escapechar\the\escapechar\relax}%
    \escapechar -1
    \edef\xint_arrayname {\string #1}%
    \XINT_restoreescapechar
    \expandafter\let\expandafter\xint_temp
                \csname\xint_arrayname 0\endcsname
    \count 255 0
    \loop
      \global\expandafter\let
             \csname\xint_arrayname\the\count255\endcsname\relax
      \ifnum \count 255 < \xint_temp
      \advance\count 255 1
    \repeat
    \global\expandafter\let\csname\xint_arrayname 00\endcsname\relax
    \global\let #1\relax
}%
\def\xintAssignArray #1\to #2% 1.06b: #1 may now be empty
{%
    \edef\XINT_restoreescapechar {\escapechar\the\escapechar\relax }%
    \escapechar -1
    \edef\xint_arrayname {\string #2}%
    \XINT_restoreescapechar
    \count 255 0
    \expandafter\XINT_assignarray_loop \romannumeral-`0#1\xint_relax
    \csname\xint_arrayname 00\endcsname
    \csname\xint_arrayname 0\endcsname
    {\xint_arrayname}%
    #2%
}%
\def\XINT_assignarray_loop #1%
{%
    \def\xint_temp {#1}%
    \ifx\xint_braced_xint_relax\xint_temp
       \expandafter\edef\csname\xint_arrayname 0\endcsname{\the\count 255 }%
       \expandafter\expandafter\expandafter\XINT_assignarray_end_a
    \else
       \advance\count 255 1
       \expandafter\edef
          \csname\xint_arrayname\the\count 255\endcsname{\xint_temp }%
       \expandafter\XINT_assignarray_loop
    \fi
}%
\def\XINT_assignarray_end_a #1%
{%
```

```
    \expandafter\XINT_assignarray_end_b\expandafter #1%
}%
\def\XINT_assignarray_end_b #1#2#3%
{%
    \expandafter\XINT_assignarray_end_c
    \expandafter #1\expandafter #2\expandafter {#3}%
}%
\def\XINT_assignarray_end_c #1#2#3#4%
{%
    \def #4##1%
    {%
        \romannumeral0\expandafter #1\expandafter{\the\numexpr ##1}%
    }%
    \def #1##1%
    {%
        \ifnum ##1< 0
            \xint_afterfi {\xintError:ArrayIndexIsNegative\space 0}%
        \else
            \xint_afterfi {%
              \ifnum ##1>#2
                  \xint_afterfi {\xintError:ArrayIndexBeyondLimit\space 0}%
              \else
                  \xint_afterfi
                  {\expandafter\expandafter\expandafter
                   \space\csname #3##1\endcsname}%
              \fi}%
        \fi
     }%
}%
\let\xintDigitsOf\xintAssignArray
```

30.17 \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}%
```

30.18 \XINT_cuz

```
\def\xint_cleanupzeros_andstop #1#2#3#4%
{%
    \expandafter\space\the\numexpr #1#2#3#4\relax
}%
\def\xint_cleanupzeros_nospace #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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \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%
}%
\def\xint_cuz_end_b #1#2#3#4#5\Z
{%
    \expandafter\space\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 }%
```

30.19 \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 xintfracis 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}%
```

30.20 \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%
}%
\def\XINT_num_end\xint_relax\XINT_num_NumEight #1\xint_relax #2\Z
{%
    \expandafter\space\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}%
```

30.21 \xintSgn

Changed in 1.05. Earlier code was unnecessarily strange. 1.09a with \xintnum

```
\def\xintiSgn {\romannumeral0\xintisgn }%
\def\xintisgn #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{\romannumeral0\XINT_sgn #1\Z }%
\def\XINT_sgn #1#2\Z
{%
    \xint_UDzerominusfork
      #1-\dummy  { 0}%
      0#1\dummy  { -1}%
       0-\dummy  { 1}%
    \krof
}%
```

30.22 \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}}%
```

30.23 \xintSgnFork

Expandable three-way fork added in 1.07. The argument #1 must expand to -1,0 or 1. A \count should be put within a \numexpr..\relax.

```
\def\xintSgnFork {\romannumeral0\xintsgnfork }%
\def\xintsgnfork #1%
{%
    \ifcase #1 \xint_afterfi{\expandafter\space\xint_secondofthree}%
            \or\xint_afterfi{\expandafter\space\xint_thirdofthree}%
          \else\xint_afterfi{\expandafter\space\xint_firstofthree}%
```

```
    \fi
}%
```

30.24 \xintifSgn

Expandable three-way fork added in 1.09a. Branches expandably depending on whether if <0, =0, >0. The use of \romannumeral0\xintsgn rather than \xintSgn is related to the (partial) acceptability of the ternary operator : in \xintNewExpr

```
\def\xintifSgn {\romannumeral0\xintifsgn }%
\def\xintifsgn #1%
{%
    \ifcase \romannumeral0\xintsgn{#1}
               \xint_afterfi{\expandafter\space\xint_secondofthree}%
            \or\xint_afterfi{\expandafter\space\xint_thirdofthree}%
          \else\xint_afterfi{\expandafter\space\xint_firstofthree}%
    \fi
}%
```

30.25 \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).

```
\def\xintifZero {\romannumeral0\xintifzero }%
\def\xintifzero #1%
{%
    \if\xintSgn{\xintAbs{#1}}0%
       \xint_afterfi{\expandafter\space\xint_firstoftwo}%
    \else
       \xint_afterfi{\expandafter\space\xint_secondoftwo}%
    \fi
}%
\def\xintifNotZero {\romannumeral0\xintifnotzero }%
\def\xintifnotzero #1%
{%
    \if\xintSgn{\xintAbs{#1}}1%
       \xint_afterfi{\expandafter\space\xint_firstoftwo}%
    \else
       \xint_afterfi{\expandafter\space\xint_secondoftwo}%
    \fi
}%
```

30.26 \xintifTrue

```
\let\xintifTrue\xintifNotZero
```

30.27 \xintifEq

\xintifEq {n}{m}{YES if n=m}{NO if n<>m}.

```
\def\xintifEq {\romannumeral0\xintifeq }%
\def\xintifeq #1#2%
{%
    \if\xintCmp{#1}{#2}0%
                \xint_afterfi{\expandafter\space\xint_firstoftwo}%
          \else\xint_afterfi{\expandafter\space\xint_secondoftwo}%
    \fi
}%
```

30.28 \xintifGt

\xintifGt {n}{m}{YES if n>m}{NO if n<=m}.

```
\def\xintifGt {\romannumeral0\xintifgt }%
\def\xintifgt #1#2%
{%
    \if\xintCmp{#1}{#2}1%
                \xint_afterfi{\expandafter\space\xint_firstoftwo}%
          \else\xint_afterfi{\expandafter\space\xint_secondoftwo}%
    \fi
}%
```

30.29 \xintifLt

\xintifLt {n}{m}{YES if n<m}{NO if n>=m}.

```
\def\xintifLt {\romannumeral0\xintiflt }%
\def\xintiflt #1#2%
{%
    \xintSgnFork{\xintCmp{#1}{#2}}%
                {\expandafter\space\xint_firstoftwo}%
                {\expandafter\space\xint_secondoftwo}%
                {\expandafter\space\xint_secondoftwo}%
}%
```

30.30 \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-\dummy  { 0}%      zero
      0#1\dummy  { }%     negative
       0-\dummy  { -#1}%  positive
    \krof
}%
```

30.31 `\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\dummy  { }%
       -\dummy  { #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\relax.%
}%
\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\relax.%
}%
\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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \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\relax.%
}%
\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\relax.%
}%
\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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \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\relax.%
}%
\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\relax.%
}%
\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\relax.%
}%
\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\relax.%
}%
\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}%
\def\XINT_addm_end #1#2#3#4#5%
    {\expandafter\space\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}%
```

30.32 \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\dummy \XINT_add_secondiszero
      #3\dummy \XINT_add_firstiszero
       0\dummy
        {\xint_UDsignsfork
          #1#3\dummy \XINT_add_minusminus          % #1 = #3 = -
           #1-\dummy \XINT_add_minusplus           % #1 = -
           #3-\dummy \XINT_add_plusminus           % #3 = -
            --\dummy \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_andstop%
    \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
}%
```

30.33 \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\dummy \XINT_sub_minusminus
           #1-\dummy \XINT_sub_minusplus   % attention, #3=0 possible
           #3-\dummy \XINT_sub_plusminus   % attention, #1=0 possible
            --\dummy {\xint_UDzerofork
                      #1\dummy \XINT_sub_secondiszero
                      #3\dummy \XINT_sub_firstiszero
                       0\dummy \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_andstop%
    \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\relax.%
}%
```

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\dummy  \XINT_sub_C   % une retenue
       0\dummy  \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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \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\relax.%
}%
\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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
    #1\W\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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \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\relax.%
}%
\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\dummy {\XINT_sub_Fdec 0}% soustraire  1. Et faire signe -
      #1#4\dummy {\XINT_sub_Finc 1}% additionner 1. Et faire signe -
        10\dummy  \XINT_sub_DD     % terminer. Mais avec signe -
    \krof
    {#3}%
}%
\def\XINT_sub_DD {\expandafter\xint_minus_andstop\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\relax.%
}%
\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_andstop\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\relax.%
}%
\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\dummy {\expandafter\xint_minus_andstop\xint_cleanupzeros_nospace}%
      0\dummy { -1}%
    \krof
    #3%
}%
\def\xint_sub_ez\W\XINT_sub_Eenter #1%
{%
    \xint_UDzerofork
      #1\dummy \XINT_sub_K % il y a une retenue
       0\dummy \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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \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\relax.%
}%
\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_andstop
    \romannumeral0\XINT_cuz_loop #3\W\W\W\W\W\W\W\Z
}%
```

30.34 \xintCmp

Release 1.09a has \xintnum added into \xintiCmp.

```
\def\xintiCmp {\romannumeral0\xinticmp }%
```

```
\def\xinticmp #1%
{%
    \expandafter\xint_cmp\expandafter{\romannumeral0\xintnum{#1}}%
}%
\let\xintCmp\xintiCmp \let\xintcmp\xinticmp
\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\dummy \XINT_cmp_minusminus
           #1-\dummy \XINT_cmp_minusplus
           #3-\dummy \XINT_cmp_plusminus
            --\dummy {\xint_UDzerosfork
                      #1#3\dummy \XINT_cmp_zerozero
                       #10\dummy \XINT_cmp_zeroplus
                       #30\dummy \XINT_cmp_pluszero
                        00\dummy \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\relax.%
}%
\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\dummy \XINT_cmp_K              %     il y a une retenue
       0\dummy \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 }%
```

30.35 \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}}%
```

30.36 \xintIsZero, \xintIsNotZero

1.09a.

```
\def\xintIsZero {\romannumeral0\xintiszero }%
\def\xintiszero #1{\xintifsgn {#1}{0}{1}{0}}%
\def\xintIsNotZero {\romannumeral0\xintisnotzero }%
\def\xintisnotzero #1{\xintifsgn {#1}{1}{0}{1}}%
```

30.37 \xintIsTrue, \xintNot

1.09c

```
\let\xintIsTrue\xintIsNotZero
\let\xintNot\xintIsZero
```

30.38 \xintIsTrue:csv

1.09c. For use by \xinttheboolexpr.

```
\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_istrue:_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_istrue:_f ,#1#2^{\xint_gobble_i #1}%
```

30.39 \xintAND, \xintOR, \xintXOR

1.09a.

```
\def\xintAND {\romannumeral0\xintand }%
\def\xintand #1#2{\xintifzero {#1}{0}{\xintifzero {#2}{0}{1}}}%
\def\xintOR {\romannumeral0\xintor }%
\def\xintor #1#2{\xintifzero {#1}{\xintifzero {#2}{0}{1}}{1}}%
\def\xintXOR {\romannumeral0\xintxor }%
\def\xintxor #1#2{\ifcase \numexpr\xintIsZero{#1}+\xintIsZero{#2}\relax
                     \xint_afterfi{ 0}%
                  \or\xint_afterfi{ 1}%
                  \else\xint_afterfi { 0}%
                  \fi }%
```

30.40 \xintANDof

New with 1.09a. \xintANDof works 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
           {\xintifZero{#1}{\XINT_andof_no}{\XINT_andof_a}}%
\def\XINT_andof_no #1\relax { 0}%
\def\XINT_andof_e #1\Z { 1}%
```

30.41 \xintANDof:csv

1.09a. For use by \xintexpr.

```
\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,{\xintifZero{#1}{\XINT_andof:_no}{\XINT_andof:_a}}%
\def\XINT_andof:_no #1^{0}%
\def\XINT_andof:_e  #1^{1}%
```

30.42 \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
           {\xintifZero{#1}{\XINT_orof_a}{\XINT_orof_yes}}%
\def\XINT_orof_yes #1\relax { 1}%
\def\XINT_orof_e #1\Z { 0}%
```

30.43 \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,{\xintifZero{#1}{\XINT_orof:_a}{\XINT_orof:_yes}}%
\def\XINT_orof:_yes #1^{1}%
\def\XINT_orof:_e   #1^{0}%
```

30.44 \xintXORof

New with 1.09a. Works with an empty list, too.

```
\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%
           {\xintifZero {#1}{\XINT_xorof_a #2}{\ifcase #2
                                    \xint_afterfi{\XINT_xorof_a 1}%
                                    \else
                                    \xint_afterfi{\XINT_xorof_a 0}%
                                    \fi }%
           }%
\def\XINT_xorof_e #1\Z #2{ #2}%
```

30.45 \xintXORof:csv

1.09a. For use by \xintexpr.

```
\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_xorof:_e
                      \else\expandafter\XINT_xorof:_c\fi #1}%
\def\XINT_xorof:_c #1,#2%
           {\xintifZero {#1}{\XINT_xorof:_a #2}{\ifcase #2
                                       \xint_afterfi{\XINT_xorof:_a 1}%
                                       \else
                                       \xint_afterfi{\XINT_xorof:_a 0}%
                                       \fi }%
           }%
\def\XINT_xorof:_e  ,#1#2^{#1}% allows empty list
```

30.46 \xintGeq

Release 1.09a has \xintnum added into \xintiGeq. PLUS GRAND OU ÉGAL attention compare les **valeurs absolues**

```
\def\xintiGeq {\romannumeral0\xintigeq }%
\def\xintigeq #1%
{%
    \expandafter\xint_geq\expandafter {\romannumeral0\xintnum{#1}}%
}%
\let\xintGeq\xintiGeq \let\xintgeq\xintigeq
\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\dummy \XINT_geq_secondiszero % |#1#2|=0
      #3\dummy \XINT_geq_firstiszero % |#1#2|>0
       0\dummy {\xint_UDsignsfork
                  #1#3\dummy \XINT_geq_minusminus
                   #1-\dummy \XINT_geq_minusplus
                   #3-\dummy \XINT_geq_plusminus
                    --\dummy \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\relax.%
}%
\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\dummy { 0}             %     il y a une retenue
       0\dummy { 1}             %     pas de retenue
    \krof
}%
```

30.47 \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\dummy \XINT_max_minusminus  % A < 0, B < 0
           #1-\dummy \XINT_max_minusplus   % B < 0, A >= 0
           #3-\dummy \XINT_max_plusminus   % A < 0, B >= 0
            --\dummy {\xint_UDzerosfork
                      #1#3\dummy \XINT_max_zerozero % A = B = 0
                       #10\dummy \XINT_max_zeroplus % B = 0, A > 0
                       #30\dummy \XINT_max_pluszero % A = 0, B > 0
                        00\dummy \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_andstop }%
```

```
\def\XINT_max_zeroplus  #1#2#3#4{\xint_firstoftwo_andstop }%
\def\XINT_max_pluszero  #1#2#3#4{\xint_secondoftwo_andstop }%
\def\XINT_max_minusplus #1#2#3#4{\xint_firstoftwo_andstop }%
\def\XINT_max_plusminus #1#2#3#4{\xint_secondoftwo_andstop }%
\def\XINT_max_plusplus  #1#2#3#4%
{%
    \ifodd\XINT_Geq {#4#2}{#3#1}
      \expandafter\xint_firstoftwo_andstop
    \else
      \expandafter\xint_secondoftwo_andstop
    \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_andstop
    \else
      \expandafter\xint_secondoftwo_andstop
    \fi
}%
```

30.48 \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
```

