

The **xint** bundle

JEAN-FRANÇOIS BURNOL

jfbu (at) free (dot) fr

Package version: 1.09h (2013/11/28)

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

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

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

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

Further modules:

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

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

xintgcd implements the Euclidean algorithm and its typesetting.

xintcfrc 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	2	11	Use of count registers	18
2	Recent changes	3	12	Dimensions	19
3	Overview	6	13	\ifcase, \ifnum, ... constructs	19
4	Missing things	7	14	Assignments	20
5	Some examples	7	15	Utilities for expandable manipulations	22
6	Origins of the package	11	16	A new kind of for loop	22
7	Expansions	11	17	A new kind of expandable loop	22
8	Input formats	13	18	Exceptions (error messages)	22
9	Output formats	16	19	Common input errors when using the package macros	23
10	Multiple outputs	18			

20 Package namespace	24	23 The <code>\xintexpr</code> math parser (I)	25
21 Loading and usage	24	24 The <code>\xintexpr</code> math parser (II)	27
22 Installation	25		
25 Commands of the <code>xinttools</code> package	31	33 Package <code>xinttools</code> implementation	125
26 Commands of the <code>xint</code> package	59	34 Package <code>xint</code> implementation	153
27 Commands of the <code>xintfrac</code> package	70	35 Package <code>xintbinhex</code> implementation	241
28 Expandable expressions with the <code>xintexpr</code> package	81	36 Package <code>xintgcd</code> implementation	256
29 Commands of the <code>xintbinhex</code> package	88	37 Package <code>xintfrac</code> implementation	270
30 Commands of the <code>xintgcd</code> package	90	38 Package <code>xintseries</code> implementation	325
31 Commands of the <code>xintseries</code> package	93	39 Package <code>xintcfrc</code> implementation	336
32 Commands of the <code>xintcfrc</code> package	110	40 Package <code>xintexpr</code> implementation	357

1 Quick introduction

The `xint` bundle consists of the three principal components `xint`, `xintfrac` (which loads `xint`), and `xintexpr` (which loads `xintfrac`), and four additional modules. Release 1.09g has moved the macros of `xint` not dealing with the manipulation of big numbers to a separate package `xinttools` (which is automatically loaded by `xint`), of independent interest.

All components may be used as regular packages with \LaTeX or loaded directly via `\input` (e.g. `\input xint.sty\relax`) in any other format based on \TeX . Each of them automatically loads those not already loaded it depends on.

The ε - \TeX extensions must be enabled; this is the case in modern distributions by default, except if you invoke \TeX under the name `tex` in command line (`etex` should be used then, or `pdftex` in DVI output mode).

The goal is to compute *exactly*, purely by expansion, without count registers nor assignments nor definitions, with arbitrarily big numbers and fractions. The only non-algebraic operation which is currently implemented is the extraction of square roots.

2 Recent changes

The package macros expand their arguments¹; 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 (expandable!) `\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 macros (not provided by the package:)

```
\def\allowssplits #1%
  {\ifx #1\relax \else #1\hskip 0pt plus 1pt\relax\expandafter\allowssplits\fi}%
\def\printnumber #1{\expandafter\expandafter\expandafter\allowssplits #1\relax}%
%% (all macros from the xint bundle expand in two steps to their final output).
```

An alternative (footnote 10) 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 across lines). Recently I became aware of the `seqsplit` package² which can be used to achieve this splitting across lines, and does work in inline math mode.

The utilities provided by `xinttools` (section 25), some completely expandable, others not, are of independent interest. Their use is illustrated through various examples: among those, it is shown in subsection 25.25 how to implement in a completely expandable way the quick sort algorithm and also how to illustrate it graphically. Other examples include some dynamically constructed alignments with automatically computed prime number cells (subsection 25.11, subsection 25.20).

Some other computational examples are the computations of π and $\log 2$ using `xint` and the computation of the convergents of e with the further help of the `xintcfrac` package.

2 Recent changes

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

- parts of the documentation have been re-written or re-organized, particularly the discussion of expansion issues and of input and output formats.
- the expansion types of macro arguments are documented in the margin of the macro descriptions, with conventions mainly taken over from those in the L^AT_EX3 documentation.
- a dependency of `xinttools` on `xint` (inside `\xintSeq`) has been removed.
- `\xintTypesetEuclideanAlgorithm` and `\xintTypesetBezoutAlgorithm` have been slightly modified (regarding indentation).
- macros `\xintiSum` and `\xintiPrd` are renamed to `\xintiiSum` and `\xintiiPrd`.
- a count register used in 1.09g in the `\xintFor` loops for parsing purposes has been removed and replaced by use of a `\numexpr`.
- the few uses of `\loop` have been replaced by `\xintloop`/`\xintilloop`.
- all macros of `xinttools` for which it makes sense are now declared `\long`.

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

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

¹see in section 7 the related explanations.

²<http://ctan.org/pkg/seqsplit>

2 Recent changes

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

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

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

- new `\xintintegers`, `\xintdimensions`, `\xintrationals` for infinite `\xintFor` loops, interrupted with `\xintBreakFor` and `\xintBreakForAndDo`.
- new `\xintifForFirst`, `\xintifForLast` for the `\xintFor` and `\xintFor*` loops,
- the `\xintFor` and `\xintFor*` loops are now `\long`, the replacement text and the items may contain explicit `\par`’s.
- bug fix, the `\xintFor` loop (not `\xintFor*`) did not correctly detect an empty list.
- new conditionals `\xintifCmp`, `\xintifInt`, `\xintifOdd`.
- bug fix, `\xintiSqrt {0}` crashed. :-((
- the documentation has been enriched with various additional examples, such as the [the quick sort algorithm illustrated](#) or the computation of some prime tables ([subsection 25.11](#), [subsection 25.20](#)).
- the documentation explains with more details various expansion related issues, particularly in relation to conditionals.

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

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

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

- added `bool` and `togl` to the `\xintexpr` syntax; also added `\xintboolexpr` and `\xintifboolexpr`.
- added `\xintNewNumExpr` 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`,
- `\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.

2 Recent changes

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 defined 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.

3 Overview

- 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 32767.³ 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 \LaTeX 3 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 $\varepsilon\text{\TeX}$ ’s `\numexpr` which does expandable computations using standard infix notations with \TeX integers. But $\varepsilon\text{\TeX}$ 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 $\varepsilon\text{\TeX}$ 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.^{4, 5}

³but values higher than 100 or 200 will presumably give too slow evaluations.

⁴currently (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’.

⁵multiplication of two floats with $P=\text{\xinttheDigits}$ 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.)

4 Missing things

The L^AT_EX3 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.⁶

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 T_EX 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.^{7 8}

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.

5 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}: 11473818116626655663327333000845458674702548042  
34261029758895454373590894697032027622647054266320583469027086822116
```

⁶without 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.

⁷I could, naturally, be proven wrong!

⁸The LuaT_EX project possibly makes endeavours such as [xint](#) appear even more insane that they are, in truth.

5 Some examples

81334152500324038762776168953222117634295872033762216088606915850757
16801971671071208769703353650737748777873778498781606749999798366581
25172327521549705416595667384911533326748541075607669718906235189958
32377826369998110953239399323518999222056458781270149587767914316773
54372538584459487155941215197416398666125896983737258716757394949435
52017095026186580166519903071841443223116967837696

1234/56789 with 1500 digits after the decimal point:

```
\xintTrunc{1500}{1234/56789}\dots: 0.021729560302171195125816619415731
91991406786525559527373258905774005529239817570304108189966366725950
44815016992727464825934600010565426403000581098452165031960414869076
75782281779922168025497895719241402384264558277131134550705242212400
28878832168201588335769251087358467308809804715701984539259363609149
65926499850323125957491767771927662047227456021412597510081177692863
05446477310746799556252091073975593865009068657662575498776171441652
43268942929088379791861099860888552360492348870379827079187870890489
35533289897691454330944373029988201940516649351106728415714310870062
86428709785345753579038194016446847100670904576590536899751712479529
48634418637412174893025057669619116378171829051400799450597827043969
78288048741833805842680800859321347444047262674109422599447076018242
96958918100336332740495518498300727253517406539998943457359699941890
15478349680395851309232421771822007783197450210428075859761573544172
28688654492947577875997112116783179841166423074891264153269119019528
42980154607406363908503407350014967687404250823222807233795277254397
85874024899188223071369455352268925320044374790892602440613499093134
23374245012238285583475673105707091162020813890013911144763950765112
96201729208121291095106446671010230854566905562697001179805948335064
88932715842856891299371357129021465424642096180598355315289932909542
34094631002482875204705136558136258782510697494233038088362182817094
85992005494021729560302171195125816619415731919914067865255595273732
589057740055292398175703041081899663667...
```

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

```
\xinttheexpr trunc(.99^-100,200)\relax\dots: 2.731999026429026003846671
72125783743550535164293857207083343057250824645551870534304481430137
84806140368055624765019253070342696854891531946166122710159206719138
4034885148574794308647096392073177979303...
```

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

```
\xintAssign\xintBezout {\xintthenumexpr 7^200-3^200\relax}
{\xintthenumexpr 2^200-1\relax}\to\A\B\U\V\D
\U$\times$(7^200-3^200)+\xintiOpp\V$\times$(2^200-1)=\D
-220045702773594816771390169652074193009609478853\times(7^200-3^200)+1432
58949362763693185913068326832046547441686338771408915838167247899192
11328201191274624371580391777549768571912876931442406050669914563361
43205677696774891\times(2^200-1)=1803403947125
```

The Euclidean algorithm applied to 22,206,980,239,027,589,097 and 8,169,486,210,102,119,256:⁹

⁹this example is computed tremendously faster than the other ones, but we had to limit the space taken

5 Some examples

```
\xintTypesetEuclideanAlgorithm {22206980239027589097}{8169486210102119256}
22206980239027589097 = 2 × 8169486210102119256 + 5868007818823350585
8169486210102119256 = 1 × 5868007818823350585 + 2301478391278768671
5868007818823350585 = 2 × 2301478391278768671 + 1265051036265813243
2301478391278768671 = 1 × 1265051036265813243 + 1036427355012955428
1265051036265813243 = 1 × 1036427355012955428 + 228623681252857815
1036427355012955428 = 4 × 228623681252857815 + 121932630001524168
228623681252857815 = 1 × 121932630001524168 + 106691051251333647
121932630001524168 = 1 × 106691051251333647 + 15241578750190521
106691051251333647 = 7 × 15241578750190521 + 0
```

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

```
\def\coeff #1%
{\xintiRound {12}{1/\xintiSqr{\the\numexpr 4*#1*#1-9\relax }}[0]}}
\xintRound {9}{\xintiSeries {1}{500}{\coeff}[-12]}: 0.062366080
The complete series, extended to infinity, has value  $\frac{\pi^2}{144} - \frac{1}{162} = 0.062, 366, 079, 945,$ 
836, 595, 346, 844, 45...10 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} expands to:
2.306,488,000,584,534,696,558,06 × 10301,029,995
```

where the \numprint macro from package **numprint** was used.

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

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

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

by the output.

¹⁰This 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 \$\pi\$ from scratch](#).

5 Some examples

```
\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.¹¹ 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`; using seven rather than three `\expandafter`'s in `\printnumber` would allow to use it directly as `\printnumber\mynumber` when `\mynumber` has been defined via a `\def` or `\newcommand` using a chain of package macros.

Just to show off (again), let's print 300 digits (after the decimal point) of the decimal expansion of 0.7^{-25} :¹²

```
\np {\xinttheexpr trunc(.7^-25,300)\relax}\dots
7,456,739,985,837,358,837,609,119,727,341,853,488,853,339,101,579,533,
584,812,792,108,394,305,337,246,328,231,852,818,407,506,767,353,741,
490,769,900,570,763,145,015,081,436,139,227,188,742,972,826,645,967,
904,896,381,378,616,815,228,254,509,149,848,168,782,309,405,985,245,
368,923,678,816,256,779,083,136,938,645,362,240,130,036,489,416,562,
067,450,212,897,407,646,036,464,074,648,484,309,937,461,948,589...
```

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

```
\xintDigits:=20; \np{\xintthefloatexpr .7123045678952^-243\relax}
.7123045678952^-243 ≈ 6.342,022,117,488,416,127,3 × 1035
```

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

To see more of `xint` in action, jump to the [section 31](#) describing the commands of the `xintseries` package, especially as illustrated with the [traditional computations of \$\pi\$ and \$\log 2\$](#) , or also see the [computation of the convergents of \$e\$](#) made with the `xintcfrac` package.

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.

¹¹as 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).

¹²the `\np` typesetting macro is from the `numprint` package.

6 Origins of the package

Package `bigintcalc` by HEIKO OBERDIEK already provides expandable arithmetic operations on “big integers”, exceeding the \TeX limits (of $2^{\{31\}}-1$), so why another¹³ 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.¹⁴ 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 $\varepsilon\text{-TeX}$ `\numexpr` primitive, they worked one digit at a time, having previously stored carry-arithmetic in 1200 macros.

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

The present package is the result of this initial questioning.

7 Expansions

By convention in this manual *f*-expansion (“full expansion” or “full first expansion”) is the process of expanding repeatedly the first token seen until hitting against something not further expandable like an unexpandable \TeX -primitive or an opening brace `{` or a character (inactive). For those familiar with \LaTeX 3 (which is not used by `xint`) this is what is called in its documentation full expansion. Technically, macro arguments in `xint` which are submitted to such a *f*-expansion are so via prefixing them with `\romannumeral-‘0`. An explicit or implicit space token stops such an expansion and is gobbled.

Most of the package macros, and all those dealing with computations, are expandable in the strong sense that they expand to their final result via this *f*-expansion. Again copied from \LaTeX 3 documentation conventions, this will be signaled in the description of the macro by a star in the margin. All¹⁵ expandable macros of the `xint` packages completely expand in two steps.

Furthermore the macros dealing with computations, as well as many utilities from `xint-tools`, apply this process of *f*-expansion to their arguments. Again from \LaTeX 3’s conventions this will be signaled by a margin annotation. Some additional parsing which is done by most macros of `xint` is indicated with a variant; and the extended fraction parsing done by most macros of `xintfrac` has its own symbol. When the argument has a priori to obey the \TeX bound of 2147483647 it is systematically fed to a `\numexpr`. `\relax` hence the expansion is then a *complete* one, signaled with an *x* in the margin. This means not only

Num *f*
f Frac
f
 num
x

¹³this section was written before the `xintfrac` package; the author is not aware of another package allowing expandable computations with arbitrarily big fractions.

¹⁴the `\ReverseOrder` could be avoided in that circumstance, but it does play a crucial rôle here.

¹⁵except `\xintloop` and `\xintilloop`.

complete expansion, but also that spaces are ignored, infix algebra is possible, count registers are allowed, etc. . .

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

f → **f* ^{Frac}

n, resp. *o* A few macros from **xinttools** do not expand, or expand only once their argument. This is also signaled in the margin with notations à la L^AT_EX3.

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

Conversely, wherever the package expects on input a “big” integers, or a “fraction”, *f*-expansion of the argument *must result in a complete expansion* for this argument to be acceptable.^{16 17} The main exception is inside `\xintexpr . . . \relax` where everything will be expanded from left to right, completely.

Summary of important expansion aspects:

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

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

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

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

2. using `\if . . . \fi` constructs inside macro arguments requires suitably mastering `\expandafter`'s and other techniques; it is much easier to use the package pre-defined conditionals such as `\xintifSgn`, `\xintifGt`, or `\xintifOdd` for example, or the **etoolbox**¹⁸ conditionals (for small integers only).

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

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

3. after the definition `\def\x {12}`, one can not use `-\x` as input to one of the package macros: the *f*-expansion will act only on the minus sign, hence do nothing. The only

¹⁶this is not quite as stringent as claimed here, see [section 11](#) for more details.

¹⁷this is particularly important when one tries to insert `\if . . . \fi`'s inside such arguments; suitable `\expandafter`'s or swapping techniques *must be employed* else the expansion from a `\romannumeral-`'0 will not absorb the `\else` or closing `\fi`, and some error will probably arise in further processing. Therefore it is highly recommended to use the package provided conditionals such as `\xintifEq`, `\xintifGt`, . . . , or, for L^AT_EX users and when dealing with short integers the **etoolbox** expandable conditionals such as `\ifnumequal`, `\ifnumgreater`, Use of *non-expandable* things such as `\ifthenelse` is impossible inside the arguments of **xint** macros.

¹⁸<http://www.ctan.org/pkg/etoolbox>

8 Input formats

way is to use the `\xintOpp` macro, or perhaps here rather `\xintiOpp` which does maintains integer format on output, as they replace a number with its opposite.

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

4. With the definition

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

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

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

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

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

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

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

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

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

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

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

8 Input formats

The core bundle constituents are `xint`, `xintfrac`, `xintexpr`, each one loading its predecessor. The base constituent `xint` only handles (big) integers, and `xintfrac` additionally manages decimal numbers, numbers in scientific notation, and fractions. Both load `xint-tools` which provides utilities not directly related to big numbers.

The package macros first *f*-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 $\text{T}_{\text{E}}\text{X}$ 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:

- f* 1. the strict format is for some macros of `xint` which only *f*-expand their arguments. After this *f*-expansion the input should be a string of digits, optionally preceded by a unique minus sign. The first digit can be zero only if the number is zero. A plus sign is not accepted. `-0` is not legal in the strict format.

8 Input formats

Num
f

- the macro `\xintNum` normalizes into strict format an input having arbitrarily many minus and plus signs, followed by a string of zeros, then digits:

```
\xintNum {+---+-----+---+---00000000009876543210}=-9876543210
```

The extended integer format is thus for the arithmetic macros of `xint` which automatically parse their arguments via this `\xintNum`.

Frac
f

- the fraction format is what is expected by the macros of `xintfrac`: a fraction is constituted of a numerator A and optionally a denominator B, separated by a forward slash / and A and B may be macros which will be automatically given to `\xintNum`. Each of A and B may be decimal numbers (the decimal mark must be a `.`). Here is an example:¹⁹

```
\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 for both numerator and denominator of a fraction, and is produced on output by `\xintFloat`:

```
\xintAdd{10.1e1}{101.010e3}=101111/1[0]
```

```
\xintFloatAdd{10.1e1}{101.010e3}=1.011110000000000e5
```

```
\xintFloat{\xintiPow {2}{100}}=1.267650600228229e30
```

Produced fractions with a denominator equal to one are nevertheless generally printed as fraction. In math mode `\xintFrac` will remove such dummy denominators, and in inline text mode one has `\xintPraw` with the similar effect.

```
\xintPraw{\xintAdd{10.1e1}{101.010e3}}=101111
```

```
\xintRaw{1.234e5/6.789e3}=1234/6789[2]
```

```
\xintFloat[24]{1/66049}=1.51402746445820527184363e-5
```

- the `expression format` is for inclusion in an `\xintexpr... \relax`, it uses infix notations, function names, complete expansion, and is described in its devoted section (section 24).

Even with `xintfrac` loaded, some macros by their nature can not accept fractions on input. Those parsing their inputs through `\xintNum` will accept a fraction reducing to an integer. For example `\xintQuo {100/2}{12/3}` works, because its arguments are, after simplification, integers. In this documentation, I often say “numbers or fractions”, although at times the vocable “numbers” by itself may also include “fractions”; and “decimal numbers” are counted among “fractions”.

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 exemplified above by the numerators and denominators (thus, possibly with a ‘scientific’ exponent part, with a lowercase ‘e’). Or one may

¹⁹the square brackets one sees in various outputs are explained near the end of this section.

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

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

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

Num
f

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

Frac
f

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

Summary of the input formats for the bundle macros dealing with numbers (except `\xintexpr...\relax`):

num
x

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.²⁰ 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.

f

2. ‘long’ integers in strict format: only one optional minus sign, anything starting with zero is treated as zero. Some macros of **xint** require this format, but most accept the extended format described in the next item; they may then have a ‘strict’ variant for optimizing purposes with a ‘ii’ in their names, this variant remains available even with **xintfrac** loaded. A count register can serve as argument only if prefixed by `\the` or `\number`.

Num
f

3. ‘long’ integers automatically parsed by `\xintNum`, they may have leading signs followed by leading zeros, and they may be count registers with no need of being prefixed by `\the` or `\number`.²¹ The number of digits must (as in the strict format) be less than 2,147,483,647.

Frac
f

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

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

²⁰the bound has even been lowered for them but the float power function limits the exponent only to the \TeX bound, and 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.

²¹A \TeX `\value{counternum}` is accepted, if there is nothing else, especially before, in the macro argument.

9 Output formats

IMPORTANT!

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

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 ignored (except when they occur inside arguments to some some macros, thus escaping the `\xintexpr` parser). See the [documentation](#).

9 Output formats

With package **xintfrac** loaded, the routines `\xintAdd`, `\xintSub`, `\xintMul`, `\xintPow`, `\xintSum`, `\xintPrd` are modified to allow fractions on input,^{22 23 24 25} and produce on output a fractional number `f=A/B[n]` where `A` and `B` are integers, with `B` positive, and `n` is a “short” integer (*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

²²the power function does not accept a fractional exponent. Or rather, does not expect, and errors will result if one is provided.

²³macros `\xintiAdd`, `\xintiSub`, `\xintiMul`, `\xintiPow`, are the original ones dealing only with integers. They are available as synonyms, also when **xintfrac** is not loaded. With **xintfrac** loaded they accept on input also fractions, if these fractions reduce to integers, and the output format is the original **xint**’s one. The macros `\xintiiAdd`, `\xintiiSub`, `\xintiiMul`, `\xintiiPow`, `\xintiiSum`, `\xintiiPrd` are strictly integer-only: they skip the overhead of parsing their arguments via `\xintNum`.

²⁴also `\xintCmp`, `\xintSgn`, `\xintGeq`, `\xintOpp`, `\xintAbs`, `\xintMax`, `\xintMin` are extended to fractions; and the last four have the integer-only variants `\xintiOpp`, `\xintiAbs`, `\xintiMax`, `\xintiMin`.

²⁵and `\xintFac`, `\xintQuo`, `\xintRem`, `\xintDivision`, `\xintFDg`, `\xintLDg`, `\xintOdd`, `\xintMON`, `\xintMMON` all accept a fractional input as long as it reduces to an integer.

9 Output formats

(and is parsed more quickly than fractions containing decimal points; the input may be a number without denominator).²⁶

Thus loading `xintfrac` not only relaxes the format of the inputs; it also modifies the format of the outputs: except when a fraction is filtered on output by `\xintIrr` or `\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 $\boxed{A/B[n]}$ form (which stands for $(A/B) \times 10^n$). 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. The command `\xintFrac` is not accepted as input to the package macros, it is for typesetting only (in math mode).

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

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. 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` (or one can use `\xintPRaw` on top of `\xintIrr`).

The macro `\xintTrunc{N}{f}` prints²⁷ the decimal expansion of f with N digits after the decimal point.²⁸ 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 $-0.0\dots 0$, 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

²⁶at 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 $2^{31}-9$.

²⁷‘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.

²⁸the current release does not provide a macro to get the period of the decimal expansion.

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`.

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

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

10 Multiple outputs

Some macros have an output consisting of more than one number or fraction, each one is then returned within braces. Examples of multiple-output macros are `\xintDivision` which gives first the quotient and then the remainder of euclidean division, `\xintBezout` from the `xintgcd` package which outputs five numbers, `\xintFtoCv` from the `xintcfrac` package which returns the list of the convergents of a fraction, ... [section 14](#) and [section 15](#) mention utilities, expandable or not, to cope with such outputs.

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

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

11 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 submitted to a complete expansion which must deliver an integer, and count registers and even algebraic expressions with them like `\mycountA+\mycountB*17-\mycountC/12+\mycountD` are admissible arguments (the slash stands here for the rounded integer division done by `\numexpr`). This applies in particular to the number of digits to truncate or round with, to the indices of a series partial sum, ...

The macros dealing with long numbers or fractions allow (when not limited to the ‘strict integer’ format on input) *to some extent* the direct 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*.²⁹ The slash for rounded division in a `\numexpr` should be written with braces `{/}` to not be confused with the `xintfrac` delimiter between numerator and denominator (braces will be removed internally). Example: `\mycountA+\mycountB{/}17/1+\mycountA*\mycountB`, or `\count 0+\count 2{/}17/1+`

IMPORTANT! {²⁹ Attention! there is no problem with a \LaTeX `\value{countname}` if it comes first, but if it comes later in the input it will not get expanded, and braces around the name will be removed and chaos will ensue inside a `\numexpr`. One should enclose the whole input in `\the\numexpr... \relax` in such cases.

`\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.

12 Dimensions

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

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

One may thus compute areas or volumes with no limitations, in units of sp² respectively sp³, do arithmetic with them, compare them, etc..., and possibly express some final result back in another unit, with the suitable conversion factor and a rounding to a given number of decimal places.

13 \ifcase, \ifnum, ... constructs

When using things such as `\ifcase \xintSgn{A}` one has to make sure to leave a space after the closing brace for T_EX to stop its scanning for a number: once T_EX 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\xintSgnA` 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 \xintSgnA      0\or OK\else ERROR\fi    ---> gives ERROR
\ifcase \xintSgnA\space 0\or OK\else ERROR\fi    ---> gives OK
\ifcase \xintSgn{A}    0\or OK\else ERROR\fi    ---> gives OK
```