30.49 \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\dummy \XINT_min_minusminus  % A < 0, B < 0
           #1-\dummy \XINT_min_minusplus   % B < 0, A >= 0
           #3-\dummy \XINT_min_plusminus   % A < 0, B >= 0
            --\dummy {\xint_UDzerosfork
                        #1#3\dummy \XINT_min_zerozero % A = B = 0
                         #10\dummy \XINT_min_zeroplus % B = 0, A > 0
                         #30\dummy \XINT_min_pluszero % A = 0, B > 0
                          00\dummy \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_andstop }%
\def\XINT_min_zeroplus  #1#2#3#4{\xint_secondoftwo_andstop }%
\def\XINT_min_pluszero  #1#2#3#4{\xint_firstoftwo_andstop }%
\def\XINT_min_minusplus #1#2#3#4{\xint_secondoftwo_andstop }%
\def\XINT_min_plusminus #1#2#3#4{\xint_firstoftwo_andstop }%
\def\XINT_min_plusplus  #1#2#3#4%
{%
    \ifodd\XINT_Geq {#4#2}{#3#1}
      \expandafter\xint_secondoftwo_andstop
    \else
      \expandafter\xint_firstoftwo_andstop
    \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_andstop
    \else
      \expandafter\xint_firstoftwo_andstop
```

```
    \fi
}%
```

30.50 \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
```

30.51 \xintSum, \xintSumExpr

\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 \xintnum (I would need for this to re-organize the code first).

```
\def\xintiSum {\romannumeral0\xintisum }%
\def\xintisum #1{\xintisumexpr #1\relax }%
\def\xintiSumExpr {\romannumeral0\xintisumexpr }%
\def\xintisumexpr {\expandafter\XINT_sumexpr\romannumeral-‘0}%
\let\xintSum\xintiSum \let\xintsum\xintisum
\let\xintSumExpr\xintiSumExpr \let\xintsumexpr\xintisumexpr
\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\dummy \XINT_sum_N
       -\dummy {\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}}%
```

30.52 \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 dig-
its 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 be-
sides actually computing their lengths.
```

```
\def\XINT_mul_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\dummy \XINT_mul_zero
      #3\dummy \XINT_mul_zero
       0\dummy
        {\xint_UDsignsfork
          #1#3\dummy \XINT_mul_minusminus          % #1 = #3 = -
           #1-\dummy {\XINT_mul_minusplus #3}%          % #1 = -
           #3-\dummy {\XINT_mul_plusminus #1}%          % #3 = -
            --\dummy {\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\XINT_length {#2}}%
      {\romannumeral0\XINT_length {#1}}{#1}{#2}%
}%
\def\XINT_mul_minusplus #1#2#3%
{%
    \expandafter\xint_minus_andstop\romannumeral0\expandafter
    \XINT_mul_choice_a
    \expandafter{\romannumeral0\XINT_length {#1#3}}%
    {\romannumeral0\XINT_length {#2}}{#2}{#1#3}%
}%
\def\XINT_mul_plusminus #1#2#3%
{%
    \expandafter\xint_minus_andstop\romannumeral0\expandafter
    \XINT_mul_choice_a
    \expandafter{\romannumeral0\XINT_length {#3}}%
    {\romannumeral0\XINT_length {#1#2}}{#1#2}{#3}%
}%
\def\XINT_mul_plusplus #1#2#3#4%
{%
    \expandafter\XINT_mul_choice_a
```

```
    \expandafter{\romannumeral0\XINT_length {#2#4}}%
    {\romannumeral0\XINT_length {#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\relax.{#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 *nettoy-
age 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\XINT_rev{#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%
}%
\def\XINT_mul_M_end #1#2#3#4#5#6#7#8%
{%
    \expandafter\space\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\XINT_rev{#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}%
```

30.53 \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
}%
```

30.54 \xintPrd, \xintPrdExpr

\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.

```
\def\xintiPrd {\romannumeral0\xintiprd }%
\def\xintiprd #1{\xintiprdexpr #1\relax }%
\let\xintPrd\xintiPrd
\let\xintprd\xintiprd
\def\xintiPrdExpr {\romannumeral0\xintiprdexpr }%
\def\xintiprdexpr {\expandafter\XINT_prdexpr\romannumeral-`0}%
\let\xintPrdExpr\xintiPrdExpr
\let\xintprdexpr\xintiprdexpr
\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}%
```

30.55 \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\XINT_Length {#1}>9 but for the time being I leave things as they stand. With release 1.05, rather than using \XINT_Length 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.

```
\def\xintiFac {\romannumeral0\xintifac }%
\def\xintifac #1%
{%
    \expandafter\XINT_fac_fork\expandafter{\the\numexpr #1}%
}%
\def\xintFac {\romannumeral0\xintfac }%
\def\xintfac #1%
{%
    \expandafter\XINT_fac_fork\expandafter{\romannumeral0\xintnum{#1}}%
}%
\def\XINT_fac_fork #1%
{%
    \ifcase\XINT_Sgn {#1}
       \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>999999
         \xint_afterfi{\expandafter\xintError:FactorialOfTooBigNumber
                       \expandafter\space\expandafter 1\xint_gobble_i }%
    \else
         \xint_afterfi{\ifnum #1>9999
                         \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
}%
```

30.56 \xintPow

1.02 modified the \XINT_posprod routine, and this meant that the original version was moved here and renamed to \XINT_pow_posprod, as it was well adapted for computing powers. Then I moved in 1.03 the special variants of multiplication (hence of addition) which were needed to earlier in this file. Modified in 1.06, the exponent is given to a \numexpr rather than twice expanded. \xintnum added in 1.09a.

```
\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\dummy \XINT_pow_Aneg
       -\dummy \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_Sgn {#1}
         \xint_afterfi { 1}%
     \or
         \xint_afterfi { 0}%
     \else
         \xint_afterfi {\xintError:DivisionByZero\space 0}%
     \fi
}%
\def\XINT_pow_AatleastTwo #1%
{%
    \ifcase\XINT_Sgn {#1}
        \expandafter\XINT_pow_BisZero
    \or
        \expandafter\XINT_pow_checkBsize
    \else
        \expandafter\XINT_pow_BisNegative
    \fi
    {#1}%
}%
\def\XINT_pow_BisNegative #1#2{\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#2%
{%
    \ifnum #1>999999999
        \expandafter\XINT_pow_BtooBig
    \else
        \expandafter\XINT_pow_loop
    \fi
    {#1}{#2}\XINT_pow_posprod
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
}%
\def\XINT_pow_BtooBig #1\xint_relax #2\xint_relax
                         {\xintError:ExponentTooBig\space 0}%
\def\XINT_pow_loop #1#2%
{%
```

```
    \ifnum #1 = 1
        \expandafter\XINT_pow_loop_end
    \else
        \xint_afterfi{\expandafter\XINT_pow_loop_a
            \expandafter{\the\numexpr 2*(#1/2)-#1\expandafter }% b mod 2
            \expandafter{\the\numexpr #1-#1/2\expandafter }%     [b/2]
            \expandafter{\romannumeral0\xintiisqr{#2}}}%
    \fi
    {{#2}}%
}%
\def\XINT_pow_loop_end {\romannumeral0\XINT_rord_main {}\relax }%
\def\XINT_pow_loop_a #1%
{%
    \ifnum #1 = 1
        \expandafter\XINT_pow_loop
    \else
        \expandafter\XINT_pow_loop_throwaway
    \fi
}%
\def\XINT_pow_loop_throwaway #1#2#3%
{%
   \XINT_pow_loop {#1}{#2}%
}%
```

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.

```
\def\XINT_pow_posprod #1%
{%
    \XINT_pow_pprod_checkifempty #1\Z
}%
\def\XINT_pow_pprod_checkifempty #1%
{%
    \xint_gob_til_relax #1\XINT_pow_pprod_emptyproduct\relax
    \XINT_pow_pprod_RQfirst #1%
}%
\def\XINT_pow_pprod_emptyproduct #1\Z { 1}%
\def\XINT_pow_pprod_RQfirst #1\Z
{%
    \expandafter\XINT_pow_pprod_getnext\expandafter
    {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z}%
}%
\def\XINT_pow_pprod_getnext #1#2%
{%
    \XINT_pow_pprod_checkiffinished #2\Z {#1}%
}%
```

```
\def\XINT_pow_pprod_checkiffinished #1%
{%
    \xint_gob_til_relax #1\XINT_pow_pprod_end\relax
    \XINT_pow_pprod_compute #1%
}%
\def\XINT_pow_pprod_compute #1\Z #2%
{%
    \expandafter\XINT_pow_pprod_getnext\expandafter
    {\romannumeral0\XINT_mulr_enter #2\Z\Z\Z\Z #1\W\W\W\W }%
}%
\def\XINT_pow_pprod_end\relax\XINT_pow_pprod_compute #1\Z #2%
{%
    \expandafter\xint_cleanupzeros_andstop
    \romannumeral0\XINT_rev {#2}%
}%
```

30.57 \xintDivision, \xintQuo, \xintRem

1.09a inserts the use of \xintnum

```
\def\xintiQuo {\romannumeral0\xintiquo }%
\def\xintiRem {\romannumeral0\xintirem }%
\def\xintiquo {\expandafter\xint_firstoftwo_andstop
               \romannumeral0\xintidivision }%
\def\xintirem {\expandafter\xint_secondoftwo_andstop
               \romannumeral0\xintidivision }%
\let\xintQuo\xintiQuo \let\xintquo\xintiquo
\let\xintRem\xintiRem \let\xintrem\xintirem
```

#1 = A, #2 = B. On calcule le quotient de A par B.
1.03 adds the detection of 1 for B.

```
\def\xintiDivision {\romannumeral0\xintidivision }%
\def\xintidivision #1%
{%
    \expandafter\xint_division\expandafter {\romannumeral0\xintnum{#1}}%
}%
\let\xintDivision\xintiDivision \let\xintdivision\xintidivision
\def\xint_division #1#2%
{%
    \expandafter\XINT_div_fork \romannumeral0\xintnum{#2}\Z #1\Z
}%
\def\XINT_Division #1#2{\romannumeral0\XINT_div_fork #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\dummy \XINT_div_BisZero
      #3\dummy \XINT_div_AisZero
```

```
        0\dummy
         {\xint_UDsignfork
            #1\dummy \XINT_div_BisNegative  % B < 0
            #3\dummy \XINT_div_AisNegative  % A < 0, B > 0
             -\dummy \XINT_div_plusplus     % B > 0, A > 0
          \krof }%
    \krof
    {#2}{#4}#1#3% #1#2=B, #3#4=A
}%
\def\XINT_div_BisZero #1#2#3#4{\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.

```
\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_post
    \romannumeral0\XINT_div_fork #1\Z #4#2\Z
}%
\def\XINT_div_BisNegative_post #1%
{%
    \expandafter\space\expandafter {\romannumeral0\XINT_opp #1}%
}%
```

B = #3#1 > 0, A =-#2< 0

```
\def\XINT_div_AisNegative #1#2#3#4%
{%
    \expandafter\XINT_div_AisNegative_post
    \romannumeral0\XINT_div_prepare {#3#1}{#2}{#3#1}%
}%
\def\XINT_div_AisNegative_post #1#2%
{%
    \ifcase\XINT_Sgn {#2}
       \expandafter \XINT_div_AisNegative_zerorem
    \or
       \expandafter \XINT_div_AisNegative_posrem
    \fi
    {#1}{#2}%
```

```
}%
```

en #3 on a une copie de B (à l'endroit)

```
\def\XINT_div_AisNegative_zerorem #1#2#3%
{%
    \expandafter\space\expandafter {\romannumeral0\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_posrem #1%
{%
    \expandafter \XINT_div_AisNegative_posrem_b \expandafter
       {\romannumeral0\xintiiopp{\xintInc {#1}}}%
}%
\def\XINT_div_AisNegative_posrem_b #1#2#3%
{%
    \expandafter \xint_exchangetwo_keepbraces_andstop \expandafter
    {\romannumeral0\XINT_sub {#3}{#2}}{#1}%
}%
```

par la suite A et B sont > 0. #1 = B. Pour le moment à l'endroit. Calcul du plus petit K = 4n >= longueur de B
1.03 adds the interception of B=1

```
\def\XINT_div_prepare #1%
{%
    \expandafter \XINT_div_prepareB_aa \expandafter
        {\romannumeral0\XINT_length {#1}}{#1}% B > 0 ici
}%
\def\XINT_div_prepareB_aa #1%
{%
    \ifnum #1=1
      \expandafter\XINT_div_prepareB_ab
    \else
      \expandafter\XINT_div_prepareB_a
    \fi
    {#1}%
}%
\def\XINT_div_prepareB_ab #1#2%
{%
    \ifnum #2=1
      \expandafter\XINT_div_prepareB_BisOne
    \else
      \expandafter\XINT_div_prepareB_e
    \fi {000}{3}{4}{#2}%
}%
\def\XINT_div_prepareB_BisOne #1#2#3#4#5{ {#5}{0}}%
\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}%
}%
```

#1 = K

```
\def\XINT_div_prepareB_c #1#2%
{%
    \ifcase \numexpr #1-#2\relax
       \expandafter\XINT_div_prepareB_d
    \or
       \expandafter\XINT_div_prepareB_di
    \or
       \expandafter\XINT_div_prepareB_dii
    \or
       \expandafter\XINT_div_prepareB_diii
    \fi {#1}%
}%
\def\XINT_div_prepareB_d    {\XINT_div_prepareB_e {}{0}}%
\def\XINT_div_prepareB_di   {\XINT_div_prepareB_e {0}{1}}%
\def\XINT_div_prepareB_dii  {\XINT_div_prepareB_e {00}{2}}%
\def\XINT_div_prepareB_diii {\XINT_div_prepareB_e {000}{3}}%
```

#1 = zéros à rajouter à B, #2=c, #3=K, #4 = B

```
\def\XINT_div_prepareB_e #1#2#3#4%
{%
    \XINT_div_prepareB_f #4#1\Z {#3}{#2}{#1}%
}%
```

x = #1#2#3#4 = 4 premiers chiffres de B. #1 est non nul. Ensuite on renverse B pour calculs plus rapides par la suite.

```
\def\XINT_div_prepareB_f #1#2#3#4#5\Z
{%
    \expandafter \XINT_div_prepareB_g \expandafter
        {\romannumeral0\XINT_rev {#1#2#3#4#5}}{#1#2#3#4}%
}%
```

#3= K, #4 = c, #5= {} ou {0} ou {00} ou {000}, #6 = A initial #1 = B préparé et renversé, #2 = x = quatre premiers chiffres On multiplie aussi A par 10^c. B, x, K, c, {} ou {0} ou {00} ou {000}, A initial

```
\def\XINT_div_prepareB_g #1#2#3#4#5#6%
{%
    \XINT_div_prepareA_a {#6#5}{#2}{#3}{#1}{#4}%
}%
```

A, x, K, B, c,

```
\def\XINT_div_prepareA_a #1%
{%
```

```
    \expandafter \XINT_div_prepareA_b \expandafter
       {\romannumeral0\XINT_length {#1}}{#1}% A >0 ici
}%
```

L0, A, x, K, B, ...

```
\def\XINT_div_prepareA_b #1%
{%
  \expandafter\XINT_div_prepareA_c\expandafter{\the\numexpr 4*((#1+1)/4)}{#1}%
}%
```

L, L0, A, x, K, B,...