In order to use successfully `\if...\fi` constructions either as arguments to the **xint** bundle expandable macros, or when building up a completely expandable macro of one’s own, one needs some T_EXnical expertise (this is briefly commented upon in [footnote 17](#)).

It is thus much to be recommended to opt rather for already existing expandable branching macros, such as the ones which are provided by **xint**: `\xintSgnFork`, `\xintifSgn`, `\xintifZero`, `\xintifNotZero`, `\xintifTrueFalse`, `\xintifCmp`, `\xintifGt`, `\xintifLt`, `\xintifEq`, `\xintifOdd`, and `\xintifInt`. See their respective documentations. All these conditionals always have either two or three branches, and empty brace pairs `{}` for unused branches should not be forgotten.

If these tests are to be applied to standard \TeX short integers, it is more efficient to use (under \LaTeX) the equivalent conditional tests from the **etoolbox**³⁰ package.

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:->114813249641507505482278393872551066259805517784186172883663478065826541894704737970419535798876630484358265060061503749531707793118627774829601 and `\meaning\B`: macro:->5493629452133983225138128786223912807341050049847605059532189961231327664902288388132878702444582075129603152041054804964625083138567652624386837205668069376.

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 $(-9) \times 357 - (-10) \times 323 = 17$ 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 n th element of the array, here the n th 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
```

³⁰<http://www.ctan.org/pkg/etoolbox>

```

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

Warning: macros `\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: `\xintiiSum{\xintiPow{2}{100}}=115`. Indeed, `\xintiiSum` is usually used on braced items as in

`\xintiiSum{{123}{-345}}{\xintFac{7}}{\xintiOpp{\xintRem{3347}{591}}}=4426` but in the previous example each digit of 2^{100} was treated as one item 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 `\xintTypesetEuclideanAlgorithm`:

```
\xintAssignArray\xintEuclideanAlgorithm {#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 `\xintTypesetEuclideanAlgorithm` 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` since 1.06, `\xintApplyUnbraced`, since 1.06b, `\xintloop` and `\xintiloop` since 1.09g.³¹

As an example the following code uses only expandable operations:

```
|2^{100}| (= \xintiPow {2}{100}) has \xintLen{\xintiPow {2}{100}} digits
and the sum of their squares is
```

```
\xintiiSum{\xintApply {\xintISqr}{\xintiPow {2}{100}}}.
```

These digits are, from the least to the most significant:

```
\xintListWithSep {, }{\xintRev{\xintiPow {2}{100}}}. The thirteenth most
significant digit is \xintNthElt{13}{\xintiPow {2}{100}}. The seventh
least significant one is \xintNthElt{7}{\xintRev{\xintiPow {2}{100}}}.
```

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

It would be 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.

Expandably computing primes is done in [subsection 25.10](#).

16 A new kind of for loop

As part of the [utilities](#) coming with the [xinttools](#) package, there is a new kind of for loop, `\xintFor`. Check it out ([subsection 25.16](#)).

17 A new kind of expandable loop

Also included in [xinttools](#), `\xintiloop` is an expandable loop giving access to an iteration index, without using count registers which would break expandability. Check it out ([subsection 25.13](#)).

18 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
```

³¹ All these utilities, as well as `\xintAssign`, `\xintAssignArray` and the `\xintFor` loops are now available from the [xinttools](#) package, independently of the big integers facilities of [xint](#).


```

\xintError:ArrayIndexBeyondLimit
\xintError:FactorialOfNegativeNumber
\xintError:FactorialOfTooBigNumber
\xintError:DivisionByZero
\xintError:NaN
\xintError:FractionRoundedToZero
\xintError:NotAnInteger
\xintError:ExponentTooBig
\xintError:TooBigDecimalShift
\xintError:TooBigDecimalsSplit
\xintError:RootOfNegative
\xintError:NoBezoutForZeros
\xintError:ignored
\xintError:removed
\xintError:inserted
\xintError:use_xintthe!
\xintError:bigtroubleahead
\xintError:unknownfunction

```

19 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`.³²
- 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 $\text{T}_{\text{E}}\text{X}$ 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`.

³²to the contrary, this is allowed inside an `\xintexpr`-ession.

20 Package namespace

Inner macros of `xinttools`, `xint`, `xintfrac`, `xintexpr`, `xintbinhex`, `xintgcd`, `xintseries`, and `xintcfrac` all begin either with `\XINT_` or with `\xint_`.³³ 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.

21 Loading and usage

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

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

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

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

This is for the loading of the packages.

For the input of numbers as macro arguments the minus sign must have its standard category code (“*other*”). Similarly the slash used for fractions must have its standard category

³³starting with release 1.06b the style files use for macro names a more modern underscore `_` rather than the `@` sign. A handful of private macros starting with `\XINT` do not have the underscore for technical reasons: `\XINTsetupcatcodes`, `\XINTdigits` and macros with names starting with `XINTinFloat` or `XINTinfloat`.

code. And the square brackets, if made use of in the input, also must be of category code *other*. The ‘e’ of the scientific notation must be of category code letter.

All of those requirements are relaxed for tokens parsed 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 ‘e’ may be uppercased: ‘E’.

The **xint** bundle packages presuppose that the usual `\space` and `\empty` macros are pre-defined, as is the case in Plain \TeX as well as in \LaTeX .

22 Installation

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

Else, either `"latex xint.dtx"` or `"pdflatex xint.dtx"` or `"tex xint.dtx"` will also generate the style files (and the `xint.ins` file). The `tex` run stops (contrarily to `latex/pdflatex`) with the complaint that it does not understand `\NeedsTeXFormat`, but the style files will already have been extracted by that time.

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

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

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

23 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 28](#)). So, a utility `\xintNewExpr` is provided to do the work of translating an `\xintexpr`-ession with parameters into a chain of macro evaluations.³⁴

```
\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`).

- 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 `\xint-`

³⁴As it makes some macro definitions, it is not an expandable command. It does not need protection against active characters as it does it itself.

`ifboolexpr` (subsection 28.9) and the discussion of the `bool` and `toggle` functions in section 23. Here is an example:

```
\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}
0 AND (0 OR 0) is 0    0 OR (0 AND 0) is 0    0 XOR 0 XOR 0 is 0
0 AND (0 OR 1) is 0    0 OR (0 AND 1) is 0    0 XOR 0 XOR 1 is 1
0 AND (1 OR 0) is 0    0 OR (1 AND 0) is 0    0 XOR 1 XOR 0 is 1
0 AND (1 OR 1) is 0    0 OR (1 AND 1) is 1    0 XOR 1 XOR 1 is 0
1 AND (0 OR 0) is 0    1 OR (0 AND 0) is 1    1 XOR 0 XOR 0 is 1
1 AND (0 OR 1) is 1    1 OR (0 AND 1) is 1    1 XOR 0 XOR 1 is 0
1 AND (1 OR 0) is 1    1 OR (1 AND 0) is 1    1 XOR 1 XOR 0 is 0
1 AND (1 OR 1) is 1    1 OR (1 AND 1) is 1    1 XOR 1 XOR 1 is 1
```

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

```
\xintthefloatexpr 2^1000000\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 2^1000000\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.

24 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 $-2^{-10} - 5 \cdot 3$ does $((-(2^{-10}))^{(-5))) \cdot 3$.

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,³⁵ 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. Next³⁶ are the postfix operators: `!` for the factorial, `?` and `:` are two-fold way and three-fold way branching constructs. Then the *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 $x \neq 0$, 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`, `toggle`.

`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

³⁵this 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.

³⁶in releases earlier than 1.09c, these postfix operators took precedence on a previous function name; the opposite now holds.

YES if not (the `if` conditional is described below; the `\xintifbool` test automatically encapsulates its first argument in an `\xintexpr` and follows the first branch if the result is non-zero (see [subsection 28.9](#))).

The alternative syntax `25*4-\ifmode100\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 L^AT_EX package `etoolbox`³⁷ 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{\ifsome-test}` 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 `|` logic operators: `\ifsometest10 & \ifsomeothertest10 | \ifsomeotherdtest10`, etc... YES or NO above stand for material compatible with the `\xintexpr` parser syntax.

See also `\xintifbool`, 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.

³⁷<http://www.ctan.org/pkg/etoolbox>

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.