```
\def\XINT_div_prepareA_c #1#2%
{%
    \ifcase \numexpr #1-#2\relax
       \expandafter\XINT_div_prepareA_d
    \or
       \expandafter\XINT_div_prepareA_di
    \or
       \expandafter\XINT_div_prepareA_dii
    \or
       \expandafter\XINT_div_prepareA_diii
    \fi {#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,

```
\def\XINT_div_prepareA_e #1#2#3%
{%
    \XINT_div_startswitch {#1#3}{#2}%
}%
```

A, L, x, K, B, c

```
\def\XINT_div_startswitch #1#2#3#4%
{%
    \ifnum #2 > #4
      \expandafter\XINT_div_body_a
    \else
    \ifnum #2 = #4
      \expandafter\expandafter\expandafter\XINT_div_final_a
    \else
      \expandafter\expandafter\expandafter\XINT_div_finished_a
    \fi\fi {#1}{#4}{#3}{0000}{#2}%
}%
```

---- "Finished": A, K, x, Q, L, B, c

```
\def\XINT_div_finished_a #1#2#3%
{%
 \expandafter\XINT_div_finished_b\expandafter {\romannumeral0\XINT_cuz {#1}}%
}%
```

A, Q, L, B, c no leading zeros in A at this stage

```
\def\XINT_div_finished_b #1#2#3#4#5%
{%
    \ifcase \XINT_Sgn {#1}
       \xint_afterfi {\XINT_div_finished_c {0}}%
    \or
       \xint_afterfi {\expandafter\XINT_div_finished_c\expandafter
                      {\romannumeral0\XINT_dsh_checksignx #5\Z {#1}}%
                     }%
    \fi
    {#2}%
}%
\def\XINT_div_finished_c #1#2%
{%
    \expandafter\space\expandafter {\romannumeral0\XINT_rev_andcuz {#2}}{#1}%
}%
```

---- "Final": A, K, x, Q, L, B, c

```
\def\XINT_div_final_a #1%
{%
    \XINT_div_final_b #1\Z
}%
\def\XINT_div_final_b #1#2#3#4#5\Z
{%
    \xint_gob_til_zeros_iv #1#2#3#4\xint_div_final_c0000%
    \XINT_div_final_c {#1#2#3#4}{#1#2#3#4#5}%
}%
\def\xint_div_final_c0000\XINT_div_final_c #1{\XINT_div_finished_a }%
```

a, A, K, x, Q, L, B ,c 1.01: code ré-écrit pour optimisations diverses. 1.04: again, code rewritten for tiny speed increase (hopefully).

```
\def\XINT_div_final_c #1#2#3#4%
{%
    \expandafter \XINT_div_final_da \expandafter
    {\the\numexpr #1-(#1/#4)*#4\expandafter }\expandafter
    {\the\numexpr #1/#4\expandafter }\expandafter
    {\romannumeral0\xint_cleanupzeros_andstop #2}%
}%
```

r, q, A sans leading zéros, Q, L, B à l'envers sur 4n, c

```
\def\XINT_div_final_da #1%
{%
    \ifnum #1>\xint_c_ix
```

```
        \expandafter\XINT_div_final_dP
    \else
        \xint_afterfi
        {\ifnum #1<\xint_c_
         \expandafter\XINT_div_final_dN
         \else
         \expandafter\XINT_div_final_db
         \fi }%
    \fi
}%
\def\XINT_div_final_dN #1%
{%
    \expandafter\XINT_div_final_dP\the\numexpr #1-\xint_c_i\relax
}%
\def\XINT_div_final_dP #1#2#3#4#5% q,A,Q,L,B (puis c)
{%
    \expandafter \XINT_div_final_f \expandafter
    {\romannumeral0\xintiisub {#2}%
        {\romannumeral0\XINT_mul_M {#1}#5\Z\Z\Z\Z }}%
    {\romannumeral0\XINT_add_A 0{}#1000\W\X\Y\Z #3\W\X\Y\Z }%
}%
\def\XINT_div_final_db #1#2#3#4#5% q,A,Q,L,B (puis c)
{%
    \expandafter\XINT_div_final_dc\expandafter
    {\romannumeral0\xintiisub {#2}%
        {\romannumeral0\XINT_mul_M {#1}#5\Z\Z\Z\Z }}%
    {#1}{#2}{#3}{#4}{#5}%
}%
\def\XINT_div_final_dc #1#2%
{%
    \ifnum\XINT_Sgn{#1}<\xint_c_
    \xint_afterfi
    {\expandafter\XINT_div_final_dP\the\numexpr #2-\xint_c_i\relax}%
    \else \xint_afterfi {\XINT_div_final_e {#1}#2}%
    \fi
}%
\def\XINT_div_final_e #1#2#3#4#5#6% A final, q, trash, Q, L, B
{%
    \XINT_div_final_f {#1}%
    {\romannumeral0\XINT_add_A 0{}#2000\W\X\Y\Z #4\W\X\Y\Z }%
}%
\def\XINT_div_final_f #1#2#3% R,Q \`a d\'evelopper,c
{%
    \ifcase \XINT_Sgn {#1}
        \xint_afterfi {\XINT_div_final_end {0}}%
    \or
        \xint_afterfi {\expandafter\XINT_div_final_end\expandafter
                        {\romannumeral0\XINT_dsh_checksignx #3\Z {#1}}%
                      }%
```

```
    \fi
    {#2}%
}%
\def\XINT_div_final_end #1#2%
{%
    \expandafter\space\expandafter {#2}{#1}%
}%
```

Boucle Principale (on reviendra en div_body_b pas div_body_a)
A, K, x, Q, L, B, c

```
\def\XINT_div_body_a #1%
{%
    \XINT_div_body_b #1\Z {#1}%
}%
\def\XINT_div_body_b #1#2#3#4#5#6#7#8#9\Z
{%
    \XINT_div_body_c {#1#2#3#4#5#6#7#8}%
}%
```

a, A, K, x, Q, L, B, c

```
\def\XINT_div_body_c #1#2#3%
{%
    \XINT_div_body_d {#3}{}#2\Z {#1}{#3}%
}%
\def\XINT_div_body_d #1#2#3#4#5#6%
{%
    \ifnum #1 >\xint_c_
        \expandafter\XINT_div_body_d
        \expandafter{\the\numexpr #1-\xint_c_iv\expandafter }%
    \else
        \expandafter\XINT_div_body_e
    \fi
    {#6#5#4#3#2}%
}%
\def\XINT_div_body_e #1#2\Z #3%
{%
    \XINT_div_body_f {#3}{#1}{#2}%
}%
```

a, alpha (à l'envers), alpha' (à l'endroit), K, x, Q, L, B (à l'envers), c

```
\def\XINT_div_body_f #1#2#3#4#5#6#7#8%
{%
    \expandafter\XINT_div_body_gg
    \the\numexpr (#1+(#5+\xint_c_i)/\xint_c_ii)/(#5+\xint_c_i)+99999\relax
    {#8}{#2}{#8}{#4}{#5}{#3}{#6}{#7}{#8}%
}%
```

q1 sur six chiffres (il en a 5 au max), B, alpha, B, K, x, alpha', Q, L, B, c

```
\def\XINT_div_body_gg #1#2#3#4#5#6%
{%
    \xint_UDzerofork
      #2\dummy \XINT_div_body_gk
       0\dummy {\XINT_div_body_ggk #2}%
    \krof
    {#3#4#5#6}%
}%
\def\XINT_div_body_gk #1#2#3%
{%
    \expandafter\XINT_div_body_h
    \romannumeral0\XINT_div_sub_xpxp
    {\romannumeral0\XINT_mul_Mr {#1}#2\Z\Z\Z\Z }{#3}\Z {#1}%
}%
\def\XINT_div_body_ggk #1#2#3%
{%
    \expandafter \XINT_div_body_gggk \expandafter
    {\romannumeral0\XINT_mul_Mr {#1}0000#3\Z\Z\Z\Z }%
    {\romannumeral0\XINT_mul_Mr {#2}#3\Z\Z\Z\Z }%
    {#1#2}%
}%
\def\XINT_div_body_gggk #1#2#3#4%
{%
    \expandafter\XINT_div_body_h
    \romannumeral0\XINT_div_sub_xpxp
    {\romannumeral0\expandafter\XINT_mul_Ar
     \expandafter0\expandafter{\expandafter}#2\W\X\Y\Z #1\W\X\Y\Z }%
    {#4}\Z {#3}%
}%
```

alpha1 = alpha-q1 B, \Z, q1, B, K, x, alpha', Q, L, B, c

```
\def\XINT_div_body_h #1#2#3#4#5#6#7#8#9\Z
{%
    \ifnum #1#2#3#4>\xint_c_
        \xint_afterfi{\XINT_div_body_i {#1#2#3#4#5#6#7#8}}%
    \else
        \expandafter\XINT_div_body_k
    \fi
    {#1#2#3#4#5#6#7#8#9}%
}%
\def\XINT_div_body_k #1#2#3%
{%
    \XINT_div_body_l {#1}{#2}%
}%
```

a1, alpha1 (à l'endroit), q1, B, K, x, alpha', Q, L, B, c

```
\def\XINT_div_body_i #1#2#3#4#5#6%
{%
    \expandafter\XINT_div_body_j
```

```
    \expandafter{\the\numexpr (#1+(#6+1)/2)/(#6+1)-1}%
       {#2}{#3}{#4}{#5}{#6}%
}%
\def\XINT_div_body_j #1#2#3#4%
{%
    \expandafter \XINT_div_body_l \expandafter
    {\romannumeral0\XINT_div_sub_xpxp
       {\romannumeral0\XINT_mul_Mr {#1}#4\Z\Z\Z\Z }{\XINT_Rev{#2}}}%
    {#3+#1}%
}%
```

alpha2 (à l'endroit, ou alpha1), q1+q2 (ou q1), K, x, alpha', Q, L, B, c

```
\def\XINT_div_body_l #1#2#3#4#5#6#7%
{%
   \expandafter\XINT_div_body_m
   \the\numexpr \xint_c_x^viii+#2\relax {#6}{#3}{#7}{#1#5}{#4}%
}%
```

chiffres de q, Q, K, L, A'=nouveau A, x, B, c

```
\def\XINT_div_body_m 1#1#2#3#4#5#6#7#8%
{%
    \ifnum #1#2#3#4>\xint_c_
       \xint_afterfi {\XINT_div_body_n {#8#7#6#5#4#3#2#1}}%
    \else
       \xint_afterfi {\XINT_div_body_n {#8#7#6#5}}%
    \fi
}%
```

q renversé, Q, K, L, A', x, B, c

```
\def\XINT_div_body_n #1#2%
{%
    \expandafter\XINT_div_body_o\expandafter
    {\romannumeral0\XINT_addr_A 0{}#1\W\X\Y\Z #2\W\X\Y\Z }%
}%
```

q+Q, K, L, A', x, B, c

```
\def\XINT_div_body_o #1#2#3#4%
{%
    \XINT_div_body_p {#3}{#2}{}#4\Z {#1}%
}%
```

L, K, {}, A'\Z, q+Q, x, B, c

```
\def\XINT_div_body_p #1#2#3#4#5#6#7%
{%
    \ifnum #1 > #2
        \xint_afterfi
        {\ifnum #4#5#6#7 > \xint_c_
           \expandafter\XINT_div_body_q
```

```
        \else
          \expandafter\XINT_div_body_repeatp
        \fi }%
    \else
        \expandafter\XINT_div_gotofinal_a
    \fi
    {#1}{#2}{#3}#4#5#6#7%
}%
```

L, K, zeros, A' avec moins de zéros\Z, q+Q, x, B, c

```
\def\XINT_div_body_repeatp #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_div_body_p\expandafter{\the\numexpr #1-4}{#2}{0000#3}%
}%
```

L -> L-4, zeros->zeros+0000, répéter jusqu'à ce que soit L=K soit on ne trouve plus 0000
nouveau L, K, zeros, nouveau A=#4, \Z, Q+q (à l'envers), x, B, c

```
\def\XINT_div_body_q #1#2#3#4\Z #5#6%
{%
    \XINT_div_body_b #4\Z {#4}{#2}{#6}{#3#5}{#1}%
}%
```

A, K, x, Q, L, B, c --> iterate
Boucle Principale achevée. ATTENTION IL FAUT AJOUTER 4 ZEROS DE MOINS QUE CEUX QUI ONT ÉTÉ PRÉPARÉS DANS #3!!
L, K (L=K), zeros, A\Z, Q, x, B, c

```
\def\XINT_div_gotofinal_a #1#2#3#4\Z %
{%
    \XINT_div_gotofinal_b #3\Z {#4}{#1}%
}%
\def\XINT_div_gotofinal_b 0000#1\Z #2#3#4#5%
{%
    \XINT_div_final_a {#2}{#3}{#5}{#1#4}{#3}%
}%
```

La soustraction spéciale.
Elle fait l'expansion (une fois pour le premier, deux fois pour le second) de ses arguments. Ceux-ci doivent être à l'envers sur 4n. De plus on sait a priori que le second est > le premier. Et le résultat de la différence est renvoyé **avec la même longueur que le second** (donc avec des leading zéros éventuels), et *à l'endroit*.

```
\def\XINT_div_sub_xpxp #1%
{%
    \expandafter \XINT_div_sub_xpxp_a  \expandafter{#1}%
}%
\def\XINT_div_sub_xpxp_a #1#2%
{%
```

```
    \expandafter\expandafter\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#5#4#3#2-#6+#1-\xint_c_i\relax.%
}%
\def\XINT_div_sub_backtoA #1#2#3.#4%
{%
    \XINT_div_sub_A #2{#3#4}%
}%
\def\xint_div_sub_bz\W\XINT_div_sub_onestep #1#2#3#4#5#6#7%
{%
    \xint_UDzerofork
      #1\dummy  \XINT_div_sub_C   %
       0\dummy  \XINT_div_sub_D   % pas de retenue
    \krof
    {#7}#2#3#4#5%
}%
\def\XINT_div_sub_D #1#2\W\X\Y\Z
{%
    \expandafter\space
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
    #1%
}%
\def\XINT_div_sub_C #1#2#3#4#5%
{%
    \xint_gob_til_W #2\xint_div_sub_cz\W
    \XINT_div_sub_AC_onestep {#5#4#3#2}{#1}%
```

```
}%
\def\XINT_div_sub_AC_onestep #1%
{%
    \expandafter\XINT_div_sub_backtoC\the\numexpr 11#1-\xint_c_i\relax.%
}%
\def\XINT_div_sub_backtoC #1#2#3.#4%
{%
    \XINT_div_sub_AC_checkcarry #2{#3#4}% la retenue va \^etre examin\'ee
}%
\def\XINT_div_sub_AC_checkcarry #1%
{%
    \xint_gob_til_one #1\xint_div_sub_AC_nocarry 1\XINT_div_sub_C
}%
\def\xint_div_sub_AC_nocarry 1\XINT_div_sub_C #1#2\W\X\Y\Z
{%
    \expandafter\space
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
    #1%
}%
\def\xint_div_sub_cz\W\XINT_div_sub_AC_onestep #1#2{ #2}%
\def\xint_div_sub_az\W\XINT_div_sub_B #1#2#3#4\Z { #3}%
```


DECIMAL OPERATIONS: FIRST DIGIT, LASTDIGIT, ODDNESS, MULTIPLICATION BY TEN, QUOTIENT BY TEN, QUOTIENT OR MULTIPLICATION BY POWER OF TEN, SPLIT OPERATION.

30.58 \xintFDg

FIRST DIGIT. Code simplified in 1.05. And prepared for redefinition by xintfrac to parse through \xintNum. Version 1.09a inserts the \xintnum here.

```
\def\xintiFDg {\romannumeral0\xintifdg }%
\def\xintifdg #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-\dummy  { 0}%   zero
      0#1\dummy  { #2}%  negative
       0-\dummy  { #1}%  positive
    \krof
}%
```

30.59 \xintLDg

LAST DIGIT. Simplified in 1.05. And prepared for extension by xintfrac to parse through \xintNum. 1.09a has it here.

```
\def\xintiLDg {\romannumeral0\xintildg }%
\def\xintildg #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\XINT_rev {#1}\Z
}%
\def\XINT_ldg_ #1#2\Z{ #1}%
```

30.60 \xintMON

MINUS ONE TO THE POWER N

```
\def\xintiMON {\romannumeral0\xintimon }%
\def\xintimon #1%
{%
    \ifodd\xintiLDg {#1}
        \xint_afterfi{ -1}%
    \else
        \xint_afterfi{ 1}%
    \fi
}%
\def\xintiMMON {\romannumeral0\xintimmon }%
\def\xintimmon #1%
{%
    \ifodd\xintiLDg {#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
}%
```

30.61 \xintOdd

ODDNESS. 1.05 defines \xintiOdd, so \xintOdd can be modified by xintfrac to parse through \xintNum.

```
\def\xintiOdd {\romannumeral0\xintiodd }%
\def\xintiodd #1%
{%
    \ifodd\xintiLDg{#1}
        \xint_afterfi{ 1}%
    \else
        \xint_afterfi{ 0}%
    \fi
}%
\def\XINT_Odd #1%
{\romannumeral0%
    \ifodd\XINT_LDg{#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
}%
```

30.62 \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}%
```

30.63 \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\XINT_rev
}%
\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\XINT_rev {#2#3}%
}%
\def\xint_dsr_onedigit #1\XINT_rev #2{ 0}%
\def\XINT_dsr_removew #1\W { }%
```

30.64 \xintDSH, \xintDSHr

DECIMAL SHIFTS \xintDSH {x}{A}
si x <= 0, fait A -> A.10^(|x|). v1.03 corrige l'oversight pour A=0.n 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 revise the legacy code on another occasion.

```
\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\dummy \XINT_dshr_xzeroorneg
      #1-\dummy \XINT_dshr_xzeroorneg
       0-\dummy \XINT_dshr_xpositive
    \krof #1%
}%
\def\XINT_dshr_xzeroorneg #1\Z #2{ 0}%
\def\XINT_dshr_xpositive #1\Z
{%
    \expandafter\xint_secondoftwo_andstop\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-\dummy  \XINT_dsh_xiszero
      0#1\dummy  \XINT_dsx_xisNeg_checkA     % on passe direct dans DSx
       0-\dummy  {\XINT_dsh_xisPos #1}%
    \krof
}%
\def\XINT_dsh_xiszero #1\Z #2{ #2}%
\def\XINT_dsh_xisPos #1\Z #2%
{%
    \expandafter\xint_firstoftwo_andstop
```

```
    \romannumeral0\XINT_dsx_checksignA #2\Z {#1}% via DSx
}%
```

30.65 \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.

In 1.07, I decide to modify the coding of \XINT_dsx_zeroloop, to avoid impacting the input stack (which prevented doing truncation or rounding or float with more than eight times the size of input stack; 40000 = 8x5000 digits on my installation.) I think this was the only place in the code with such non tail recursion, as I recall being careful to avoid problems within the Factorial and Power routines, but I would need to check. Too tired now after having finished \xintexpr, \xintNewExpr, and \xintfloatexpr!

```
\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-\dummy  \XINT_dsx_xisZero
      0#1\dummy  \XINT_dsx_xisNeg_checkA
       0-\dummy  {\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>999999999
       \xint_afterfi
       {\xintError:TooBigDecimalShift
        \expandafter\space\expandafter 0\xint_gobble_iv }%
    \else
       \expandafter \XINT_dsx_zeroloop
    \fi
}%
\def\XINT_dsx_zeroloop #1#2%
{%
    \ifnum #1<9 \XINT_dsx_exita\fi
    \expandafter\XINT_dsx_zeroloop\expandafter
        {\the\numexpr #1-8}{#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-\dummy  \XINT_dsx_AisZero
      0#1\dummy  \XINT_dsx_AisNeg
       0-\dummy  {\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}}%
}%
\def\XINT_dsx_end #1#2%
{%
    \expandafter\space\expandafter{#2}{#1}%
}%
```

30.66 \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.

```
\def\xintDecSplitL {\romannumeral0\xintdecsplitl }%
\def\xintDecSplitR {\romannumeral0\xintdecsplitr }%
\def\xintdecsplitl
{%
    \expandafter\xint_firstoftwo_andstop
    \romannumeral0\xintdecsplit
}%
\def\xintdecsplitr
{%
    \expandafter\xint_secondoftwo_andstop
    \romannumeral0\xintdecsplit
}%
\def\xintDecSplit {\romannumeral0\xintdecsplit }%
\def\xintdecsplit #1#2%
{%
    \expandafter \xint_split \expandafter
    {\romannumeral0\xintiabs {#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_Sgn {#1}
    \or \xint_afterfi { }{#2}}% positive big x
    \else
        \xint_afterfi { {#2}{}}% negative big x
    \fi
```

```
}%
\def\XINT_split_xfork #1%
{%
    \xint_UDzerominusfork
      #1-\dummy  \XINT_split_zerosplit
      0#1\dummy  \XINT_split_fromleft
       0-\dummy  {\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<8 \XINT_split_fromleft_exita\fi
    \expandafter\XINT_split_fromleft_loop_perhaps\expandafter
    {\the\numexpr #1-8\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 #1-8\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\XINT_rev {#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<8 \XINT_split_fromright_exita\fi
    \expandafter\XINT_split_fromright_loop_perhaps\expandafter
    {\the\numexpr #1-8\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 #1-8\expandafter
{%
    \csname XINT_split_fromright_endsplit_\romannumeral #1\endcsname
}%
\def\XINT_split_fromright_endsplit_ #1#2\W #3\Z #4%
{%
    \expandafter\space\expandafter {\romannumeral0\XINT_rev{#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}}%
```

30.67 \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-\dummy  \XINT_dbl_zero
      0#1\dummy  \XINT_dbl_neg
       0-\dummy {\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_andstop\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%
}%
\def\XINT_dbl_end_b #1#2#3#4#5#6#7#8%
{%
    \expandafter\space\the\numexpr #1#2#3#4#5#6#7#8\relax
}%
```

30.68 \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-\dummy  \XINT_half_zero
      0#1\dummy  \XINT_half_neg
       0-\dummy {\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%
}%
\def\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
{%
    \expandafter\space
    \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%
}%
\def\XINT_half_end_b 0\space 0#1#2#3#4#5#6#7%
{%
    \expandafter\space\the\numexpr #1#2#3#4#5#6#7\relax
}%
```

30.69 \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-\dummy  \XINT_dec_zero
      0#1\dummy  \XINT_dec_neg
       0-\dummy {\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_andstop\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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
    #1%
}%
\def\XINT_dec_cleanup #1#2#3#4#5#6#7#8%
{\expandafter\space\the\numexpr #1#2#3#4#5#6#7#8\relax }%
```

30.70 \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-\dummy  \XINT_inc_zero
      0#1\dummy  \XINT_inc_neg
       0-\dummy {\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}%
```

30.71 \xintiSqrt, \xintiSquareRoot

v1.08. 1.09a uses \xintnum

```
\def\XINT_dsx_addzerosnofuss #1{\XINT_dsx_zeroloop {#1}{}\Z }%
\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-\dummy  \XINT_sqrt_iszero
     0#1\dummy  \XINT_sqrt_isneg
      0-\dummy {\XINT_sqrt #1}%
    \krof
}%
\def\XINT_sqrt_iszero #1\Z { 0}%
\def\XINT_sqrt_isneg  #1\Z {\xintError:RootOfNegative\space 0}%
\def\XINT_sqrt #1\Z
{%
    \expandafter\XINT_sqrt_start\expandafter
    {\romannumeral0\XINT_length {#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%
    \ifcase #1
    \or 2\or 2\or 2\or 3\or 3\or 3\or 3\or 3\or %3+5
    4\or 4\or 4\or 4\or 4\or 4\or 4\or           %+7
    5\or 5\or 5\or 5\or 5\or 5\or 5\or 5\or 5\or %+9
    6\or 6\or 6\or 6\or 6\or 6\or 6\or 6\or 6\or 6\or 6\or %+11
    7\or 7\or 7\or 7\or 7\or 7\or 7\or 7\or 7\or 7\or 7\or 7\or 7\or %+13
    8\or 8\or 8\or 8\or 8\or 8\or 8\or
    8\or 8\or 8\or 8\or 8\or 8\or 8\or 8\or %+15
    9\or 9\or 9\or 9\or 9\or 9\or 9\or 9\or
    9\or 9\or 9\or 9\or 9\or 9\or 9\or 9\or 9\or %+17
    10\or 10\or 10\or 10\or 10\or 10\or 10\or 10\or 10\or
    10\or 10\or 10\or 10\or 10\or 10\or 10\or 10\or 10\or 10\or\fi %+19
}%
\def\XINT_sqrt_small_d #1\or #2\fi #3%
{%
```

```
   \fi
   \expandafter\XINT_sqrt_small_de
   \ifcase \numexpr #3/\xint_c_ii-\xint_c_i\relax
      {}%
   \or
      0%
   \or
      {00}%
   \or
      {000}%
   \or
      {0000}%
   \or
   \fi {#1}%
}%
\def\XINT_sqrt_small_de #1\or #2\fi #3%
{%
    \fi\XINT_sqrt_small_e {#3#1}%
}%
\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\or #2\fi #3%
{%
   \fi
   \ifodd #3
     \xint_afterfi{\expandafter\XINT_sqrt_big_eB}%
```

```
  \else
    \xint_afterfi{\expandafter\XINT_sqrt_big_eA}%
  \fi
  \expandafter{\the\numexpr #3/\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\xintidivision{#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%
{%
    \ifcase\XINT_Sgn {#1}
        \expandafter \XINT_sqrt_big_end
    \or \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}}%
\XINT_restorecatcodes_endinput%
```

31 Package xintbinhex implementation

The commenting is currently (2013/10/22) very sparse.

Contents

31.1 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}{Package xint is required}%
       \y{xintbinhex}{Will try \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}{Package xint is required}%
          \y{xintbinhex}{Will try \string\RequirePackage{xint}}%
          \def\z{\endgroup\RequirePackage{xint}}%
        \fi
    \else
      \y{xintbinhex}{I was already loaded, aborting input}%
      \aftergroup\endinput
    \fi
  \fi
\fi
\z%
```

31.2 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%
```

31.3 Catcodes

Perhaps catcodes have changed after the loading of **xint** and prior to the current loading of **xintbinhex**, so we redefine the \XINT_restorecatcodes_endinput in this style file.

```
\XINTsetupcatcodes%
```

31.4 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintbinhex}%
  [2013/10/22 v1.09d Expandable binary and hexadecimal conversions (jfB)]%
```

31.5 Constants, etc...

v1.08

```
\chardef\xint_c_xvi           16
\chardef\xint_c_ii^v          32
\chardef\xint_c_ii^vi         64
\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_tmp_def #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_tmp_def
    {{0}{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}{12}{13}{14}{15}}%
\def\XINT_tmp_def #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_tmp_def
    {{0}{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}{12}{13}{14}{15}}%
\let\XINT_tmp_def\empty
\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
}%
```

31.6 \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\dummy \XINT_dth_N
        -\dummy {\XINT_dth_P #1}%
     \krof
}%
\def\XINT_dth_N {\expandafter\xint_minus_andstop\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\dummy \XINT_dtb_N
        -\dummy {\XINT_dtb_P #1}%
     \krof
}%
\def\XINT_dtb_N {\expandafter\xint_minus_andstop\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
}%
```

```
\def\XINT_dtb_end_d #1#2#3#4#5#6#7#8#9%
{%
    \expandafter\space\the\numexpr #1#2#3#4#5#6#7#8#9\relax
}%
```

31.7 \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\dummy \XINT_htd_neg
        -\dummy {\XINT_htd_I {0000}#1}%
     \krof
}%
\def\XINT_htd_neg {\expandafter\xint_minus_andstop
                   \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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \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%
}%
\def\XINT_htd_II_end_b #1#2#3#4#5#6#7#8%
{%
    \expandafter\space\the\numexpr #1#2#3#4#5#6#7#8\relax
}%
```

31.8 \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\dummy \XINT_btd_neg
        -\dummy {\XINT_btd_I {000000}#1}%
     \krof
}%
\def\XINT_btd_neg {\expandafter\xint_minus_andstop
                               \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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
}%
\def\XINT_btd_II_c_end #1#2#3#4#5#6%
{%
    \expandafter\space\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%
}%
\def\XINT_btd_II_end_b #1#2#3#4#5#6#7#8#9%
{%
    \expandafter\space\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 }%
```

31.9 \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\dummy  \XINT_bth_N
        -\dummy {\XINT_bth_P #1}%
     \krof
}%
\def\XINT_bth_N {\expandafter\xint_minus_andstop\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{ }%
```

31.10 \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\dummy  \XINT_htb_N
        -\dummy {\XINT_htb_P #1}%
     \krof
}%
\def\XINT_htb_N {\expandafter\xint_minus_andstop\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
}%
```

31.11 \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\dummy  \XINT_chtb_N
        -\dummy {\XINT_chtb_P #1}%
     \krof
}%
\def\XINT_chtb_N {\expandafter\xint_minus_andstop\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%
```

32 Package xintgcd implementation

The commenting is currently (2013/10/22) very sparse.

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@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}{Package xint is required}%
         \y{xintgcd}{Will try \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}{Package xint is required}%
            \y{xintgcd}{Will try \string\RequirePackage{xint}}%
            \def\z{\endgroup\RequirePackage{xint}}%
          \fi
      \else
        \y{xintgcd}{I was already loaded, aborting input}%
        \aftergroup\endinput
      \fi
    \fi
  \fi
\z%
```

32.2 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%
```

32.3 Catcodes

```
\XINTsetupcatcodes%
```

32.4 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintgcd}%
  [2013/10/22 v1.09d Euclide algorithm with xint package (jfB)]%
```

32.5 \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\dummy \XINT_gcd_BisZero
      #3\dummy \XINT_gcd_AisZero
       0\dummy \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}%
}%
```

32.6 \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}%
```

32.7 \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_gcdof:_e
                        \else\expandafter\XINT_gcdof:_d\fi #1}%
\def\XINT_gcdof:_d #1,{\expandafter\XINT_gcdof:_b\romannumeral0\xintgcd {#1}}%
\def\XINT_gcdof:_e ,#1,{#1}%
```

32.8 \xintLCM

New with 1.09a

```
\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\dummy \XINT_lcm_BisZero
      #3\dummy \XINT_lcm_AisZero
       0\dummy \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}{\xintQuo{#3}{#1}}}%
```

32.9 \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}%
```

32.10 \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_lcmof:_e
                     \else\expandafter\XINT_lcmof:_d\fi #1}%
\def\XINT_lcmof:_d #1,{\expandafter\XINT_lcmof:_a\romannumeral0\xintlcm {#1}}%
\def\XINT_lcmof:_e ,#1,{#1}%
```

32.11 \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\dummy \XINT_bezout_botharezero
```

```
      #10\dummy \XINT_bezout_secondiszero
      #30\dummy \XINT_bezout_firstiszero
       00\dummy
        {\xint_UDsignsfork
          #1#3\dummy \XINT_bezout_minusminus % A < 0, B < 0
           #1-\dummy \XINT_bezout_minusplus  % A > 0, B < 0
           #3-\dummy \XINT_bezout_plusminus  % A < 0, B > 0
            --\dummy \XINT_bezout_plusplus   % A > 0, B > 0
         \krof }%
    \krof
    {#2}{#4}#1#3{#3#4}{#1#2}% #1#2=B, #3#4=A
}%
\def\XINT_bezout_botharezero #1#2#3#4#5#6%
{%
    \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\dummy { {0}{#3#1}{0}{1}{#1}}%
       -\dummy { {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\dummy{ {#4#2}{0}{-1}{0}{#2}}%
        -\dummy{ {#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}%
}%
\def\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}%
}%
\def\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\xintiopp{#1}}%
}%
\def\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.