25 Commands of the **xinttools** package

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

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

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

Contents

.1	<code>\xintReverseOrder</code>	31	<code>\xintbreakilooop</code> , <code>\xintbreak-</code>	
.2	<code>\xintRevWithBraces</code>	32	<code>ilooopanddo</code> , <code>\xintilooopskip-</code>	
.3	<code>\xintLength</code>	32	<code>tonext</code> , <code>\xintilooopskipandredo</code>	41
.4	<code>\xintZapFirstSpaces</code> , <code>\xintZapLastSpaces</code> , <code>\xintZapSpaces</code> , <code>\xintZapSpacesB</code>	32	.14 A table of factorizations	44
.5	<code>\xintCSVtoList</code>	33	.15 <code>\xintApplyInline</code>	46
.6	<code>\xintNthElt</code>	34	.16 <code>\xintFor</code> , <code>\xintFor*</code>	47
.7	<code>\xintListWithSep</code>	35	.17 <code>\xintifForFirst</code> , <code>\xintifFor-</code>	
.8	<code>\xintApply</code>	35	<code>Last</code>	50
.9	<code>\xintApplyUnbraced</code>	36	.18 <code>\xintBreakFor</code> , <code>\xintBreak-</code>	
.10	<code>\xintSeq</code>	36	<code>ForAndDo</code>	50
.11	Completely expandable prime test	37	.19 <code>\xintintegers</code> , <code>\xintdimen-</code>	
.12	<code>\xintloop</code> , <code>\xintbreakloop</code> , <code>\xintbreakloopanddo</code> , <code>\xint-</code> <code>loopskiptonext</code>	39	<code>sions</code> , <code>\xinrationals</code>	51
.13	<code>\xintilooop</code> , <code>\xintilooopin-</code> <code>dex</code> , <code>\xintouterilooopindex</code> ,		.20 Another table of primes	53
			.21 <code>\xintForpair</code> , <code>\xintForthree</code> ,	
			<code>\xintForfour</code>	54
			.22 <code>\xintAssign</code>	55
			.23 <code>\xintAssignArray</code>	55
			.24 <code>\xintRelaxArray</code>	55
			.25 The Quick Sort algorithm illustrated	55

25.1 `\xintReverseOrder`

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

`\xintReverseOrder{\xintDigitsOf\xintiPow {2}{100}\to\Stuff}`
gives: `\Stuff\to1002\xintiPow\xintDigitsOf`

25.2 `\xintRevWithBraces`

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

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

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

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

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

25.3 `\xintLength`

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

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

25.4 `\xintZapFirstSpaces`, `\xintZapLastSpaces`, `\xintZapSpaces`, `\xintZapSpacesB`

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

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

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

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

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

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

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

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

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

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

- n* ★ `\xintZapSpacesB{⟨stuff⟩}` does not do *any* expansion of its argument, nor does it alter `⟨stuff⟩` in anyway apart from stripping away all leading and all ending spaces and possibly removing one level of braces if `⟨stuff⟩` had the shape `<spaces>{braced}<spaces>`. The same remarks as for `\xintZapFirstSpaces` apply.

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

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

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

25.5 `\xintCSVtoList`

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

So `\xintCSVtoList` takes on input a ‘comma separated list of items’ and converts it into a ‘ \TeX list of braced items’. The argument to `\xintCSVtoList` may be a macro: it will first be *f-expanded*. Hence the item before the first comma, if it is itself a macro, will be expanded which may or may not be a good thing. A space inserted at the start of the first

- n* ★ item serves to stop that expansion (and disappears). The macro `\xintCSVtoListNoExpand` does the same job without the initial expansion of the list argument.

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

Contiguous spaces, tab characters, or other blanc spaces (empty lines not allowed) are collapsed by \TeX into single spaces. All such spaces around commas³⁸ are removed, as well as the spaces at the start and the spaces at the end of the list.³⁹

³⁸and multiple space tokens are not a problem; but those at the top level (not hidden inside braces) *must* be of character code 32.

³⁹let us recall that this is all done completely expandably... There is absolutely no alteration of any sort of the item apart from the stripping of initial and final space tokens (of character code 32) and brace removal if and only if the item apart from initial and final spaces (or more generally multiple char 32

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

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

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

From the rules above: for an empty argument (only spaces, no braces, no comma) the output is {} (a list with one empty item), for “<opt. spaces>{}<opt. spaces>” the output is {} (again a list with one empty item, the braces were removed), for “{ }” the output is {} (again a list with one empty item, the braces were removed and then the inner space was removed), for “ { }” the output is {} (again a list with one empty item, the initial space served only to stop the expansion, so this was like “{ }” as input, the braces were removed and the inner space was stripped), for “ { } ” the output is { } (this time the ending space of the first item meant that after brace removal the inner spaces were kept; recall though that T_EX collapses on input consecutive blanks into one space token), for “,” the output consists of two consecutive empty items {}-{}. Recall that on output everything is braced, a {} is an “empty” item. Most of the above is mainly irrelevant for every day use, apart perhaps from the fact to be noted that an empty input does not give an empty output but a one-empty-item list (it is as if an ending comma was always added at the end of the input).

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

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

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

Else, we may have direct use:

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

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

For the similar conversion from comma separated list to braced items list, but without removal of spaces around the commas, there is \xintCSVtoListNonStripped and

f ★ \xintCSVtoListNonStrippedNoExpand.

n ★

25.6 \xintNthElt

^{num}*x* *f* ★ \xintNthElt{x}{<list>} gets (expandably) the xth braced item of the <list>. An unbraced space tokens) is braced.

item token will be returned as is. The list itself may be a macro which is first *f*-expanded.

```
\xintNthElt {3}{\agh}\u{zzz}\v{Z}} is zzz
\xintNthElt {3}{\agh}\u{{zzz}}\v{Z}} is {zzz}
\xintNthElt {2}{\agh}\u{{zzz}}\v{Z}} is \u
\xintNthElt {37}{\xintFac {100}}=9 is the thirty-seventh digit of 100!.
\xintNthElt {10}{\xintFtoCv {566827/208524}}=1457/536
```

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

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

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

If $x<0$, the macro returns the x th element from the end of the list.

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

num
x *n* ★

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

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

25.7 **\xintListWithSep**

nf ★

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

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

nn ★

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

25.8 **\xintApply**

ff ★

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

Being expandable, **\xintApply** is useful for example inside alignments where implicit groups make standard loops constructs usually fail. In such situation it is often not wished

that the new list elements be braced, see `\xintApplyUnbraced`. The `\macro` does not have to be expandable: `\xintApply` will try to expand it, the expansion may remain partial.

The $\langle list \rangle$ 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 $\langle list \rangle$ expands to some positive number, then each digit will be replaced by the result of applying `\macro` on it.

```
\def\macro #1{\the\numexpr 9-#1\relax}
\xintApply\macro{\xintFac {20}}=7567097991823359999
```

fn ★ The macro `\xintApplyNoExpand` does the same job without the first initial expansion which gave the $\langle list \rangle$ of braced tokens to which `\macro` is applied.

25.9 `\xintApplyUnbraced`

ff ★ `\xintApplyUnbraced{\macro}{\langle list \rangle}` 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}{\langle list \rangle}}
```

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

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

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

25.10 `\xintSeq`

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

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

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

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

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

The produced integers are with explicit literal digits, so if used in `\ifnum` or other tests they should be properly terminated⁴⁰.

25.11 Completely expandable prime test

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

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

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

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

We used the **xint** provided expandable tests (on big integers or fractions) to maintain the complete expandability of `\IsPrime` in a strong sense⁴¹.

Our integers are short, but without `\expandafter`'s with `\@firstoftwo`, or some other related techniques, direct use of `\ifnum...fi` tests is dangerous. So to make the macro more efficient we are going to use the expandable tests provided by the package **etoolbox**⁴². The macro becomes:

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

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

⁴⁰a `\space` will stop the \TeX scanning of a number and be gobbled in the process, maintaining expandability if this is required; the `\relax` stops the scanning but is not gobbled and remains afterwards as a token.

⁴¹technically, prefixing it with `\romannumeral-‘0` must expand it completely; this is the case of all **xint** expandable macros, and in turn the arguments must be of this type.

⁴²<http://ctan.org/pkg/etoolbox>

```

\def\IsNotDivisibleBy #1#2%
  {\ifnum\numexpr #1-(#1/#2)*#2=0 \expandafter 0\else \expandafter1\fi}
where the \expandafter's are crucial for this macro to be completely expandable in the
restricted sense mentioned in footnote 41 which we want for applying confidently \xint-
ANDof. Anyhow, now that we have loaded etoolbox, we might as well use:
\newcommand{\IsNotDivisibleBy}[2]{\ifnumequal{#1-(#1/#2)*#2}{0}{0}{1}}
Let us enhance our prime macro to work also on the small primes:
\newcommand{\IsPrime}[1] % returns 1 if #1 is prime, and 0 if not
  {\ifnumodd {#1}
    {\ifnumless {#1}{8}
      {\ifnumequal{#1}{1}{0}{1}}% 3,5,7 are primes
      {\xintANDof
        {\xintApply
          { \IsNotDivisibleBy {#1}}{\xintSeq [2]{3}{\xintiSqrt{#1}}}}%
        }% END OF THE ODD BRANCH
      {\ifnumequal {#1}{2}{1}{0}}% END OF THE EVEN BRANCH
    }
  }

```

The input is still assumed positive. There is a deliberate blank before \IsNotDivisibleBy to use this feature of \xintApply: a space stops the expansion of the applied macro (and disappears). This expansion will be done by \xintANDof, which has been designed to skip everything as soon as it finds a false (i.e. zero) input. This way, the efficiency is considerably improved. We did generate via \xintSeq too many divisors though; if we really wanted to optimize even further it would be reasonable to drop the requirement of complete expandability and use the tools provided by the \xintFor loop.

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

```

\newcounter{primecount}
\newcounter{cellcount}
\newcommand{\NbOfColumns}{13}
\newcommand{\OneCell}[1]{%
  \ifnumequal{\IsPrime{#1}}{1}
    {\stepcounter{primecount}
      \ifnumequal{value{cellcount}}{\NbOfColumns}
        {\setcounter{cellcount}{1}#1}
        {\&\stepcounter{cellcount}#1}%
      } % was prime
    {}% not a prime, nothing to do
  }
\newcommand{\OneTab}[1]{&}
\begin{tabular}{|*{\NbOfColumns}{r}|}
\hline
2 \setcounter{cellcount}{1}\setcounter{primecount}{1}%
  \xintApplyUnbraced \OneCell {\xintSeq [2]{3}{999}}%
  \xintApplyUnbraced \OneTab

```

```

{\xintSeq [1]{1}{\the\numexpr\NbOfColumns-\value{cellcount}\relax}}%
\\
\hline
\end{tabular}

```

There are `\arabic{primecount}` prime numbers up to 1000.

We had to be careful to use the optional argument [1] to `\xintSeq` in this last row to not generate a decreasing sequence from 1 to 0, but an empty sequence when the row turns out to already have all its cells.

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

There are 168 prime numbers up to 1000.

25.12 `\xintloop`, `\xintbreakloop`, `\xintbreakloopanddo`, `\xintloopskiptonext`

- ★ `\xintloop<stuff>\if<test>...\repeat` is an expandable loop compatible with nesting. If a sub-loop is to be used all the material from the start and up to the complete subloop inclusive should be braced; these braces will be removed and do not create a group.

As this loop and `\xinttiloop` will primarily be of interest to experienced T_EX macro programmers, my description will assume that the user is knowledgeable enough. The loop and its various constituents are declared short, they can not handle `\par` tokens or empty lines.

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

One will usually employ some count registers to manage the exit test from the loop; this breaks expandability, see `\xinttiloop` for an expandable integer indexed loop. Use in alignments will be complicated by the fact that cells create groups, and also from the fact that any encountered unexpandable material will cause the T_EX input scanner to insert `\endtemplate` on each encountered `&` or `\cr`; thus `\xintbreakloop` may not work as expected, but the situation can be resolved via `\xint_firstofone{&}` or use of `\TAB`

with `\def\tab{&}`. It is thus simpler for alignments to use rather than `\xintloop` either the expandable `\xintApplyUnbraced` or the non-expandable but alignment compatible `\xintApplyInline`, `\xintFor` or `\xintFor*`.

As an example, let us suppose we have two macros $A_{\langle i \rangle \langle j \rangle}$ and $B_{\langle i \rangle \langle j \rangle}$ behaving like (small) integer valued matrix entries, and we want to define a macro $C_{\langle i \rangle \langle j \rangle}$ giving the matrix product (i and j may be count registers). We will assume that $A[I]$ expands to the number of rows, $A[J]$ to the number of columns and want the produced C to act in the same manner. The code is very dispendious in use of `\count` registers, not optimized in any way, not made very robust (the defined macro can not have the same name as the first two matrices for example), we just wanted to quickly illustrate use of the nesting capabilities of `\xintloop`.⁴³