```
\def\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
}%
\def\XINT_bezout_exiteven #1#2#3#4#5%
{%
    \space {#5}{#4}{#1}%
}%
\def\XINT_bezout_exitodd #1#2#3#4#5%
{%
    \space {-#5}{-#4}{#1}%
}%
```

32.12 \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\dummy \XINT_euc_BisZero
      #3\dummy \XINT_euc_AisZero
       0\dummy \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_undef\xint_undef\xint_undef\xint_undef
      \xint_undef\xint_undef\xint_undef\xint_undef
    \xint_relax
}%
\def\xint_euc_end_ #1#2#3%
{%
    \space {#1}{#3}{#2}%
}%
```

32.13 \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\dummy \XINT_bezalg_BisZero
      #3\dummy \XINT_bezalg_AisZero
       0\dummy \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_undef\xint_undef\xint_undef\xint_undef
      \xint_undef\xint_undef\xint_undef\xint_undef
    \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}
```

```
\def\xint_bezalg_end_ #1#2#3#4%
{%
    \space {#1}{#3}{0}{1}{#2}{#4}{1}{0}%
}%
```

32.14 \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)
```

```
\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}}%
    \noindent
    \count 255 1
    \loop
      \hbox to \wd 0 {\hfil$\U{\numexpr 2*\count 255\relax}$}%
      ${} =  \U{\numexpr 2*\count 255 + 3\relax}
      \times \U{\numexpr 2*\count 255 + 2\relax}
          +  \U{\numexpr 2*\count 255 + 4\relax}$%
    \ifnum \count 255 < \N
      \hfill\break
      \advance \count 255 1
    \repeat
  \par
  \endgroup
}%
```

32.15 \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
```

```
\def\xintTypesetBezoutAlgorithm #1#2%
{%
  \par
  \begingroup
    \parindent0pt
    \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}}%
    \count 255 1
    \loop
      \noindent
```

```
      \hbox to \wd 0 {\hfil$\BEZ{4*\count 255 - 2}$}%
      ${} =  \BEZ{4*\count 255 + 5}
      \times \BEZ{4*\count 255 + 2}
          +  \BEZ{4*\count 255 + 6}$\hfill\break
      \hbox to \wd 0 {\hfil$\BEZ{4*\count 255 +7}$}%
      ${} = \BEZ{4*\count 255 + 5}
      \times \BEZ{4*\count 255 + 3}
          +  \BEZ{4*\count 255 - 1}$\hfill\break
      \hbox to \wd 0 {\hfil$\BEZ{4*\count 255 +8}$}%
      ${} =  \BEZ{4*\count 255 + 5}
      \times \BEZ{4*\count 255 + 4}
          +  \BEZ{4*\count 255 }$
      \endgraf
    \ifnum \count 255 < \N
    \advance \count 255 1
  \repeat
  \par
    \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
}%
\XINT_restorecatcodes_endinput%
```

33 Package xintfrac implementation

The commenting is currently (2013/10/22) 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@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}{Package xint is required}%
         \y{xintfrac}{Will try \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}{Package xint is required}%
            \y{xintfrac}{Will try \string\RequirePackage{xint}}%
            \def\z{\endgroup\RequirePackage{xint}}%
          \fi
      \else
        \y{xintfrac}{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{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%
```

33.3 Catcodes

```
\XINTsetupcatcodes%
```

33.4 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintfrac}%
  [2013/10/22 v1.09d Expandable operations on fractions (jfB)]%
\chardef\xint_c_vi     6
\chardef\xint_c_vii    7
\chardef\xint_c_xviii 18
\mathchardef\xint_c_x^iv 10000
```

33.5 \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
}%
```

33.6 \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}%
}%
```

33.7 \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)

```
\def\XINT_outfrac #1#2#3%
{%
    \ifcase\XINT_Sgn{#3}
        \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}%
\def\XINT_outfrac_P #1#2%
{%
    \ifcase\XINT_Sgn{#1}
        \expandafter\XINT_outfrac_Zero
    \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}%
}%
```

33.8 \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\dummy \XINT_infrac_A
      \W\dummy \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\dummy \XINT_infrac_BCa
     \W\dummy {\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}}%
```

33.9 \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\dummy \XINT_frac_A
     \W\dummy {\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\dummy  \XINT_frac_Ua
      \W\dummy {\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\dummy  \XINT_frac_Ba
      \W\dummy {\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\XINT_length {#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\dummy  \XINT_frac_Ta
      \W\dummy {\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\dummy \XINT_frac_Ca
      \W\dummy {\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\XINT_length {#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-\dummy  \XINT_frac_Gdivisionbyzero
      0#1\dummy  \XINT_frac_Gneg
       0-\dummy  {\XINT_frac_Gpos #1}%
    \krof
}%
\def\XINT_frac_Gdivisionbyzero #1\Z #2#3%
{%
   \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}}%
```

33.10 \XINT_factortens, \XINT_cuz_cnt

```
\def\XINT_factortens #1%
{%
    \expandafter\XINT_cuz_cnt_loop\expandafter
    {\expandafter}\romannumeral0\XINT_rord_main {}#1%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax }{#1}%
}%
```

33.11 \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]}%
```

33.12 \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}%
}%
```

33.13 \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_Sgn {#1}
      \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}%
```

33.14 \xintFloor

1.09a

```
\def\xintFloor {\romannumeral0\xintfloor }%
\def\xintfloor #1{\expandafter\XINT_floor
                  \romannumeral0\xintrawwithzeros {#1}.}%
\def\XINT_floor #1/#2.{\xintiquo {#1}{#2}}%
```

33.15 \xintCeil

1.09a

```
\def\xintCeil {\romannumeral0\xintceil }%
\def\xintceil #1{\xintiiopp {\xintFloor {\xintOpp{#1}}}}%
```

33.16 \xintNumerator

```
\def\xintNumerator {\romannumeral0\xintnumerator }%
\def\xintnumerator
{%
    \expandafter\XINT_numer\romannumeral0\XINT_infrac
}%
\def\XINT_numer #1%
{%
    \ifcase\XINT_Sgn {#1}
      \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}%
```

33.17 \xintDenominator

```
\def\xintDenominator {\romannumeral0\xintdenominator }%
\def\xintdenominator
{%
    \expandafter\XINT_denom\romannumeral0\XINT_infrac
}%
\def\XINT_denom #1%
{%
    \ifcase\XINT_Sgn {#1}
      \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}}%
```

33.18 \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 #1#2#3#4#5%
{%
    \ifcase\XINT_isOne {#5}
    \or \xint_afterfi {\expandafter\xint_firstoftwo_andstop\xint_gobble_ii }%
    \fi
    \space
    \frac {#4}{#5}%
}%
\def\XINT_fracfrac_D #1#2#3%
{%
    \ifcase\XINT_isOne {#3}
    \or \XINT_fracfrac_E
    \fi
    \space
    \frac {#2}{#3}#1%
```

```
}%
\def\XINT_fracfrac_E \fi #1#2#3#4{\fi \space #3\cdot }%
```

33.19 \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\dummy \XINT_sgnfrac_N
       -\dummy {\XINT_sgnfrac_P #1}%
    \krof
}%
\def\XINT_sgnfrac_P #1\Z #2%
{%
    \XINT_fracfrac_A {#2}{#1}%
}%
\def\XINT_sgnfrac_N
{%
    \expandafter\xint_minus_andstop\romannumeral0\XINT_sgnfrac_P
}%
```

33.20 \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%
{%
    \ifcase\XINT_isOne {#5}
       \xint_afterfi { {#4\over #5}}%
    \or
       \xint_afterfi { #4}%
    \fi
}%
```

```
\def\XINT_fwover_D #1#2#3%
{%
    \ifcase\XINT_isOne {#3}
      \xint_afterfi { {#2\over #3}}%
    \or
      \xint_afterfi { #2\cdot }%
    \fi
    #1%
}%
```

33.21 \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\dummy \XINT_sgnfwover_N
       -\dummy {\XINT_sgnfwover_P #1}%
    \krof
}%
\def\XINT_sgnfwover_P #1\Z #2%
{%
    \XINT_fwover_A {#2}{#1}%
}%
\def\XINT_sgnfwover_N
{%
    \expandafter\xint_minus_andstop\romannumeral0\XINT_sgnfwover_P
}%
```

33.22 \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-\dummy \XINT_rez_zero
      0#1\dummy \XINT_rez_neg
       0-\dummy {\XINT_rez_B #1}%
    \krof
}%
\def\XINT_rez_zero #1\Z #2#3{ 0/1[0]}%
\def\XINT_rez_neg {\expandafter\xint_minus_andstop\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]}%
```

33.23 \xintE

added with with 1.07, together with support for 'floats'. The fraction comes first here, contrarily to \xintTrunc and \xintRound.

```
\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\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\XINT_e_end #1#2#3{ #2/#3[#1]}%
\let\XINTinFloatfE\xintfE
```

33.24 \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
{%
    \ifcase\XINT_isOne {#3}
      \xint_afterfi
          {\xint_UDsignfork
               #1\dummy \XINT_irr_negative
                -\dummy {\XINT_irr_nonneg #1}%
           \krof}%
    \or
      \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_andstop}%
\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\dummy \XINT_irr_indeterminate
       #30\dummy  \XINT_irr_divisionbyzero
       #10\dummy  \XINT_irr_zero
        00\dummy  \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\xintiquo {#3}{#2}}%
    {\romannumeral0\xintiquo {#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
```

33.25 \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 }%
\def\XINT_intcheck #1/#2\Z
{%
    \ifcase\XINT_isOne {#2}
      \xintError:NotAnInteger
    \fi\space #1%
}%
```

33.26 \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
{%
    \ifcase\XINT_isOne {#3}
      \xint_afterfi
          {\xint_UDsignfork
```

```
              #1\dummy \XINT_jrr_negative
               -\dummy {\XINT_jrr_nonneg #1}%
          \krof}%
    \or
      \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_andstop }%
\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\dummy \XINT_jrr_indeterminate
       #30\dummy  \XINT_jrr_divisionbyzero
       #10\dummy  \XINT_jrr_zero
        00\dummy  \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}%
}%
```

33.27 \xintTrunc, \xintiTrunc

Modified in 1.06 to give the first argument to a \numexpr.

```
\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_iszero #1#2#3#4#5{ 0\Z 0}%
\def\XINT_trunc_B #1%
{%
    \ifcase\XINT_Sgn {#1}
      \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_E
    \romannumeral0\xint_dsh {#3}{#1}\Z #2\Z
}%
\def\XINT_trunc_D #1#2%
{%
    \expandafter \XINT_trunc_DE \expandafter
    {\romannumeral0\xint_dsh {#2}{-#1}}%
}%
\def\XINT_trunc_DE #1#2{\XINT_trunc_E #2\Z #1\Z }%
\def\XINT_trunc_E #1#2\Z #3#4\Z
{%
    \xint_UDsignsfork
          #1#3\dummy \XINT_trunc_minusminus
           #1-\dummy {\XINT_trunc_minusplus #3}%
           #3-\dummy {\XINT_trunc_plusminus #1}%
            --\dummy {\XINT_trunc_plusplus  #3#1}%
    \krof
    {#4}{#2}%
}%
\def\XINT_trunc_minusminus #1#2{\xintiquo {#1}{#2}\Z \space}%
\def\XINT_trunc_minusplus #1#2#3{\xintiquo {#1#2}{#3}\Z \xint_minus_andstop}%
\def\XINT_trunc_plusminus #1#2#3{\xintiquo {#2}{#1#3}\Z \xint_minus_andstop}%
\def\XINT_trunc_plusplus  #1#2#3#4{\xintiquo {#1#3}{#2#4}\Z \space}%
\def\XINT_itrunc_G #1#2\Z #3#4%
{%
    \xint_gob_til_zero #1\XINT_trunc_zero 0\xint_firstoftwo {#3#1#2}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\XINT_length {#1}-#3}{#3}{#1}#2%
}%
\def\XINT_trunc_zero 0#10{ 0}%
\def\XINT_trunc_H #1#2%
{%
    \ifnum #1 > 0
        \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 possible!
}%
```

33.28 \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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \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
}%
```

33.29 \xintRound:csv

1.09a. For use by \xintthenumexpr.

```
\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_round:_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}}%
\def\XINT_round:_f ,#1#2^{\xint_gobble_i #1}%
```

33.30 \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 }%
```

33.31 \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. Macro \xintFloat:csv added in 1.09 for use by xintexpr.

```
\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-\dummy  \XINT_float_zero
     0#1\dummy  \XINT_float_J
      0-\dummy  {\XINT_float_K #1}%
    \krof
}%
\def\XINT_float_zero #1\Z #2#3#4#5{ 0.e0}%
```

```
\def\XINT_float_J {\expandafter\xint_minus_andstop\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\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\XINT_length{#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\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\xintiquo {#2}%
         {\XINT_dsx_addzerosnofuss {-#1}{#3}}\Z {#1}%
}%
\def\XINT_float_Rii #1#2#3%
{%
    \expandafter\XINT_float_Sa
    \romannumeral0\xintiquo
         {\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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \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}%
```

33.32 \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_float:_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}}%
\def\XINT_float:_f ,#1#2^{\xint_gobble_i #1}%
```

33.33 \XINT_inFloat

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.

For convenience in xintexpr.sty (special r\^ole of the underscore in \xintNewExpr) 1.09a adds \XINTinFloat. I also decide in 1.09a not to use anymore \romannumeral`-0 mais \romannumeral0 in the float routines, for consistency of style.

```
\def\XINTinFloat {\romannumeral0\XINT_inFloat }%
\def\XINT_inFloat [#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-\dummy  \XINT_infloat_zero
     0#1\dummy  \XINT_infloat_J
      0-\dummy  {\XINT_float_K #1}%
    \krof
}%
\def\XINT_infloat_zero #1\Z #2#3#4#5{ 0[0]}%
\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\xintiquo {#2}%
         {\XINT_dsx_addzerosnofuss {-#1}{#3}}}{#1}%
}%
\def\XINT_infloat_Rii #1#2#3%
{%
    \expandafter\XINT_infloat_S\expandafter
    {\romannumeral0\xintiquo
         {\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_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \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]}%
```

33.34 \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}}%
```

33.35 \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}}%
```

33.36 \xintSum, \xintSumExpr

```
\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}%
```

33.37 \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_sum:_e
                        \else\expandafter\XINT_sum:_d\fi #1}%
\def\XINT_sum:_d #1,#2{\expandafter\XINT_sum:_b\expandafter
                          {\romannumeral0\xintadd {#2}{#1}}}%
\def\XINT_sum:_e  ,#1#2^{#1}% allows empty list
```

33.38 \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}}%
```

33.39 \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}%
```

33.40 \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. The \xintexpr can thus use directly \xintPow.

```
\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-\dummy  \XINT_fpow_zero
      0#1\dummy  \XINT_fpow_neg
       0-\dummy  {\XINT_fpow_pos #1}%
    \krof
    {#2}%
}%
\def\XINT_fpow_zero #1#2#3#4%
{%
    \space 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\xintipow {#5}{#1#2}}%
    {\romannumeral0\xintipow {#4}{#1#2}}%
}%
\def\XINT_fpow_neg #1#2#3#4%
{%
    \expandafter\XINT_fpow_pos_A\expandafter
    {\the\numexpr -#1*#2\expandafter}\expandafter
    {\romannumeral0\xintipow {#3}{#1}}%
    {\romannumeral0\xintipow {#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}}%
```

33.41 \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}}%
}%
```

33.42 \xintPrd, \xintPrdExpr

```
\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}%
```

33.43 \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_prd:_e
                      \else\expandafter\XINT_prd:_d\fi #1}%
\def\XINT_prd:_d #1,#2{\expandafter\XINT_prd:_b\expandafter
                        {\romannumeral0\xintmul {#2}{#1}}}%
\def\XINT_prd:_e  ,#1#2^{#1}% allows empty list
```

33.44 \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}}%
```

33.45 \xintIsOne

New with 1.09a. Could be more efficient. For fractions with big powers of tens, it is better to use \xintCmp{f}{1}.

```
\def\xintIsOne {\romannumeral0\xintisone }%
\def\xintisone #1{\expandafter\XINT_fracisone
                  \romannumeral0\xintrawwithzeros{#1}\Z }%
\def\XINT_fracisone #1/#2\Z{\xintsgnfork{\XINT_Cmp {#1}{#2}}{0}{1}{0}}%
```

33.46 \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%
{%
    \xintSgnFork
    {\xintiSgn{\the\numexpr #2+\xintLength{#3}-\xintLength{#1}\relax}}%
    { 0}{\XINT_fgeq_E #2\Z {#3}{#1}}{ 1}%
}%
\def\XINT_fgeq_E #1%
{%
    \xint_UDsignfork
        #1\dummy  \XINT_fgeq_Fd
         -\dummy {\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}%
}%
```

33.47 `\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\dummy \XINT_fmax_minusminus
       -#5\dummy \XINT_fmax_firstneg
       #1-\dummy \XINT_fmax_secondneg
        --\dummy \XINT_fmax_nonneg_a
    \krof
    #1#5{#2/#3[#4]}{#6/#7[#8]}%
}%
\def\XINT_fmax_minusminus --%
   {\expandafter\xint_minus_andstop\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%
{%
    \ifcase\romannumeral0\XINT_fgeq_A #1#2
        \xint_afterfi{ #1}%
    \or \xint_afterfi{ #2}%
    \fi
}%
```

33.48 \xintMaxof

\xintMaxof:csv is for private use in \xintexpr. Even with only one argument, there does not seem to be really a motive for using \xintraw.

```
\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}%
```

33.49 \xintMaxof:csv

1.09a. For use by \xintexpr.

```
\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_maxof:_e
                       \else\expandafter\XINT_maxof:_d\fi #1}%
\def\XINT_maxof:_d #1,{\expandafter\XINT_maxof:_b\romannumeral0\xintmax {#1}}%
\def\XINT_maxof:_e ,#1,{#1}%
```

33.50 \xintFloatMaxof

1.09a, for use by \xintNewFloatExpr.

```
\def\xintFloatMaxof      {\romannumeral0\xintflmaxof }%
\def\xintflmaxof    #1{\expandafter\XINT_flmaxof_a\romannumeral-`0#1\relax }%
\def\XINT_flmaxof_a #1{\expandafter\XINT_flmaxof_b
                       \romannumeral0\XINT_inFloat [\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}%
```

33.51 \xintFloatMaxof:csv

1.09a. For use by \xintfloatexpr.

```
\def\xintFloatMaxof:csv #1{\expandafter\XINT_flmaxof:_a\romannumeral-`0#1,,}%
\def\XINT_flmaxof:_a #1,{\expandafter\XINT_flmaxof:_b
                         \romannumeral0\XINT_inFloat [\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_flmaxof:_e
                       \else\expandafter\XINT_flmaxof:_d\fi #1}%
\def\XINT_flmaxof:_d #1,%
      {\expandafter\XINT_flmaxof:_b\romannumeral0\xintmax
                   {\XINTinFloat [\XINTdigits]{#1}}}%
\def\XINT_flmaxof:_e ,#1,{#1}%
```

33.52 \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\dummy \XINT_fmin_minusminus
       -#5\dummy \XINT_fmin_firstneg
       #1-\dummy \XINT_fmin_secondneg
        --\dummy \XINT_fmin_nonneg_a
    \krof
    #1#5{#2/#3[#4]}{#6/#7[#8]}%
}%
\def\XINT_fmin_minusminus --%
   {\expandafter\xint_minus_andstop\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%
{%
    \ifcase\romannumeral0\XINT_fgeq_A #1#2
        \xint_afterfi{ #2}%
    \or \xint_afterfi{ #1}%
    \fi
}%
```

33.53 `\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}%
```

33.54 `\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_minof:_e
                       \else\expandafter\XINT_minof:_d\fi #1}%
```

```
\def\XINT_minof:_d #1,{\expandafter\XINT_minof:_b\romannumeral0\xintmin {#1}}%
\def\XINT_minof:_e ,#1,{#1}%
```

33.55 \xintFloatMinof

1.09a, for use by \xintNewFloatExpr.

```
\def\xintFloatMinof      {\romannumeral0\xintflminof }%
\def\xintflminof    #1{\expandafter\XINT_flminof_a\romannumeral-`0#1\relax }%
\def\XINT_flminof_a #1{\expandafter\XINT_flminof_b
                       \romannumeral0\XINT_inFloat [\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}%
```

33.56 \xintFloatMinof:csv

1.09a. For use by \xintfloatexpr.

```
\def\xintFloatMinof:csv #1{\expandafter\XINT_flminof:_a\romannumeral-`0#1,,}%
\def\XINT_flminof:_a #1,{\expandafter\XINT_flminof:_b
                         \romannumeral0\XINT_inFloat [\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_flminof:_e
                       \else\expandafter\XINT_flminof:_d\fi #1}%
\def\XINT_flminof:_d #1,%
      {\expandafter\XINT_flminof:_b\romannumeral0\xintmin
                   {\XINTinFloat [\XINTdigits]{#1}}}%
\def\XINT_flminof:_e ,#1,{#1}%
```

33.57 \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\dummy \XINT_fcmp_minusminus
       -#5\dummy \XINT_fcmp_firstneg
       #1-\dummy \XINT_fcmp_secondneg
        --\dummy \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\dummy \XINT_fcmp_zerozero
       0#2\dummy \XINT_fcmp_firstzero
       #10\dummy \XINT_fcmp_secondzero
        00\dummy \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%
{%
    \xintSgnFork
    {\xintiSgn{\the\numexpr #2+\xintLength{#3}-\xintLength{#1}\relax}}%
```

```
    { -1}{\XINT_fcmp_E #2\Z {#3}{#1}}{ 1}%
}%
\def\XINT_fcmp_E #1%
{%
    \xint_UDsignfork
        #1\dummy  \XINT_fcmp_Fd
         -\dummy {\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}%
}%
```

33.58 \xintAbs

```
\def\xintAbs {\romannumeral0\xintabs }%
\def\xintabs #1%
{%
    \expandafter\xint_fabs\romannumeral0\XINT_infrac {#1}%
}%
\def\xint_fabs #1#2%
{%
    \expandafter\XINT_outfrac\expandafter
    {\the\numexpr #1\expandafter}\expandafter
    {\romannumeral0\XINT_abs #2}%
}%
```

33.59 \xintOpp

```
\def\xintOpp {\romannumeral0\xintopp }%
\def\xintopp #1%
{%
    \expandafter\xint_fopp\romannumeral0\XINT_infrac {#1}%
}%
\def\xint_fopp #1#2%
{%
    \expandafter\XINT_outfrac\expandafter
    {\the\numexpr #1\expandafter}\expandafter
    {\romannumeral0\XINT_opp #2}%
}%
```

33.60 \xintSgn

```
\def\xintSgn {\romannumeral0\xintsgn }%
\def\xintsgn #1%
{%
    \expandafter\xint_fsgn\romannumeral0\XINT_infrac {#1}%
}%
\def\xint_fsgn #1#2#3{\xintisgn {#2}}%
```

33.61 \xintDivision, \xintQuo, \xintRem

```
\def\xintDivision {\romannumeral0\xintdivision }%
\def\xintdivision #1%
{%
    \expandafter\xint_xdivision\expandafter{\romannumeral0\xintnum {#1}}%
}%
\def\xint_xdivision #1#2%
{%
    \expandafter\XINT_div_fork\romannumeral0\xintnum {#2}\Z #1\Z
}%
\def\xintQuo {\romannumeral0\xintquo }%
\def\xintRem {\romannumeral0\xintrem }%
\def\xintquo {\expandafter\xint_firstoftwo_andstop
              \romannumeral0\xintdivision }%
\def\xintrem {\expandafter\xint_secondoftwo_andstop
              \romannumeral0\xintdivision }%
```

33.62 \xintFDg, \xintLDg, \xintMON, \xintMMON, \xintOdd

```
\def\xintFDg {\romannumeral0\xintfdg }%
\def\xintfdg #1%
{%
    \expandafter\XINT_fdg\romannumeral0\xintnum {#1}\W\Z
}%
\def\xintLDg {\romannumeral0\xintldg }%
\def\xintldg #1%
{%
    \expandafter\XINT_ldg\expandafter{\romannumeral0\xintnum {#1}}%
}%
\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
}%
\def\xintOdd {\romannumeral0\xintodd }%
\def\xintodd #1%
{%
    \ifodd\xintLDg{#1}
        \xint_afterfi{ 1}%
    \else
        \xint_afterfi{ 0}%
    \fi
}%
```

33.63 \xintFloatAdd

1.07

```
\def\xintFloatAdd      {\romannumeral0\xintfloatadd }%
\def\xintfloatadd     #1{\XINT_fladd_chkopt \xintfloat #1\Z }%
\def\XINTinFloatAdd   {\romannumeral0\XINTinfloatadd }%
\def\XINTinfloatadd #1{\XINT_fladd_chkopt \XINT_inFloat #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+2}{#2}{#3}}%
}%
\def\XINT_fladd_opt #1[\Z #2]#3#4%
{%
    #1[#2]{\XINT_FL_Add {#2+2}{#3}{#4}}%
}%
\def\XINT_FL_Add #1#2%
{%
    \expandafter\XINT_FL_Add_a\expandafter{\the\numexpr #1\expandafter}%
    \expandafter{\romannumeral0\XINT_inFloat [#1]{#2}}%
}%
\def\XINT_FL_Add_a #1#2#3%
{%
    \expandafter\XINT_FL_Add_b\romannumeral0\XINT_inFloat [#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%
{%
    \xintSgnFork {\ifnum \numexpr #2-#4-#5>1 \expandafter 1%
                  \else\ifnum \numexpr #4-#2-#5>1
                       \xint_afterfi {\expandafter-\expandafter1}%
                       \else \expandafter\expandafter\expandafter0%
                       \fi
                  \fi}%
    {#3[#4]}{\xintAdd {#1[#2]}{#3[#4]}}{#1[#2]}%
}%
\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]}%
```

33.64 `\xintFloatSub`

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 \XINT_inFloat #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+2}{#2}{\xintOpp{#3}}}%
}%
\def\XINT_flsub_opt #1[\Z #2]#3#4%
{%
    #1[#2]{\XINT_FL_Add {#2+2}{#3}{\xintOpp{#4}}}%
}%
```

33.65 `\xintFloatMul`

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 \XINT_inFloat #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+2}{#2}{#3}}%
}%
\def\XINT_flmul_opt #1[\Z #2]#3#4%
{%
    #1[#2]{\XINT_FL_Mul {#2+2}{#3}{#4}}%
}%
\def\XINT_FL_Mul #1#2%
{%
    \expandafter\XINT_FL_Mul_a\expandafter{\the\numexpr #1\expandafter}%
    \expandafter{\romannumeral0\XINT_inFloat [#1]{#2}}%
}%
\def\XINT_FL_Mul_a #1#2#3%
{%
    \expandafter\XINT_FL_Mul_b\romannumeral0\XINT_inFloat [#1]{#3}#2%
}%
\def\XINT_FL_Mul_b #1[#2]#3[#4]{\xintE{\xintiiMul {#1}{#3}}{#2+#4}}%
```

33.66 \xintFloatDiv

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 \XINT_inFloat #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+2}{#2}{#3}}%
}%
\def\XINT_fldiv_opt #1[\Z #2]#3#4%
{%
    #1[#2]{\XINT_FL_Div {#2+2}{#3}{#4}}%
}%
\def\XINT_FL_Div #1#2%
{%
```

```
    \expandafter\XINT_FL_Div_a\expandafter{\the\numexpr #1\expandafter}%
    \expandafter{\romannumeral0\XINT_inFloat [#1]{#2}}%
}%
\def\XINT_FL_Div_a #1#2#3%
{%
    \expandafter\XINT_FL_Div_b\romannumeral0\XINT_inFloat [#1]{#3}#2%
}%
\def\XINT_FL_Div_b #1[#2]#3[#4]{\xintE{#3/#1}{#4-#2}}%
```

33.67 \xintFloatSum

1.09a: quick write-up, for use by \xintfloatexpr, will need to be thought through again.

```
\def\xintFloatSum      {\romannumeral0\xintfloatsum }%
\def\xintfloatsum    #1{\expandafter\XINT_floatsum_a\romannumeral-`0#1\relax }%
\def\XINT_floatsum_a #1{\expandafter\XINT_floatsum_b
                        \romannumeral0\xintraw{#1}\Z }% normalizes if only 1
\def\XINT_floatsum_b #1\Z #2%            but a bit wasteful
           {\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}%
```

33.68 \xintFloatSum:csv

1.09a. For use by \xintfloatexpr.

```
\def\xintFloatSum:csv #1{\expandafter\XINT_floatsum:_a\romannumeral-`0#1,,^}%
\def\XINT_floatsum:_a {\XINT_floatsum:_b {0/1[0]}}%
\def\XINT_floatsum:_b #1#2,%
             {\expandafter\XINT_floatsum:_c\romannumeral-`0#2,{#1}}%
\def\XINT_floatsum:_c #1{\if #1,\expandafter\XINT_floatsum:_e
                      \else\expandafter\XINT_floatsum:_d\fi #1}%
\def\XINT_floatsum:_d #1,#2{\expandafter\XINT_floatsum:_b\expandafter
                        {\romannumeral0\XINTinfloatadd {#2}{#1}}}%
\def\XINT_floatsum:_e  ,#1#2^{#1}% allows empty list
```

33.69 \xintFloatPrd

1.09a: quick write-up, for use by \xintfloatexpr, will need to be thought through again.

```
\def\xintFloatPrd      {\romannumeral0\xintfloatprd }%
\def\xintfloatprd    #1{\expandafter\XINT_floatprd_a\romannumeral-`0#1\relax }%
\def\XINT_floatprd_a #1{\expandafter\XINT_floatprd_b
                        \romannumeral0\xintraw{#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}%
```

33.70 \xintFloatPrd:csv

1.09a. For use by \xintfloatexpr.

```
\def\xintFloatPrd:csv #1{\expandafter\XINT_floatprd:_a\romannumeral-`0#1,,^}%
\def\XINT_floatprd:_a {\XINT_floatprd:_b {1/1[0]}}%
\def\XINT_floatprd:_b #1#2,%
                {\expandafter\XINT_floatprd:_c\romannumeral-`0#2,{#1}}%
\def\XINT_floatprd:_c #1{\if #1,\expandafter\XINT_floatprd:_e
                         \else\expandafter\XINT_floatprd:_d\fi #1}%
\def\XINT_floatprd:_d #1,#2{\expandafter\XINT_floatprd:_b\expandafter
                           {\romannumeral0\XINTinfloatmul {#2}{#1}}}%
\def\XINT_floatprd:_e  ,#1#2^{#1}% allows empty list
```

33.71 \xintFloatPow

1.07

```
\def\xintFloatPow {\romannumeral0\xintfloatpow}%
\def\xintfloatpow #1{\XINT_flpow_chkopt \xintfloat #1\Z }%
\def\XINTinFloatPow {\romannumeral0\XINTinfloatpow }%
\def\XINTinfloatpow #1{\XINT_flpow_chkopt \XINT_inFloat #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-\dummy  \XINT_flpow_BisZero
      0#1\dummy  {\XINT_flpow_checkB_b 1}%
       0-\dummy  {\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\XINT_length{#2}}{#3}{#2}#1%
}%
\def\XINT_flpow_checkB_c #1#2%
{%
    \expandafter\XINT_flpow_checkB_d \expandafter
    {\the\numexpr \expandafter\XINT_Length\expandafter
                  {\the\numexpr #1*20/3}+#1+#2+1}%
}%
\def\XINT_flpow_checkB_d #1#2#3#4%
{%
    \expandafter \XINT_flpow_a
    \romannumeral0\XINT_inFloat [#1]{#4}{#1}{#2}#3%
}%
\def\XINT_flpow_a #1%
{%
    \xint_UDzerominusfork
      #1-\dummy \XINT_flpow_zero
      0#1\dummy {\XINT_flpow_b 1}%
       0-\dummy {\XINT_flpow_b 0#1}%
    \krof
}%
\def\XINT_flpow_zero [#1]#2#3#4#5%
{%
    \if #41 \xint_afterfi {\xintError:DivisionByZero\space 1.e2147483647}%
    \else \xint_afterfi { 0.e0}\fi
}%
\def\XINT_flpow_b #1#2[#3]#4#5%
{%
    \XINT_flpow_c {#4}{#5}{#2[#3]}{#1*\ifodd #5 1\else 0\fi}%
}%
\def\XINT_flpow_c #1#2#3#4%
{%
    \XINT_flpow_loop {#1}{#2}{#3}{{#1}}\XINT_flpow_prd
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax {#4}%
```

```
}%
\def\XINT_flpow_loop #1#2#3%
{%
    \ifnum #2 = 1
        \expandafter\XINT_flpow_loop_end
    \else
        \xint_afterfi{\expandafter\XINT_flpow_loop_a
            \expandafter{\the\numexpr 2*(#2/2)-#2\expandafter }% b mod 2
            \expandafter{\the\numexpr #2-#2/2\expandafter }%     [b/2]
            \expandafter{\romannumeral0\XINTinfloatmul [#1]{#3}{#3}}}%
    \fi
    {#1}{{#3}}%
}%
\def\XINT_flpow_loop_a #1#2#3#4%
{%
    \ifnum #1 = 1
        \expandafter\XINT_flpow_loop
    \else
        \expandafter\XINT_flpow_loop_throwaway
    \fi
    {#4}{#2}{#3}%
}%
\def\XINT_flpow_loop_throwaway #1#2#3#4%
{%
   \XINT_flpow_loop {#1}{#2}{#3}%
}%
\def\XINT_flpow_loop_end #1{\romannumeral0\XINT_rord_main {}\relax }%
\def\XINT_flpow_prd #1#2%
{%
    \XINT_flpow_prd_getnext {#2}{#1}%
}%
\def\XINT_flpow_prd_getnext #1#2#3%
{%
    \XINT_flpow_prd_checkiffinished #3\Z {#1}{#2}%
}%
\def\XINT_flpow_prd_checkiffinished #1%
{%
    \xint_gob_til_relax #1\XINT_flpow_prd_end\relax
    \XINT_flpow_prd_compute #1%
}%
\def\XINT_flpow_prd_compute #1\Z #2#3%
{%
    \expandafter\XINT_flpow_prd_getnext\expandafter
    {\romannumeral0\XINTinfloatmul [#3]{#1}{#2}}{#3}%
}%
\def\XINT_flpow_prd_end\relax\XINT_flpow_prd_compute
    \relax\Z #1#2#3%
{%
    \expandafter\XINT_flpow_conclude \the\numexpr #3\relax #1%
```

```
}%
\def\XINT_flpow_conclude #1#2[#3]#4%
{%
    \expandafter\XINT_flpow_conclude_really\expandafter
    {\the\numexpr\if #41 -\fi#3\expandafter}%
    \xint_UDzerofork
        #4\dummy {{#2}}%
         0\dummy {{1/#2}}%
    \krof #1%
}%
\def\XINT_flpow_conclude_really #1#2#3#4%
{%
    \xint_UDzerofork
    #3\dummy {#4{#2[#1]}}%
     0\dummy {#4{-#2[#1]}}%
    \krof
}%
```