```

\newcount\rowmax \newcount\colmax \newcount\summax
\newcount\rowindex \newcount\colindex \newcount\sumindex
\newcount\tmpcount
\makeatletter
\def\MatrixMultiplication #1#2#3{%
  \rowmax #1[I]\relax
  \colmax #2[J]\relax
  \summax #1[J]\relax
  \rowindex 1
  \xintloop % loop over row index i
  {\colindex 1
    \xintloop % loop over col index k
    {\tmpcount 0
      \sumindex 1
      \xintloop % loop over intermediate index j
      \advance\tmpcount \numexpr #1\rowindex\sumindex*#2\sumindex\colindex\relax
      \ifnum\sumindex<\summax
        \advance\sumindex 1
      \repeat }%
      \expandafter\edef\csname\string#3{\the\rowindex.\the\colindex}\endcsname
      {\the\tmpcount}%
      \ifnum\colindex<\colmax
        \advance\colindex 1
      \repeat }%
      \ifnum\rowindex<\rowmax
        \advance\rowindex 1
      \repeat
      \expandafter\edef\csname\string#3{I}\endcsname{\the\rowmax}%
      \expandafter\edef\csname\string#3{J}\endcsname{\the\colmax}%
      \def #3##1{\ifx[##1\expandafter\Matrix@helper@size
        \else\expandafter\Matrix@helper@entry\fi #3{##1}}%
    }%
  }%
\def\Matrix@helper@size #1#2#3{\csname\string#1{#3}\endcsname }%
\def\Matrix@helper@entry #1#2#3%
  {\csname\string#1{\the\numexpr#2.\the\numexpr#3}\endcsname }%
\def\A #1{\ifx[#1\expandafter\A@size

```

⁴³for a more sophisticated implementation of matrix multiplication, inclusive of determinants, inverses, and display utilities, with entries big integers or decimal numbers or even fractions see <http://tex.stackexchange.com/a/143035/4686> from November 11, 2013.

```

\else\expandafter\A@entry\fi {#1}}%
\def\A@size #1#2{\ifx I#23\else4\fi}% 3rows, 4columns
\def\A@entry #1#2{\the\numexpr #1+#2-1\relax}% not pre-computed...
\def\B #1{\ifx[#1\expandafter\B@size
\else\expandafter\B@entry\fi {#1}}%
\def\B@size #1#2{\ifx I#24\else3\fi}% 4rows, 3columns
\def\B@entry #1#2{\the\numexpr #1-#2\relax}% not pre-computed...
\makeatother
\MatrixMultiplication\A\B\C \MatrixMultiplication\C\C\D % etc...
\[\begin{pmatrix}
\A11&\A12&\A13&\A14\\
\A21&\A22&\A23&\A24\\
\A31&\A32&\A33&\A34
\end{pmatrix}
\times
\begin{pmatrix}
\B11&\B12&\B13\\
\B21&\B22&\B23\\
\B31&\B32&\B33\\
\B41&\B42&\B43
\end{pmatrix}
=
\begin{pmatrix}
\C11&\C12&\C13\\
\C21&\C22&\C23\\
\C31&\C32&\C33
\end{pmatrix}
\]
\[\begin{pmatrix}
\A11&\A12&\A13&\A14\\
\A21&\A22&\A23&\A24\\
\A31&\A32&\A33&\A34
\end{pmatrix}
\times
\begin{pmatrix}
0 & -1 & -2 \\
1 & 0 & -1 \\
2 & 1 & 0 \\
3 & 2 & 1
\end{pmatrix}
=
\begin{pmatrix}
20 & 10 & 0 \\
26 & 12 & -2 \\
32 & 14 & -4
\end{pmatrix}
\]
\[\begin{pmatrix}
20 & 10 & 0 \\
26 & 12 & -2 \\
32 & 14 & -4
\end{pmatrix}^2
=
\begin{pmatrix}
660 & 320 & -20 \\
768 & 376 & -16 \\
876 & 432 & -12
\end{pmatrix}
\]
\[\begin{pmatrix}
20 & 10 & 0 \\
26 & 12 & -2 \\
32 & 14 & -4
\end{pmatrix}^3
=
\begin{pmatrix}
20880 & 10160 & -560 \\
24624 & 11968 & -688 \\
28368 & 13776 & -816
\end{pmatrix}
\]
\[\begin{pmatrix}
20 & 10 & 0 \\
26 & 12 & -2 \\
32 & 14 & -4
\end{pmatrix}^4
=
\begin{pmatrix}
663840 & 322880 & -18080 \\
781632 & 380224 & -21184 \\
899424 & 437568 & -24288
\end{pmatrix}
\]
\[\begin{pmatrix}
\A11&\A12&\A13\\
\A21&\A22&\A23\\
\A31&\A32&\A33
\end{pmatrix}^2
=
\begin{pmatrix}
\D11&\D12&\D13\\
\D21&\D22&\D23\\
\D31&\D32&\D33
\end{pmatrix}
\]

```

25.13 **\xintloop**, **\xintloopindex**, **\xintouterloopindex**, **\xintbreakilooop**, **\xintbreakilooopanddo**, **\xintloopskipnext**, **\xintloopskipandredo**

- ★ **\xintloop**[start+delta]<stuff>\if<test> ... **\repeat** is a completely expandable nestable loop having access via **\xintloopindex** to the integer index of the iteration, with starting value start (which may be a **\count**) and increment delta (*id.*). Currently [start+delta] is a *mandatory argument*, it is an error to omit it; perhaps a future release will make it optional with default 1+1. A space after the closing square bracket is not significant, it will be ignored. Spaces inside the square brackets will also be ignored as the two arguments are first given to a **\numexpr**...**\relax**. Empty lines and explicit **\par** tokens are accepted.

As with **\xintloop**, this tool will mostly be of interest to advanced users. For nesting, one puts inside braces all the material from the start (immediately after [start+delta])

and up to and inclusive of the inner loop, these braces will be removed and do not create a loop. In case of nesting, `\xintouterloopindex` gives access to the index of the outer loop. If needed one could write on its model a macro giving access to the index of the outer loop (or even to the *n*th outer loop).

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

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

One has `\xintbreakiloop` and `\xintbreakiloopanddo` to abort the loop, `\xintloopskiptonext` to abort the current iteration and skip to the next, `\xintloopskipandredo` to skip to the end of the current iteration and redo it with the same value of the index (something else will have to change for this not to become an eternal loop...).

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

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

```
\edef\z
{\xintloop [10001+2]%
  {\xintloop [3+2]%
    \ifnum\xintouterloopindex<\numexpr\xintloopindex*\xintloopindex\relax
      \xintouterloopindex,
      \expandafter\xintbreakiloop
    \fi
    \ifnum\xintouterloopindex=\numexpr
      (\xintouterloopindex/\xintloopindex)*\xintloopindex\relax
    \else
      \repeat
    }% no space here
  \ifnum \xintloopindex < 10999 \repeat }%
\meaning\z macro:->10007, 10009, 10037, 10039, 10061, 10067, 10069, 10079, 10091,
10093, 10099, 10103, 10111, 10133, 10139, 10141, 10151, 10159, 10163, 10169, 10177,
10181, 10193, 10211, 10223, 10243, 10247, 10253, 10259, 10267, 10271, 10273, 10289,
10301, 10303, 10313, 10321, 10331, 10333, 10337, 10343, 10357, 10369, 10391, 10399,
10427, 10429, 10433, 10453, 10457, 10459, 10463, 10477, 10487, 10499, 10501, 10513,
10529, 10531, 10559, 10567, 10589, 10597, 10601, 10607, 10613, 10627, 10631, 10639,
10651, 10657, 10663, 10667, 10687, 10691, 10709, 10711, 10723, 10729, 10733, 10739,
10753, 10771, 10781, 10789, 10799, 10831, 10837, 10847, 10853, 10859, 10861, 10867,
10883, 10889, 10891, 10903, 10909, 10937, 10939, 10949, 10957, 10973, 10979, 10987,
```


10993, and we should have taken some steps to not have a trailing comma, but the point was to show that one can do that in an `\edef`!

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

```
\tabskiplex
\halign{&\hfil#\hfil\cr
  \xintilooop [1+1]
  {\expandafter\bfseries\xintilooopindex &
  \xintilooop [1+1]
  \ifnum\xintouterilooopindex=\numexpr
    (\xintouterilooopindex/\xintilooopindex)*\xintilooopindex\relax
  \xintilooopindex&\fi
  \ifnum\xintilooopindex<\xintouterilooopindex\space % \space is CRUCIAL
  \repeat \cr }%
  \ifnum\xintilooopindex<30
  \repeat }
```

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

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

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

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

25.14 A table of factorizations

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

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

```
\tabskiplex
\halign {\hfil\strut#\hfil&\hfil#\hfil\cr\noalign{\hrule}
  \xintloop ["7FFFFFFE0+1]
  \expandafter\bf\xintloopindex &
  \ifnum\xintloopindex="7FFFFFFD
    \number"7FFFFFFD\cr\noalign{\hrule}
    \expandafter\xintloopskiptonext
  \fi
  \expandafter\factorize\xintloopindex.\cr\noalign{\hrule}
  \ifnum\xintloopindex<"7FFFFFFE
  \repeat
  \bf \number"7FFFFFFF&\number "7FFFFFFF\cr\noalign{\hrule}
}
```

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

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

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

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

\def\factorize_b #1.#2.{\ifnum#1=1 \abortfactorize\fi
  % this will avoid overflow which could result from #2*#2
  \ifnum\numexpr #1-(#2-1)*#2<#2
    #1\abortfactorize % this #1 is prime
  \fi}
```

25 Commands of the *xinttools* package

```
% again, avoiding overflow as \numexpr integer division
% rounds rather than truncates.
\ifnum \numexpr #1-#2=\numexpr ((#1/#2)-1)*#2\relax
  \expandafter\xint_firstoftwo
\else\expandafter\xint_secondoftwo
\fi
{#2&\expandafter\factorize_b\the\numexpr#1/#2.#2.}%
{\expandafter\factorize_b\the\numexpr #1\expandafter.%
  \the\numexpr #2+2.}}%
```

\catcode'_ 8

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

A table of factorizations

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

25.15 `\xintApplyInline`

*o *f* `\xintApplyInline{\macro}{\langle list \rangle}` 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 the next page):

```
\def\Row #1{#1:\xintApplyInline {\Item {#1}}{0123456789}}%
```

```
\def\Item #1#2{&\xintiPow {#1}{#2}}%
\begin{tabular}{cccccccccc}
&&&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}{cccccccccc}
&&&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](#).

25.16 `\xintFor`, `\xintFor*`

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

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

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

A macro `\macro` whose definition uses internally an `\xintFor` loop may be used inside another `\xintFor` loop even if the two loops both use the same macro parameter. Note: the loop definition inside `\macro` must double the character `#` as is the general rule in \TeX with definitions done inside macros.

The macros `\xintFor` and `\xintFor*` are not expandable, one can not use them inside an `\edef`. But they may be used inside alignments (such as a \LaTeX `tabular`), as will be shown in examples.

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

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

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

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

The starred variant `\xintFor*` deals with token lists (*spaces between braced items or single tokens are not significant*) and *f-expands* each *unbraced* list item. This makes it easy to simulate concatenation of various list macros `\x, \y, ...` If `\x` expands to `{1}{2}{3}` and `\y` expands to `{4}{5}{6}` then `{\x\y}` as argument to `\xintFor*` has the same effect as `{{1}{2}{3}}{4}{5}{6}`⁴⁴. Spaces at the start, end, or in-between items are gobbled (but naturally not the spaces which may be inside *braced* items). Except if

the list argument is a single macro with no parameters, it must be braced. Each item which is not braced will be fully expanded (as the `\x` and `\y` in the example above). An empty list leads to an empty result.

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

`\xintFor* #1 in {\xintSeq [+2]{-7}{+2}}\do {stuff with #1}` will have `#1=-7, -5, -3, -1, and 1`. The `#1` as issued from the list produced by `\xintSeq` is the literal representation as would be produced by `\arabic` on a \LaTeX counter, it is not a count register. When used in `\ifnum` tests or other contexts where \TeX looks for a number it should thus be postfixed with `\relax` or `\space`.

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

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

This is a general remark applying for any nesting of loops, one should avoid recreating the inner lists of arguments at each iteration of the outer loop. However, in the example above, if the `.. some other macros ..` part closes a group which was opened before the `\edef\innersequence`, then this definition will be lost. An alternative to `\edef`, also efficient, exists when dealing with arithmetic sequences: it is to use the `\xintintegers` keyword (described later) which simulates infinite arithmetic sequences; the loops will then be terminated via a test `#1` (or `#2` etc...) and subsequent use of `\xintBreakFor`.

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

```
A:  (a → A)  (b → A)  (c → A)  (d → A)  (e → A)
B:  (a → B)  (b → B)  (c → B)  (d → B)  (e → B)
C:  (a → C)  (b → C)  (c → C)  (d → C)  (e → C)

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

When inserted inside a macro for later execution the `#` characters must be doubled.⁴⁴ 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)} }%
  }%
}%
```

⁴³braces around single token items are optional so this is the same as `{123456}`.

⁴⁴sometimes what seems to be a macro argument isn't really; in `\raisebox{1cm}{\xintFor #1 in {a,b,c}\do {#1}}` no doubling should be done.