33.72 \xintFloatPower

1.07

```
\def\xintFloatPower {\romannumeral0\xintfloatpower}%
\def\xintfloatpower #1{\XINT_flpower_chkopt \xintfloat #1\Z }%
\def\XINTinFloatPower {\romannumeral0\XINTinfloatpower}%
\def\XINTinfloatpower #1{\XINT_flpower_chkopt \XINT_inFloat #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-\dummy  \XINT_flpower_BisZero
      0#1\dummy  {\XINT_flpower_checkB_b 1}%
       0-\dummy  {\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\XINT_length{#2}}{#3}{#2}#1%
}%
\def\XINT_flpower_checkB_c #1#2%
{%
    \expandafter\XINT_flpower_checkB_d \expandafter
    {\the\numexpr \expandafter\XINT_Length\expandafter
                  {\the\numexpr #1*20/3}+#1+#2+1}%
}%
\def\XINT_flpower_checkB_d #1#2#3#4%
{%
    \expandafter \XINT_flpower_a
    \romannumeral0\XINT_inFloat [#1]{#4}{#1}{#2}#3%
}%
\def\XINT_flpower_a #1%
{%
    \xint_UDzerominusfork
      #1-\dummy \XINT_flpower_zero
      0#1\dummy {\XINT_flpower_b 1}%
       0-\dummy {\XINT_flpower_b 0#1}%
    \krof
}%
\def\XINT_flpower_zero [#1]#2#3#4#5%
{%
    \if #41
          \xint_afterfi {\xintError:DivisionByZero\space 1.e2147483647}%
    \else \xint_afterfi { 0.e0}\fi
}%
\def\XINT_flpower_b #1#2[#3]#4#5%
{%
    \XINT_flpower_c {#4}{#5}{#2[#3]}{#1*\xintOdd {#5}}%
}%
\def\XINT_flpower_c #1#2#3#4%
{%
    \XINT_flpower_loop {#1}{#2}{#3}{{#1}}\XINT_flpow_prd
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax {#4}%
}%
\def\XINT_flpower_loop #1#2#3%
```

```
{%
    \ifcase\XINT_isOne {#2}
        \xint_afterfi{\expandafter\XINT_flpower_loop_x\expandafter
            {\romannumeral0\XINTinfloatmul [#1]{#3}{#3}}%
            {\romannumeral0\xintdivision {#2}{2}}}%
    \or \expandafter\XINT_flpow_loop_end
    \fi
    {#1}{{#3}}%
}%
\def\XINT_flpower_loop_x #1#2{\expandafter\XINT_flpower_loop_a #2{#1}}%
\def\XINT_flpower_loop_a #1#2#3#4%
{%
    \ifnum #2 = 1
        \expandafter\XINT_flpower_loop
    \else
        \expandafter\XINT_flpower_loop_throwaway
    \fi
    {#4}{#1}{#3}%
}%
\def\XINT_flpower_loop_throwaway #1#2#3#4%
{%
   \XINT_flpower_loop {#1}{#2}{#3}%
}%
```

33.73 \xintFloatSqrt

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 \XINT_inFloat #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\XINT_inFloat [#1]{#2}%
}%
\def\XINT_FL_sqrt_checkifzeroorneg #1%
{%
    \xint_UDzerominusfork
     #1-\dummy  \XINT_FL_sqrt_iszero
     0#1\dummy  \XINT_FL_sqrt_isneg
      0-\dummy {\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\XINT_length {#2}}\XINT_flsqrt_big_d {#2}{#1}%
}%
\def\XINT_flsqrt_big_d #1\or #2\fi #3%
{%
   \fi
   \ifodd #3
     \xint_afterfi{\expandafter\XINT_flsqrt_big_eB}%
   \else
     \xint_afterfi{\expandafter\XINT_flsqrt_big_eA}%
   \fi
   \expandafter {\the\numexpr (#3-\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\xintidivision
    {#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%
{%
    \ifcase\XINT_Sgn {#1}
        \expandafter \XINT_flsqrt_big_end_a
    \or \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{\xintiQuo{\XINT_dsx_addzerosnofuss {#4}{#2}}{#3}}}}%
}%
\def\XINT_flsqrt_big_end_b #1#2{#2[#1]}%
\XINT_restorecatcodes_endinput%
```

34 Package xintseries implementation

The commenting is currently (2013/10/22) very sparse.

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 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}{Package xintfrac is required}%
         \y{xintseries}{Will try \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}{Package xintfrac is required}%
            \y{xintseries}{Will try \string\RequirePackage{xintfrac}}%
            \def\z{\endgroup\RequirePackage{xintfrac}}%
          \fi
      \else
        \y{xintseries}{I was already loaded, aborting input}%
        \aftergroup\endinput
      \fi
    \fi
  \fi
\z%
```

34.2 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%
```

34.3 Catcodes

```
\XINTsetupcatcodes%
```

34.4 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintseries}%
  [2013/10/22 v1.09d Expandable partial sums with xint package (jfB)]%
```

34.5 \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%
}%
```

34.6 \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%
}%
```

34.7 \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}%
}%
```

34.8 \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}%
}%
```

34.9 \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}%
}%
```

34.10 \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}}%
}%
```

34.11 \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_fppowseries_loop_ii #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_fppowseries_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_fppowseries_exit_i\fi\expandafter\XINT_fppowseries_loop_ii
    {\fi \expandafter\XINT_fppowseries_exit_ii }%
\def\XINT_fppowseries_exit_ii #1#2#3#4#5#6#7%
{%
    \xinttrunc {#7}
    {\xintiiadd {#4}{\xintiTrunc {#7}{\xintMul {#6{#2}}{#1}}}[-#7]}%
}%
```

34.12 \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_fppowseriesx\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
}%
\def\XINT_fppowseriesx #1#2#3#4#5%
{%
   \ifnum #2<#1
      \xint_afterfi { 0}%
   \else
      \xint_afterfi
        {\expandafter \XINT_fppowseriesx_pre \expandafter
         {\romannumeral-`0#4}{#1}{#2}{#3}{#5}%
        }%
   \fi
}%
\def\XINT_fppowseriesx_pre #1#2#3#4#5%
{%
    \expandafter\XINT_fppowseries_loop_pre\expandafter
       {\romannumeral0\xinttrunc {#5}{\xintPow {#1}{#2}}}%
       {#2}{#1}{#3}{#4}{#5}%
}%
```

34.13 \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}}%
}%
```

34.14 \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%
```

35 Package xintcfrac implementation

The commenting is currently (2013/10/22) 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 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}{Package xintfrac is required}%
         \y{xintcfrac}{Will try \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}{Package xintfrac is required}%
            \y{xintcfrac}{Will try \string\RequirePackage{xintfrac}}%
            \def\z{\endgroup\RequirePackage{xintfrac}}%
          \fi
      \else
        \y{xintcfrac}{I was already loaded, aborting input}%
        \aftergroup\endinput
      \fi
    \fi
  \fi
\z%
```

35.2 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%
```

35.3 Catcodes

```
\XINTsetupcatcodes%
```

35.4 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintcfrac}%
  [2013/10/22 v1.09d Expandable continued fractions with xint package (jfB)]%
```

35.5 \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\xintidivision {#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}%
```

35.6 \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}%
```

35.7 \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}%
```

35.8 \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\xintidivision {#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}%
```

35.9 \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\xintidivision {#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}}%
```

35.10 \xintFtoGC

```
\def\xintFtoGC {\romannumeral0\xintftogc }%
\def\xintftogc {\xintftocx {+1/}}%
```

35.11 \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\xintiquo {#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-\dummy \XINT_ftcc_integer
      0#1\dummy \XINT_ftcc_En
       0-\dummy {\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\xintiquo {#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-\dummy \XINT_ftcc_end
      0#1\dummy \XINT_ftcc_loop_N
       0-\dummy {\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/}%
}%
```

35.12 \xintFtoCv

```
\def\xintFtoCv {\romannumeral0\xintftocv }%
\def\xintftocv #1%
{%
    \xinticstocv {\xintFtoCs {#1}}%
}%
```

35.13 \xintFtoCCv

```
\def\xintFtoCCv {\romannumeral0\xintftoccv }%
\def\xintftoccv #1%
{%
    \xintigctocv {\xintFtoCC {#1}}%
```

```
}%
```

35.14 \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]
```

35.15 \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]
```

35.16 \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]
```

35.17 \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]
```

35.18 \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}%
```

35.19 \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]
```

35.20 \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}%
```

35.21 \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[0]}}#2}%
\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]
```

35.22 \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%
}%
```

35.23 \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%
}%
```

35.24 \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 { 0/1[0]}%
   \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,{ }%
```

35.25 \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 { 0/1[0]}%
   \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/{ }%
```

35.26 \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 { {0/1[0]}}%
   \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/{ }%
```

35.27 \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}%
```

35.28 \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%
```

36 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. 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 fraction (or macro giving on expansion one such; this does not apply to intermediate computation results, only to user input) in the A/B[n] format *with the brackets* **must** be enclosed in 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.

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@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}{Package xintfrac is required}%
         \y{xintexpr}{Will try \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}{Package xintfrac is required}%
            \y{xintexpr}{Will try \string\RequirePackage{xintfrac}}%
            \def\z{\endgroup\RequirePackage{xintfrac}}%
          \fi
      \else
        \y{xintexpr}{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{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%
```

36.3 Catcodes

```
\XINTsetupcatcodes%
```

36.4 Package identification

```
\XINT_providespackage
\ProvidesPackage{xintexpr}%
  [2013/10/22 v1.09d Expandable expression parser (jfB)]%
```

36.5 Helper macros

```
\def\xint_gob_til_dot #1.{}%
\def\xint_gob_til_dot_andstop #1.{ }%
\def\xint_gob_til_! #1!{}% nota bene: ! is of catcode 11
\def\XINT_expr_unexpectedtoken {\xintError:ignored }%
\def\XINT_newexpr_stripprefix #1>{\noexpand\romannumeral-`0}%
```

36.6 Encapsulation in pseudo names

```
\def\XINT_expr_lock #1!{\expandafter\space\csname .#1\endcsname }%
\def\XINT_expr_unlock  {\expandafter\xint_gob_til_dot\string }%
\def\XINT_expr_usethe  {use_xintthe!\xintError:use_xintthe! }%
\def\XINT_expr_done    {!\XINT_expr_usethe\XINT_expr_print }%
\def\XINT_expr_print   #1{\XINT_expr_unlock #1}%
\def\XINT_flexpr_done    {!\XINT_expr_usethe\XINT_flexpr_print }%
\def\XINT_flexpr_print  #1{\xintFloat:csv{\XINT_expr_unlock #1}}%
\def\XINT_numexpr_print #1{\xintRound:csv{\XINT_expr_unlock #1}}%
\def\XINT_boolexpr_print #1{\xintIsTrue:csv{\XINT_expr_unlock #1}}%
```

36.7 \xintifboolexpr, \xintifboolfloatexpr

1.09c. Not to be used on comma separated expressions. I could perhaps use \xintORof:csv (or AND, or XOR) to allow it?

```
\def\xintifboolexpr #1{\romannumeral0\xintifnotzero {\xinttheexpr #1\relax}}%
\def\xintifboolfloatexpr #1{\romannumeral0\xintifnotzero
                                {\xintthefloatexpr #1\relax}}%
```

36.8 \xintexpr, \xinttheexpr, \xintthe

```
\def\xintexpr    {\romannumeral0\xinteval }%
\def\xinteval
{%
    \expandafter\XINT_expr_until_end_a \romannumeral-`0\XINT_expr_getnext
}%
\def\xinttheeval {\expandafter\xint_gobble_ii\romannumeral0\xinteval }%
\def\xinttheexpr {\romannumeral-`0\xinttheeval }%
\def\XINT_numexpr_post !\XINT_expr_usethe\XINT_expr_print%
                      { !\XINT_expr_usethe\XINT_numexpr_print }%
\def\xintnumexpr {\romannumeral0\expandafter\XINT_numexpr_post
                  \romannumeral0\xinteval }%
\def\xintthenumexpr   {\romannumeral-`0\xintthe\xintnumexpr }%
\def\XINT_boolexpr_post !\XINT_expr_usethe\XINT_expr_print%
                      { !\XINT_expr_usethe\XINT_boolexpr_print }%
\def\xintboolexpr {\romannumeral0\expandafter\XINT_boolexpr_post
```

```
                    \romannumeral0\xinteval }%
\def\xintthefboolexpr  {\romannumeral-`0\xintthe\xintboolexpr }%
\def\xintfloatexpr    {\romannumeral0\xintfloateval }%
\def\xintfloateval
{%
    \expandafter\XINT_flexpr_until_end_a \romannumeral-`0\XINT_expr_getnext
}%
\def\xintthefloatexpr {\romannumeral-`0\xintthe\xintfloatexpr }%
\def\xintthe #1{\romannumeral-`0\expandafter\xint_gobble_ii\romannumeral-`0#1}%
```

36.9 \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 \xintflexpr. Perhaps some of the comments are now obsolete.

```
\def\XINT_expr_getnext
{%
    \expandafter\XINT_expr_getnext_checkforbraced_a
    \romannumeral-`0\romannumeral-`0%
}%
\def\XINT_expr_getnext_checkforbraced_a #1%
{%
    \XINT_expr_getnext_checkforbraced_b #1\W\Z {#1}%
}%
\def\XINT_expr_getnext_checkforbraced_b #1#2%
{%
```

```
    \xint_UDwfork
      #1\dummy \XINT_expr_getnext_emptybracepair
      #2\dummy \XINT_expr_getnext_onetoken_perhaps
      \W\dummy \XINT_expr_getnext_gotbracedstuff
    \krof
}%
\def\XINT_expr_getnext_onetoken_perhaps\Z #1%
{%
    \expandafter\XINT_expr_getnext_checkforbraced_c\expandafter
    {\romannumeral-`0#1}%
}%
\def\XINT_expr_getnext_checkforbraced_c #1%
{%
    \XINT_expr_getnext_checkforbraced_d #1\W\Z {#1}%
}%
\def\XINT_expr_getnext_checkforbraced_d #1#2%
{%
    \xint_UDwfork
      #1\dummy \XINT_expr_getnext_emptybracepair
      #2\dummy \XINT_expr_getnext_onetoken_wehope
      \W\dummy \XINT_expr_getnext_gotbracedstuff
    \krof
}% doubly braced things are not acceptable, will cause errors.
\def\XINT_expr_getnext_emptybracepair #1{\XINT_expr_getnext }%
\def\XINT_expr_getnext_gotbracedstuff #1\W\Z #2% {..} -> number/fraction
{%
    \expandafter\XINT_expr_getop\csname .#2\endcsname
}%
\def\XINT_expr_getnext_onetoken_wehope\Z #1% #1 isn't a control sequence!
{%
    \xint_gob_til_! #1\XINT_expr_subexpr !%
    \expandafter\XINT_expr_getnext_onetoken_fork\string #1%
}% after this #1 should be now a catcode 12 token.
\def\XINT_expr_subexpr !#1!{\expandafter\XINT_expr_getop\xint_gobble_ii }%
```

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 #.