```
\T\def\sep {\def\sep{, }}\z
      (u,x), (u,y), (u,z), (v,x), (v,y), (v,z), (w,x), (w,y), (w,z)
```

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

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

The iterated commands as well as the list items are allowed to contain explicit `\par` tokens. Neither `\xintFor` nor `\xintFor*` create groups. The effect is like piling up the iterated commands with each time `#1` (or `#2` ...) replaced by an item of the list. However, contrarily to the completely expandable `\xintApplyUnbraced`, but similarly to the non completely expandable `\xintApplyInline` each iteration is executed first before looking at the next `#1`⁴⁵ (and the starred variant `\xintFor*` keeps on expanding each unbraced item it finds, gobbling spaces).

25.17 `\xintifForFirst`, `\xintifForLast`

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

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

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

25.18 `\xintBreakFor`, `\xintBreakForAndDo`

One may immediately terminate an `\xintFor` or `\xintFor*` loop with `\xintBreakFor`. As the criterion for breaking will be decided on a basis of some test, it is recommended to use for this test the syntax of `ifthen`⁴⁶ or `etoolbox`⁴⁷ or the **xint** own conditionals, rather than one of the various `\if... \fi` of \TeX . Else (and this is without even mentioning all the various peculiarities of the `\if... \fi` constructs), one has to carefully move the break after the closing of the conditional, typically with `\expandafter\xintBreakFor\fi`.⁴⁸

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

⁴⁵to be completely honest, both `\xintFor` and `\xintFor*` initially scoop up both the list and the iterated commands; `\xintFor` scoops up a second time the entire comma separated list in order to feed it to `\xintCSVtoList`. The starred variant `\xintFor*` which does not need this step will thus be a bit faster on equivalent inputs.

⁴⁶<http://ctan.org/pkg/ifthen>

⁴⁷<http://ctan.org/pkg/etoolbox>

⁴⁸the difficulties here are similar to those mentioned in [section 13](#), although less severe, as complete expandability is not to be maintained; hence the allowed use of `ifthen`.

25.19 `\xintintegers`, `\xintdimensions`, `\xintrationals`

If the list argument to `\xintFor` (or `\xintFor*`, both are equivalent in this context) is `\xintintegers` (equivalently `\xintegers`) or more generally `\xintintegers[start+delta]` (*the whole within braces!*)⁴⁹, then `\xintFor` does an infinite iteration where #1 (or #2, ..., #9) will run through the arithmetic sequence of (short) integers with initial value `start` and increment `delta` (default values: `start=1`, `delta=1`; if the optional argument is present it must contain both of them, and they may be explicit integers, or macros or count registers). The #1 (or #2, ..., #9) will stand for `\numexpr <opt sign><digits>\relax`, and the literal representation as a string of digits can thus be obtained as `\the#1` or `\number#1`. Such a #1 can be used in an `\ifnum` test with no need to be postfixed with a space or a `\relax` and one should *not* add them.

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



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

The **graphic**, with the code on its right⁵⁰, is for illustration only, not only because of pdf rendering artefacts when displaying adjacent rules (which do *not* show in dvi output as rendered by xdv, and depend from your viewer), but because not using anything but rules it is quite inefficient and must do lots of computations to not confer a too ragged look to the borders. With a width of .5pt rather than .1pt for the rules, one speeds up the drawing by a factor of five, but the boundary is then visibly ragged.⁵¹

⁴⁹the `start+delta` optional specification may have extra spaces around the plus sign of near the square brackets, such spaces are removed. The same applies with `\xintdimensions` and `\xintrationals`.

⁵⁰the somewhat peculiar use of `_` and `$` is explained in [subsection 28.6](#); they are made necessary from the fact that the parameters are passed to a *macro* (`\DimToNum`) and not only to *functions*, as are known to `\xintexpr`. But one can also define directly the desired function, for example the constructed `\FA` turns out to have meaning `macro:#1#2->\romannumeral -'0\xintiRound 0{\xintDiv {\xintPow {\DimToNum {#2}}{3}}{\xintPow {\DimToNum {#1}}{2}}}`, where the `\romannumeral` part is only to ensure it expands in only two steps, and could be removed. A handwritten macro would use here `\xintiPow` and not `\xintPow`, as we know it has to deal with integers only. See the next footnote.

⁵¹to tell the whole truth we cheated and divided by 10 the computation time through using the following

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

```
\xintFor #1 in {\xinrationals [10/21+1/21]} \do
{#1=\xintifInt {#1}
  {\textcolor{blue}{\xintTrunc{10}{#1}}}
  {\xintTrunc{10}{#1}}% in blue if an integer
  \xintifGt {#1}{1.123}{\xintBreakFor}{, }%
}
10/21=0.4761904761, 11/21=0.5238095238, 12/21=0.5714285714, 13/21=0.6190476190,
14/21=0.6666666666, 15/21=0.7142857142, 16/21=0.7619047619, 17/21=0.8095238095,
18/21=0.8571428571, 19/21=0.9047619047, 20/21=0.9523809523, 21/21=1.0000000000,
22/21=1.0476190476, 23/21=1.0952380952, 24/21=1.1428571428
```

The example above confirms that computations are done exactly, and illustrates that the two initial (reduced) denominators are not multiplied when they are found to be equal. It is thus recommended to input `start` and `delta` with a common smallest possible denominator, or as fixed point numbers with the same numbers of digits after the decimal mark; and this is also the reason why `start` and `delta` are not by default made irreducible. As internally the computations are done with numerators and denominators completely expanded, one should be careful not to input numbers in scientific notation with exponents in the hundreds, as they will get converted into as many zeros.

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

definitions, together with a horizontal step of .25pt rather than .1pt. The displayed original code would make the slowest computation of all those done in this document using the **xint** bundle macros!

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

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

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

25.20 Another table of primes

As a further example, let us dynamically generate a tabular with the first 50 prime numbers after 12345. First we need a macro to test if a (short) number is prime. Such a completely expandable macro was given in [subsection 25.10](#), here we consider a variant which will be slightly more efficient. This new `\IsPrime` has two parameters. The first one is a macro which it redefines to expand to the result of the primality test applied to the second argument. For convenience we use the `etoolbox` wrappers to various `\ifnum` tests, although here there isn't anymore the constraint of complete expandability (but using explicit `\if... \fi` in tabulars has its quirks); equivalent tests are provided by **xint**, but they have some overhead as they are able to deal with arbitrarily big integers.

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

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

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

```
\newcounter{primecount}
\newcounter{cellcount}
\begin{figure*}[ht!]
```

```

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

```

25.21 **\xintForpair**, **\xintForthree**, **\xintForfour**

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

```

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

```

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

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

25.22 \xintAssign

$(f \rightarrow *x) *N$ `\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`. It is not expandable.

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

xN 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{10000000000000}{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}})
```

25.23 \xintAssignArray

$(f \rightarrow *x) N$ `\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.

25.24 \xintRelaxArray

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

25.25 The Quick Sort algorithm illustrated

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

If the interest is only in \TeX integers, then one should replace the macros `\QSMORE`,

QSEqual, QSLess with versions using the *etoolbox* (L^AT_EX only) \ifnumgreater, \ifnumequal and \ifnumless conditionals rather than \xintifGt, \xintifEq, \xintifLt.

```
% THE QUICK SORT ALGORITHM EXPANDABLY
\input xintfrac.sty
% HELPER COMPARISON MACROS
\def\QSMORE #1#2{\xintifGt {#2}{#1}{{#2}}{ }}
% the spaces are there to stop the \romannumeral-'\0 originating
% in \xintapplyunbraced when it applies a macro to an item
\def\QSEqual #1#2{\xintifEq {#2}{#1}{{#2}}{ }}
\def\QSLess #1#2{\xintifLt {#2}{#1}{{#2}}{ }}
%
\makeatletter
\def\QSfull { \romannumeral0\qsfull }
\def\qsfull #1{\expandafter\qsfull@a\expandafter{\romannumeral-'\0#1}}
\def\qsfull@a #1{\expandafter\qsfull@b\expandafter {\xintLength {#1}}{#1}}
\def\qsfull@b #1{\ifcase #1
    \expandafter\qsfull@empty
  \or\expandafter\qsfull@single
  \else\expandafter\qsfull@c
  \fi
}%
\def\qsfull@empty #1{ } % the space stops the \QSfull \romannumeral0
\def\qsfull@single #1{ #1}
% for simplicity of implementation, we pick up the first item as pivot
\def\qsfull@c #1{\qsfull@ci #1\undef {#1}}
\def\qsfull@ci #1#2\undef {\qsfull@d {#1}}% #3 is the list, #1 its first item
\def\qsfull@d #1#2{\expandafter\qsfull@e\expandafter
    {\romannumeral0\qsfull
    {\xintApplyUnbraced {\QSMORE {#1}}{#2}}}%
    {\romannumeral0\xintapplyunbraced {\QSEqual {#1}}{#2}}}%
    {\romannumeral0\qsfull
    {\xintApplyUnbraced {\QSLess {#1}}{#2}}}%
}%
\def\qsfull@e #1#2#3{\expandafter\qsfull@f\expandafter {#2}{#3}{#1}}%
\def\qsfull@f #1#2#3{\expandafter\space #2#1#3}
\makeatother
% EXAMPLE
\edef\z {\QSfull {{1.0}{0.5}{0.3}{1.5}{1.8}{2.0}{1.7}{0.4}{1.2}{1.4}%
    {1.3}{1.1}{0.7}{1.6}{0.6}{0.9}{0.8}{0.2}{0.1}{1.9}}}
\tt\meaning\z
\def\A {3.123456789123456789}\def\B {3.123456789123456788}
\def\C {3.123456789123456790}\def\D {3.123456789123456787}
\expandafter\def\expandafter\z\expandafter
    {\romannumeral0\qsfull {{\A}{\B}{\C}{\D}}}% \A is braced to not be expanded
\meaning\z
```

Output:

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

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

```
\input xintfrac.sty % if Plain TeX
%
\definecolor{LEFT}{RGB}{216,195,88}
\definecolor{RIGHT}{RGB}{208,231,153}
\definecolor{INERT}{RGB}{199,200,194}
\definecolor{PIVOT}{RGB}{109,8,57}
%
\def\QSMORE #1#2{\xintifGt {#2}{#1}{#{#2}}{ }}% space will be gobbled
\def\QSEQUAL #1#2{\xintifEq {#2}{#1}{#{#2}}{ }}
\def\QSLess #1#2{\xintifLt {#2}{#1}{#{#2}}{ }}
%
\makeatletter
\def\QS@a #1{\expandafter \QS@b \expandafter {\xintLength {#1}}{#1}}
\def\QS@b #1{\ifcase #1
    \expandafter\QS@empty
  \or\expandafter\QS@single
  \else\expandafter\QS@c
  \fi
}%
\def\QS@empty #1{}
\def\QS@single #1{\QSIr {#1}}
\def\QS@c #1{\QS@d #1!{#1}} % we pick up the first as pivot.
\def\QS@d #1#2!{\QS@e {#1}}% #1 = first element, #3 = list
\def\QS@e #1#2{\expandafter\QS@f\expandafter
    {\romannumeral0\xintapplyunbraced {\QSMORE {#1}}{#2}}%
    {\romannumeral0\xintapplyunbraced {\QSEQUAL {#1}}{#2}}%
    {\romannumeral0\xintapplyunbraced {\QSLess {#1}}{#2}}%
}%
\def\QS@f #1#2#3{\expandafter\QS@g\expandafter {#2}{#3}{#1}}%
% Here \QSLr, \QSIr, \QSR have been let to \relax, so expansion stops.
% #2= elements < pivot, #1 = elements = pivot, #3 = elements > pivot
\def\QS@g #1#2#3{\QSLr {#2}\QSIr {#1}\QSRr {#3}}%
%
\def\DecoLEFT #1{\xintFor* ##1 in {#1} \do {\colorbox{LEFT}{##1}}}
\def\DecoINERT #1{\xintFor* ##1 in {#1} \do {\colorbox{INERT}{##1}}}
\def\DecoRIGHT #1{\xintFor* ##1 in {#1} \do {\colorbox{RIGHT}{##1}}}
\def\DecoPivot #1{\begingroup\color{PIVOT}\advance\fbboxsep-\fbboxrule
    \fbbox{#1}\endgroup}
\def\DecoLEFTwithPivot #1{%
    \xintFor* ##1 in {#1} \do
    {\xintifForFirst {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}%
}
\def\DecoRIGHTwithPivot #1{%
    \xintFor* ##1 in {#1} \do
    {\xintifForFirst {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}%
}
%
\def\QSinitialize #1{\def\QS@list{\QSRr {#1}}%
    \let\QSRr\DecoRIGHT
```

```

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

```

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

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

```

\def\QS@c #1{\expandafter\QS@e\expandafter
\romannumeral0\xintnthelt {-1}{#1}}{#1}%
}%
\def\DecoLEFTwithPivot #1{%

```

```

\mintFor* ##1 in {#1} \do
{\mintifForLast {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}%
}
\def\DecoRIGHTwithPivot #1{%
\mintFor* ##1 in {#1} \do
{\mintifForLast {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}%
}
\def\QsInitialize #1{\def\Qs@list{\QSLr {#1}}%
\let\QSLr\DecoLEFT
% \Qs@list \par
\par\centerline{\Qs@list}
}

```

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

It is possible to modify this code to let it do `\Qsonestep` repeatedly and stop automatically when the sort is finished.⁵²

26 Commands of the **xint** package

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

⁵²<http://tex.stackexchange.com/a/142634/4686>

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

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

Some of these macros are extended by **xintfrac** to accept fractions on input, and, generally, to output a fraction. But this means that additions, subtractions, multiplications output in fraction format; to guarantee the integer format on output when the inputs are integers, the original integer-only macros `\xintAdd`, `\xintSub`, `\xintMul`, etc... are available under the names `\xintiAdd`, `\xintiSub`, `\xintiMul`, ..., also when **xintfrac** is not loaded. Even these originally integer-only macros will accept fractions on input if **xintfrac** is loaded as long as they are integers in disguise; they produce on output integers without any forward slash mark nor trailing `[n]`.

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

Contents

.1	\xintRev	61	.27	\xintMin	64
.2	\xintLen	61	.28	\xintMinof	64
.3	\xintDigitsOf	61	.29	\xintSum	64
.4	\xintNum	61	.30	\xintMul	64
.5	\xintSgn	62	.31	\xintSqr	65
.6	\xintOpp	62	.32	\xintPrd	65
.7	\xintAbs	62	.33	\xintPow	65
.8	\xintAdd	62	.34	\xintSgnFork	66
.9	\xintSub	62	.35	\xintifSgn	66
.10	\xintCmp	62	.36	\xintifZero	66
.11	\xintEq	62	.37	\xintifNotZero	66
.12	\xintGt	62	.38	\xintifTrueFalse	66
.13	\xintLt	62	.39	\xintifCmp	66
.14	\xintIsZero	62	.40	\xintifEq	66
.15	\xintNot	63	.41	\xintifGt	66
.16	\xintIsNotZero	63	.42	\xintifLt	67
.17	\xintIsOne	63	.43	\xintifOdd	67
.18	\xintAND	63	.44	\xintFac	67
.19	\xintOR	63	.45	\xintDivision	67
.20	\xintXOR	63	.46	\xintQuo	67
.21	\xintANDof	63	.47	\xintRem	67
.22	\xintORof	63	.48	\xintFDg	68
.23	\xintXORof	63	.49	\xintLDg	68
.24	\xintGeq	63	.50	\xintMON , \xintMMON	68
.25	\xintMax	64	.51	\xintOdd	68
.26	\xintMaxof	64	.52	\xintiSqrt , \xintiSquareRoot	68

.53	<code>\xintInc, \xintDec</code>	68	.58	<code>\xintDSHr, \xintDSx</code>	69
.54	<code>\xintDouble, \xintHalf</code>	69	.59	<code>\xintDecSplit</code>	70
.55	<code>\xintDSL</code>	69	.60	<code>\xintDecSplitL</code>	70
.56	<code>\xintDSR</code>	69	.61	<code>\xintDecSplitR</code>	70
.57	<code>\xintDSH</code>	69			

26.1 `\xintRev`

f ★ `\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
```

26.2 `\xintLen`

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

```
\xintLen{-12345678901234567890123456789}=29
```

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

```
\xintLen{-1e3/5.425}=10
```

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

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

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

26.3 `\xintDigitsOf`

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

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

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

26.4 `\xintNum`

f ★ `\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
```

26.5 \xintSgn

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

26.6 \xintOpp

- $\overset{\text{Num}}{f} \star$ `\xintOpp{N}` return the opposite $-N$ of the number N . Extended by **xintfrac** to fractions.
 $f \star$ `\xintiOpp` is a synonym not modified by **xintfrac**⁵³, and `\xintiiOpp` skips the `\xint-`
`Num` overhead.

26.7 \xintAbs

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

26.8 \xintAdd

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

26.9 \xintSub

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

26.10 \xintCmp

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

26.11 \xintEq

- $\overset{\text{Num}}{f} \overset{\text{Num}}{f} \star$ `\xintEq{N}{M}` returns 1 if $N=M$, 0 otherwise. Extended by **xintfrac** to fractions.

26.12 \xintGt

- $\overset{\text{Num}}{f} \overset{\text{Num}}{f} \star$ `\xintGt{N}{M}` returns 1 if $N>M$, 0 otherwise. Extended by **xintfrac** to fractions.

26.13 \xintLt

- $\overset{\text{Num}}{f} \overset{\text{Num}}{f} \star$ `\xintLt{N}{M}` returns 1 if $N<M$, 0 otherwise. Extended by **xintfrac** to fractions.

26.14 \xintIsZero

- $\overset{\text{Num}}{f} \star$ `\xintIsZero{N}` returns 1 if $N=0$, 0 otherwise. Extended by **xintfrac** to fractions.

⁵³here, and in all similar instances, this means that the macro remains integer-only both on input and output, but it does accept on input a fraction which in disguise is a (big) integer.

26.15 **\xintNot**

$\overset{\text{Num}}{f} \star$ **\xintNot** is a synonym for **\xintIsZero**.

26.16 **\xintIsNotZero**

$\overset{\text{Num}}{f} \star$ **\xintIsNotZero**{N} returns 1 if $N \neq 0$, 0 otherwise. Extended by **xintfrac** to fractions.

26.17 **\xintIsOne**

$\overset{\text{Num}}{f} \star$ **\xintIsOne**{N} returns 1 if $N=1$, 0 otherwise. Extended by **xintfrac** to fractions.

26.18 **\xintAND**

$\overset{\text{Num}}{f} \overset{\text{Num}}{f} \star$ **\xintAND**{N}{M} returns 1 if $N \neq 0$ and $M \neq 0$ and zero otherwise. Extended by **xintfrac** to fractions.

26.19 **\xintOR**

$\overset{\text{Num}}{f} \overset{\text{Num}}{f} \star$ **\xintOR**{N}{M} returns 1 if $N \neq 0$ or $M \neq 0$ and zero otherwise. Extended by **xintfrac** to fractions.

26.20 **\xintXOR**

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

26.21 **\xintANDof**

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

26.22 **\xintORof**

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

26.23 **\xintXORof**

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

26.24 **\xintGeq**

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

26.25 \xintMax

$$\begin{array}{c} \text{Num Num} \\ f \quad f \end{array} \star$$

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

26.26 \xintMaxof

$$f \rightarrow * \begin{array}{c} \text{Num} \\ f \end{array} \star$$

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

26.27 \xintMin

$$\begin{array}{c} \text{Num Num} \\ f \quad f \end{array} \star$$

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

26.28 \xintMinof

$$f \rightarrow * \begin{array}{c} \text{Num} \\ f \end{array} \star$$

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

26.29 \xintSum

$$*f \star$$

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

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

```
\xintiiSum{{123}{-98763450}}{\xintFac{7}}{\xintiMul{3347}{591}}=-96780210
\xintiiSum{1234567890}=45
```

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

26.30 \xintMul

$$\begin{array}{c} \text{Num Num} \\ f \quad f \end{array} \star$$

`\xintMul{N}{M}` returns the product of the two numbers. Extended by **xintfrac** to fractions. `\xintiMul` is a synonym not modified by **xintfrac**, and `\xintiiMul` skips the

$$ff \star$$

`\xintNum` overhead.

26.31 \xintSqr

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

26.32 \xintPrd

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

```
\xintiiPrd{\{-9876\}\{\xintFac{7}\}\{\xintiMul{3347}{591}\}}=-98458861798080
\xintiiPrd{123456789123456789}=131681894400
```

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

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

```
=\xintiiPrd {\{\xintiPow {2}{200}\}\{\xintiPow {3}{100}\}\{\xintiPow {7}{100}\}}
=26787279316615775757662795170075484023247402663740153489744596148
154264129654994900004440072407657271300001653120764065456211801435
71994015903343539244028212438966822248927862988084382716133376 With
xintexpr, the above would be coded simply as
```

```
\xintthenumexpr 2^200*3^100*7^100\relax
```

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

26.33 \xintPow

$\overset{\text{Num}}{f}$ $\overset{\text{num}}{x}$ ★ `\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 unrealistic. Already 2^{9999} has 3,010 digits,⁵⁴ so I should perhaps lower the bound to 99999.

Extended by **xintfrac** to fractions (`\xintPow`) and to floats (`\xintFloatPow`). 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`.