```
\begingroup
\lccode`*=`#
\lowercase{\endgroup
\def\XINT_expr_sixwayfork #1(-.+*\dummy #2#3\krof {#2}%
\def\XINT_expr_getnext_onetoken_fork #1%
{% The * is in truth catcode 12 #. For (clever!) use with \xintNewExpr.
    \XINT_expr_sixwayfork
       #1-.+*\dummy  {\xint_c_xviii ({}}% back to until to trigger oparen
       (#1.+*\dummy   -%
```

```
        (-#1+*\dummy  {\XINT_expr_scandec_II.}%
        (-.#1*\dummy   \XINT_expr_getnext%
         (-.+#1\dummy {\XINT_expr_scandec_II}%
         (-.+*\dummy  {\XINT_expr_scan_dec_or_func #1}%
    \krof
}}%
```

36.10 \XINT_expr_scan_dec_or_func: collecting an integer or decimal number or function name

```
\def\XINT_expr_scan_dec_or_func #1% this #1 of catcode 12
{%
    \ifnum \xint_c_ix<1#1
        \expandafter\XINT_expr_scandec_I
    \else % We assume we are dealing with a function name!!
        \expandafter\XINT_expr_scanfunc
    \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_expr_scanfunc_b #1%
{%
    \if #1(\expandafter \xint_gobble_iii\fi
    \xint_firstofone
    {% added in 1.09c for bool and togl
     \if #1)\expandafter \xint_gobble_i
      \else \expandafter \xint_firstoftwo
     \fi }%
    {\XINT_expr_scanfunc_c #1}(%
}%
\def\XINT_expr_scanfunc_panic {\xintError:bigtroubleahead(0\relax }%
\def\XINT_expr_func #1(% common to expr and flexpr
{%
    \xint_c_xviii @{#1}% functions have the highest priority.
```

```
}%
```

Scanning for a number of fraction. Once gathered, lock it and do _getop.

```
\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%
{%
    \ifnum \xint_c_ix<1\string#1
       \expandafter\expandafter\expandafter\XINT_expr_scanintpart_b
       \expandafter\string
    \else
       \if #1.%
           \expandafter\expandafter\expandafter
           \XINT_expr_scandec_transition
       \else
           \expandafter\expandafter\expandafter !% ! of catcode 11...
       \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 #1%
{%
    \expandafter.\romannumeral-`0\expandafter
    \XINT_expr_scanfracpart_a\romannumeral-`0\romannumeral-`0%
}%
\def\XINT_expr_scanfracpart_a #1%
{%
    \ifnum \xint_c_ix<1\string#1
       \expandafter\expandafter\expandafter\XINT_expr_scanfracpart_b
       \expandafter\string
    \else
       \expandafter !%
    \fi
    #1%
}%
\def\XINT_expr_scanfracpart_b #1%
```

```
{%
    \expandafter #1\romannumeral-`0\expandafter
    \XINT_expr_scanfracpart_a\romannumeral-`0\romannumeral-`0%
}%
```

36.11 \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.

```
\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 an un-expandable control sequence is found, must be the ending \relax
    \ifcat #2\relax
        \ifx #2\relax
             \expandafter\expandafter\expandafter
             \XINT_expr_foundend
        \else
             \XINT_expr_unexpectedtoken
             \expandafter\expandafter\expandafter
             \XINT_expr_getop
        \fi
    \else
        \expandafter\XINT_expr_foundop\expandafter #2%
    \fi
    #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
}%
```

36.12 Parentheses

1.09a removes some doubling of \romannumeral-`\0 from 1.08b which served no useful purpose here (I think...).

```
\def\xint_tmp_do_defs #1#2#3#4#5%
{%
    \def#1##1%
    {%
        \xint_UDsignfork
                     ##1\dummy {\expandafter#1\romannumeral-`0#3}%
                       -\dummy  {#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
    }%
}%
\expandafter\xint_tmp_do_defs
    \csname XINT_expr_until_end_a\expandafter\endcsname
    \csname XINT_expr_until_end_b\expandafter\endcsname
    \csname XINT_expr_op_-vi\expandafter\endcsname
    \csname XINT_expr_done\endcsname
    {expr}%
\expandafter\xint_tmp_do_defs
    \csname XINT_flexpr_until_end_a\expandafter\endcsname
    \csname XINT_flexpr_until_end_b\expandafter\endcsname
    \csname XINT_flexpr_op_-vi\expandafter\endcsname
    \csname XINT_flexpr_done\endcsname
    {flexpr}%
\def\XINT_expr_extra_closing_paren {\xintError:removed }%
\def\xint_tmp_do_defs #1#2#3#4#5#6%
{%
    \def #1{\expandafter #3\romannumeral-`0\XINT_expr_getnext }%
    \let #2#1%
    \def #3##1{\xint_UDsignfork
```

```
                ##1\dummy {\expandafter #3\romannumeral-`0#5}%
                  -\dummy  {#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
    }%
}%
\expandafter\xint_tmp_do_defs
    \csname XINT_expr_op_(\expandafter\endcsname
    \csname XINT_expr_oparen\expandafter\endcsname
    \csname XINT_expr_until_)_a\expandafter\endcsname
    \csname XINT_expr_until_)_b\expandafter\endcsname
    \csname XINT_expr_op_-vi\endcsname
    {expr}%
\expandafter\xint_tmp_do_defs
    \csname XINT_flexpr_op_(\expandafter\endcsname
    \csname XINT_flexpr_oparen\expandafter\endcsname
    \csname XINT_flexpr_until_)_a\expandafter\endcsname
    \csname XINT_flexpr_until_)_b\expandafter\endcsname
    \csname XINT_flexpr_op_-vi\endcsname
    {flexpr}%
\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_expr_op_)\endcsname\XINT_expr_getop
\expandafter\let\csname XINT_flexpr_precedence_)\endcsname \xint_c_i
\expandafter\let\csname XINT_flexpr_op_)\endcsname\XINT_expr_getop
```

36.13 The \XINT_expr_until_<op> macros for boolean operators, comparison operators, arithmetic operators, scientfic notation.

Extended in 1.09a with comparison and boolean operators.

```
\def\xint_tmp_def #1#2#3#4#5#6%
{%
    \expandafter\xint_tmp_do_defs
    \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}%
}%
\def\xint_tmp_do_defs #1#2#3#4#5#6#7#8%
```

```
{%
    \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\dummy {\expandafter #2\expandafter ##1\romannumeral-`0#4}%
          -\dummy {#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{\XINT_expr_unlock ##1}{\XINT_expr_unlock ##4}\endcsname }%
      \fi
    }%
    \let #7#5%
}%
\def\xint_tmp_def_a #1{\xint_tmp_def {expr}{xint}#1}%
\xintApplyInline {\xint_tmp_def_a }{%
  {|{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_tmp_def_a #1{\xint_tmp_def {flexpr}{xint}#1}%
\xintApplyInline {\xint_tmp_def_a }{%
  {|{iii}{vi}{OR}}%
  {&{iv}{vi}{AND}}%
  {<{v}{vi}{Lt}}%
  {>{v}{vi}{Gt}}%
  {={v}{vi}{Eq}}%
}%
\def\xint_tmp_def_a #1{\xint_tmp_def {flexpr}{XINTinFloat}#1}%
\xintApplyInline {\xint_tmp_def_a }{%
```

```
  {+{vi}{vi}{Add}}%
  {-{vi}{vi}{Sub}}%
  {*{vii}{vii}{Mul}}%
  {/{vii}{vii}{Div}}%
  {^{viii}{viii}{Power}}%
  {e{ix}{ix}{fE}}%
  {E{ix}{ix}{fE}}%
}%
\let\xint_tmp_def_a\empty
```

36.14 The comma as binary operator

New with 1.09a.

```
\def\xint_tmp_do_defs #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\dummy {\expandafter #2\expandafter ##1\romannumeral-`0#4}%
          -\dummy {#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
}%
\expandafter\xint_tmp_do_defs
    \csname XINT_expr_op_,\expandafter\endcsname
    \csname XINT_expr_until_,_a\expandafter\endcsname
    \csname XINT_expr_until_,_b\expandafter\endcsname
    \csname XINT_expr_op_-vi\expandafter\endcsname
    \csname XINT_expr_precedence_,\endcsname {expr}%
\expandafter\xint_tmp_do_defs
    \csname XINT_flexpr_op_,\expandafter\endcsname
    \csname XINT_flexpr_until_,_a\expandafter\endcsname
    \csname XINT_flexpr_until_,_b\expandafter\endcsname
    \csname XINT_flexpr_op_-vi\expandafter\endcsname
```

```
    \csname XINT_expr_precedence_,\endcsname {flexpr}%
```

36.15 \XINT_expr_op_-<level>: minus as prefix inherits its precedence level

```
\def\xint_tmp_def #1#2%
{%
    \expandafter\xint_tmp_do_defs
    \csname XINT_#1_op_-#2\expandafter\endcsname
    \csname XINT_#1_until_-#2_a\expandafter\endcsname
    \csname XINT_#1_until_-#2_b\expandafter\endcsname
    \csname xint_c_#2\endcsname {#1}%
}%
\def\xint_tmp_do_defs #1#2#3#4#5%
{%
    \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\dummy {\expandafter #2\romannumeral-`0#1}%
          -\dummy {#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 .\xintOpp{\XINT_expr_unlock ##3}\endcsname }%
        \fi
    }%
}%
\xintApplyInline{\xint_tmp_def {expr}}{{vi}{vii}{viii}{ix}}%
\xintApplyInline{\xint_tmp_def {flexpr}}{{vi}{vii}{viii}{ix}}%
```

36.16 ? 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_?
```

36.17 : 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_:
```

36.18 ! 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 }% [0] removed in 1.09c
\let\XINT_flexpr_op_!\XINT_expr_op_!
```

36.19 Functions

New with 1.09a.

```
\let\xint_tmp_def\empty
\let\xint_tmp_do_defs\empty
\def\XINT_expr_op_@ #1%
{%
    \ifcsname XINT_expr_onlitteral_#1\endcsname
       \expandafter\XINT_expr_funcoflitteral
    \else
       \expandafter\XINT_expr_op_@@
    \fi {#1}%
}%
\def\XINT_flexpr_op_@ #1%
{%
    \ifcsname XINT_expr_onlitteral_#1\endcsname
       \expandafter\XINT_expr_funcoflitteral
    \else
       \expandafter\XINT_flexpr_op_@@
```

```
    \fi {#1}%
}%
\def\XINT_expr_funcoflitteral #1%
{%
    \expandafter\expandafter\csname XINT_expr_onlitteral_#1\endcsname
    \romannumeral-`0\XINT_expr_scanfunc
}%
\def\XINT_expr_op_@@ #1%
{%
    \ifcsname XINT_expr_func_#1\endcsname
    \xint_afterfi{\expandafter\expandafter\csname XINT_expr_func_#1\endcsname}%
    \else \xintError:unknownfunction
          \xint_afterfi{\expandafter\XINT_expr_func_unknown}%
    \fi
    \romannumeral-`0\XINT_expr_oparen
}%
\def\XINT_flexpr_op_@@ #1%
{%
    \ifcsname XINT_flexpr_func_#1\endcsname
    \xint_afterfi{\expandafter\expandafter\csname XINT_flexpr_func_#1\endcsname}%
    \else \xintError:unknownfunction
          \xint_afterfi{\expandafter\XINT_expr_func_unknown}%
    \fi
    \romannumeral-`0\XINT_flexpr_oparen
}%
\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%
{%
    \expandafter #1\expandafter #2\csname .0[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
\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 {\XINT_expr_unlock #3}{\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_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_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
\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
\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_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_expr_oneortwo #1#2#3,#4,#5.%
{%
    \if\relax#5\relax\expandafter\xint_firstoftwo\else
                     \expandafter\xint_secondoftwo\fi
    {#1{0}{#3}}{#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_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_expr_argandopt #1,#2,#3.%
{%
    \if\relax#3\relax\expandafter\xint_firstoftwo\else
                     \expandafter\xint_secondoftwo\fi
    {[\XINTdigits]{#1}}{[\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
\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_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
\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
\def\XINT_expr_func_max #1#2#3%
```

```
{%
    \expandafter #1\expandafter #2\csname
                .\xintMaxof:csv{\XINT_expr_unlock #3}\endcsname
}%
\def\XINT_flexpr_func_max #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
                .\xintFloatMaxof: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_flexpr_func_min #1#2#3%
{%
    \expandafter #1\expandafter #2\csname
                .\xintFloatMinof: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
                .\xintFloatSum: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
                .\xintFloatPrd: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
\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_?
```

```
\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_!
\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
\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
\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
\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
\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
\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
```

36.20 \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.

```
\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),(num,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,%
        Lt,Gt,Eq,AND,OR,IsNotZero,IsZero,ifNotZero,ifSgn,%
        Irr,Num,Abs,Sgn,Opp,Quo,Rem,Add,Sub,Mul,Sqr,Div,Pow,Fac,fE} \do
 {\toks0
  \expandafter{\the\toks0\expandafter\def\csname xint#1\endcsname {_xint#1}}}%
\xintFor #1 in {GCDof,LCMof,Maxof,Minof,ANDof,ORof,XORof,%
                FloatMaxof,FloatMinof,Sum,Prd,FloatSum,FloatPrd} \do
 {\toks0
  \expandafter{\the\toks0\expandafter\def\csname xint#1:csv\endcsname
                            ####1{_xint#1 {\xintCSVtoList {####1}}}}}%
\xintFor #1 in {,Sqrt,Add,Sub,Mul,Div,Power,fE} \do
  {\toks0
   \expandafter{\the\toks0\expandafter\def\csname XINTinFloat#1\endcsname
                                           {_XINTinFloat#1}}}%
\expandafter\def\expandafter\XINT_expr_protect\expandafter{\the\toks0
    \def\XINTdigits {_XINTdigits}%
    \def\XINT_expr_print ##1{\expandafter\XINT_newexpr_print\expandafter
                {\romannumeral0\xintcsvtolist{\XINT_expr_unlock ##1}}}%
    \def\XINT_flexpr_print ##1{\expandafter\XINT_newflexpr_print\expandafter
                {\romannumeral0\xintcsvtolist{\XINT_expr_unlock ##1}}}%
    \def\XINT_numexpr_print ##1{\expandafter\XINT_newnumexpr_print\expandafter
                {\romannumeral0\xintcsvtolist{\XINT_expr_unlock ##1}}}%
    \def\XINT_boolexpr_print ##1{\expandafter\XINT_newboolexpr_print\expandafter
                {\romannumeral0\xintcsvtolist{\XINT_expr_unlock ##1}}}%
}%
\toks0 {}%
```

```
\def\xintNewExpr      {\xint_NewExpr\xinttheexpr      }%
\def\xintNewFloatExpr {\xint_NewExpr\xintthefloatexpr }%
\def\xintNewNumExpr   {\xint_NewExpr\xintthenumexpr   }%
\def\xintNewBoolExpr  {\xint_NewExpr\xinttheboolexpr  }%
\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
    \XINT_NewExpr #1%
}%
\catcode`* 13
\def\XINT_NewExpr #1#2%
{%
    \def\xintTmp ##1##2##3##4##5##6##7##8##9{#2}%
    \XINT_expr_protect
    \lccode`\*=`_ \lowercase {\def*}{!noexpand!}%
    \catcode`_ 13 \catcode`: 11 \endlinechar -1
    \everyeof {\noexpand }%
    \edef\XINTtmp ##1##2##3##4##5##6##7##8##9%
        {\scantokens
         \expandafter{\romannumeral-`0#1%
                      \xintTmp {####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_stripprefix\meaning\XINTtmp}}%
  \endgroup
}%
\let\xintexprRestoreCatcodes\relax
\def\xintexprSafeCatcodes
{% for end user.
    \edef\xintexprRestoreCatcodes  {%
        \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\relax   % =
    }% this is just for some standard situation with a few made active by Babel
        \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  % =
}%
\XINT_restorecatcodes_endinput%
```

xint:	4250.	Total number of code lines: 9986. Each package starts with circa 80 lines dealing with catcodes, package identification and reloading management, also for Plain TeX. Version 1.09d of 2013/10/22.
xintbinhex:	642.	
xintgcd:	473.	
xintfrac:	2333.	
xintseries:	419.	
xintcfrac:	946.	
xintexpr:	923.	