$\overset{f}{x}^{\text{num}}$ ★ `\xintiPow` is a synonym not modified by **xintfrac**, and `\xintiiPow` is an integer only variant skipping the `\xintNum` overhead.

⁵⁴on 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 \LaTeX `l3fp` does this with log/exp and is ten times faster (16 figures only).

26.34 \xintSgnFork

xnnn ★ `\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 `\num-expr...` 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.

26.35 \xintifSgn

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

26.36 \xintifZero

Num f nn ★ `\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.

26.37 \xintifNotZero

Num f nn ★ `\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.

26.38 \xintifTrueFalse

Num f nn ★ `\xintifTrueFalse{⟨N⟩}{⟨true branch⟩}{⟨false branch⟩}` is a synonym for `\xintifNotZero`. It is also available as `\xintifTrue` but this later name is a bit misleading as the macro must always have a false branch, possibly an empty brace pair `{}`.

26.39 \xintifCmp

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

26.40 \xintifEq

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

26.41 \xintifGt

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

26.42 \xintifLt

Num Num
f f nn ★

`\xintifLt{<A>}{}{<YES>}{<NO>}` checks if $A < B$ and in that case executes the YES branch. Extended to fractions (in particular decimal numbers) by **xintfrac**.

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

26.43 \xintifOdd

Num
f nn ★

`\xintifOdd{<A>}{<YES>}{<NO>}` checks if A is and odd integer and in that case executes the YES branch.

26.44 \xintFac

num
x ★

`\xintFac{x}` returns the factorial. It is an error if the argument is negative or at least 10^6 .

With **xintfrac** loaded, the macro is modified to accept a fraction as argument, as long as this fraction turns out to be an integer: `\xintFac {66/3}=112400072777607680000`. `\xintiFac` is a synonym not modified by the loading of **xintfrac**.

26.45 \xintDivision

Num Num
f f ★

`\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}`. The variant `\xintiiDivision` skips the overhead of parsing via `\xintNum`.

ff ★

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.

26.46 \xintQuo

Num Num
f f ★

`\xintQuo{N}{M}` returns the quotient from the euclidean division. When both N and M are positive one has `\xintQuo{N}{M}=\xintiTrunc {0}{N/M}` (using package **xintfrac**). With **xintfrac** loaded it accepts fractions on input, but they must be integers in disguise.

ff ★

The variant `\xintiiQuo` skips the overhead of parsing via `\xintNum`.

26.47 \xintRem

Num Num
f f ★

`\xintRem{N}{M}` returns the remainder from the euclidean division. With **xintfrac** loaded it accepts fractions on input, but they must be integers in disguise. The variant

ff ★

`\xintiiRem` skips the overhead of parsing via `\xintNum`.

26.48 \xintFDg

<code>\xintFDg{N}</code>	returns the first digit (most significant) of the decimal expansion. The variant
<code>\xintiFDg</code>	skips the overhead of parsing via <code>\xintNum</code> .

26.49 \xintLDg

Num
f ★ `\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. The variant `\xintiLDg` skips the overhead of parsing via `\xintNum`.

26.50 \xintMON, \xintMMON

Num
f ★ `\xintMON{N}` returns $(-1)^N$ and `\xintMMON{N}` returns $(-1)^{N-1}$.
`\xintMON {-280914019374101929}=-1`, `\xintMMON {-280914019374101929}=1`
f ★ The variants `\xintiMON` and `\xintiMMON` skip the overhead of parsing via `\xintNum`.

26.51 \xintOdd

Num
f ★ \xintOdd{N} is 1 if the number is odd and 0 otherwise. The variant \xintiiOdd skip the
f ★ overhead of parsing via \xintNum.

26.52 \xintiSqrt, \xintiSquareRoot

Num
f ★ \xintiSqrt{N} returns the largest integer whose square is at most equal to N.

```
\xintiSqrt {20000000000000000000000000000000}=1414213562373095048
\xintiSqrt {30000000000000000000000000000000}=1732050807568877293
        \xintiSqrt {\xintDSH {-120}{2}}=
1414213562373095048801688724209698078569671875376948073176679
```

Num
f ★ \xintiSquareRoot{N} returns {M}{d} with d>0, M^2-d=N and M smallest (hence =1+\xintiSqrt{N}).

```
\xintAssign\xintiSquareRoot {1700000000000000000000000}\to\A\B
        \xintiSub{\xintisqr\A}\B=\A^2-\B
1700000000000000000000000000=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`.

26.53 `\xintInc, \xintDec`

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

26.54 \xintDouble, \xintHalf

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

26.55 \xintDSL

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

26.56 \xintDSR

$f \star$ `\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.

26.57 \xintDSH

$\text{num}_x f \star$ `\xintDSH{x}{N}` is parametrized decimal shift. When x is negative, it is like iterating `\xintDSL |x|` times (*i.e.* multiplication by 10^{-x}). When x positive, it is like iterating `\DSR x` times (and is more efficient), and for a non-negative N this is thus the same as the quotient from the euclidean division by 10^x .

26.58 \xintDSHr, \xintDSx

$\text{num}_x f \star$ `\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 .

$\text{num}_x f \star$ `\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:->123456789000000000000000000000.
\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

```

26.59 **\xintDecSplit**

$\overset{\text{num}}{x}f \star$ **\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:->.

```

26.60 **\xintDecSplitL**

$\overset{\text{num}}{x}f \star$ **\xintDecSplitL{x}{N}** returns the first piece after the action of **\xintDecSplit**.

26.61 **\xintDecSplitR**

$\overset{\text{num}}{x}f \star$ **\xintDecSplitR{x}{N}** returns the second piece after the action of **\xintDecSplit**.

27 Commands of the **xintfrac** package

This package was first included in release 1.03 of the **xint** bundle. The general rule of the bundle that each macro first expands (what comes first, fully) each one of its arguments

applies.

Frac
f

f stands for an integer or a fraction (see [section 8](#) for the accepted input formats) or something which expands to an integer or fraction. It is possible to use in the numerator or the denominator of *f* count registers and even expressions with infix arithmetic operators, under some rules which are explained in the previous [Use of count registers](#) section.

num
x

As in the [xint.sty](#) documentation, *x* stands for something which will internally be embedded in a `\numexpr`. It may thus be a count register or something like `4*\count 255 + 17`, etc..., but must expand to an integer obeying the \TeX bound.

The fraction format on output is the scientific notation for the ‘float’ macros, and the `A/B[n]` format for all other fraction macros, with the exception of `\xintTrunc`, `\xintRound` (which produce decimal numbers) and `\xintIrr`, `\xintJrr`, `\xintRawWithZeros` (which returns an `A/B` with no trailing `[n]`, and prints the `B` even if it is 1), and `\xintPraw` which does not print the `[n]` if `n=0` or the `B` if `B=1`.

To be certain to print an integer output without trailing `[n]` nor fraction slash, one should use either `\xintPraw {\xintIrr {f}}` or `\xintNum {f}` when it is already known that *f* evaluates to a (big) integer. For example `\xintPraw {\xintAdd {2/5}{3/5}}` gives a perhaps disappointing `25/2555`, whereas `\xintPraw {\xintIrr {\xintAdd {2/5}{3/5}}}` returns 1. As we knew the result was an integer we could have used `\xintNum {\xintAdd {2/5}{3/5}}=1`.

Some macros (such as `\xintiTrunc`, `\xintiRound`, and `\xintFac`) always produce directly integers on output.

Contents

.1	<code>\xintNum</code>	72	.22	<code>\xintE</code>	76
.2	<code>\xintifInt</code>	72	.23	<code>\xintDigits, \xinttheDigits</code> ..	76
.3	<code>\xintLen</code>	72	.24	<code>\xintFloat</code>	76
.4	<code>\xintRaw</code>	72	.25	<code>\xintAdd</code>	77
.5	<code>\xintPraw</code>	72	.26	<code>\xintFloatAdd</code>	77
.6	<code>\xintNumerator</code>	73	.27	<code>\xintSub</code>	77
.7	<code>\xintDenominator</code>	73	.28	<code>\xintFloatSub</code>	77
.8	<code>\xintRawWithZeros</code>	73	.29	<code>\xintMul</code>	77
.9	<code>\xintREZ</code>	73	.30	<code>\xintFloatMul</code>	77
.10	<code>\xintFrac</code>	73	.31	<code>\xintSqr</code>	77
.11	<code>\xintSignedFrac</code>	74	.32	<code>\xintDiv</code>	78
.12	<code>\xintFwOver</code>	74	.33	<code>\xintFloatDiv</code>	78
.13	<code>\xintSignedFwOver</code>	74	.34	<code>\xintFac</code>	78
.14	<code>\xintIrr</code>	74	.35	<code>\xintPow</code>	78
.15	<code>\xintJrr</code>	74	.36	<code>\xintFloatPow</code>	78
.16	<code>\xintTrunc</code>	75	.37	<code>\xintFloatPower</code>	78
.17	<code>\xintiTrunc</code>	75	.38	<code>\xintFloatSqrt</code>	79
.18	<code>\xintRound</code>	75	.39	<code>\xintSum</code>	79
.19	<code>\xintiRound</code>	76	.40	<code>\xintPrd</code>	79
.20	<code>\xintFloor</code>	76	.41	<code>\xintCmp</code>	79
.21	<code>\xintCeil</code>	76	.42	<code>\xintIsOne</code>	79

⁵⁵yes, `\xintAdd` blindly multiplies denominators...

.43	<code>\xintGeq</code>	79	.49	<code>\xintSgn</code>	80
.44	<code>\xintMax</code>	80	.50	<code>\xintOpp</code>	80
.45	<code>\xintMaxof</code>	80	.51	<code>\xintDivision, \xintQuo, \xint-</code>	
.46	<code>\xintMin</code>	80		<code>Rem, \xintFDg, \xintLDg, \xint-</code>	
.47	<code>\xintMinof</code>	80		<code>MON, \xintMMON, \xintOdd</code>	80
.48	<code>\xintAbs</code>	80			

27.1 `\xintNum`

- f ★ The macro is extended to accept a fraction on input. But this fraction should reduce to an integer. If not an error will be raised. The original is available as `\xintiNum`. It is imprudent to apply `\xintNum` to numbers with a large power of ten given either in scientific notation or with the `[n]` notation, as the macro will add the necessary zeros to get an explicit integer.

27.2 `\xintifInt`

- $\frac{Frac}{f}$ nn ★ `\xintifInt{f}{YES branch}{NO branch}` expandably chooses the YES branch if `f` reveals itself after expansion and simplification to be an integer. As with the other **xint** conditionals, both branches must be present although one of the two (or both, but why then?) may well be an empty brace pair `{}`. As will all other **xint** conditionals, spaces in-between the braced things do not matter, but a space after the closing brace of the NO branch is significant.

27.3 `\xintLen`

- $\frac{Frac}{f}$ ★ The original macro is extended to accept a fraction on input.
`\xintLen {201710/298219}=11, \xintLen {1234/1}=4, \xintLen {1234}=4`

27.4 `\xintRaw`

- $\frac{Frac}{f}$ ★ This macro ‘prints’ the fraction `f` as it is received by the package after its parsing and expansion, in a form `A/B[n]` equivalent to the internal representation: the denominator `B` is always strictly positive and is printed even if it has value 1.
`\xintRaw{\the\numexpr 571*987\relax.123e-10/\the\numexpr -201+59\relax e-7}=`
`-563577123/142[-6]`

27.5 `\xintPRaw`

- $\frac{Frac}{f}$ ★ `PRaw` stands for “pretty raw”. It does *not* show the `[n]` if `n=0` and does *not* show the `B` if `B=1`.
`\xintPRaw {123e10/321e10}=123/321, \xintPRaw {123e9/321e10}=123/321[-1]`
`\xintPRaw {\xintIrr{861/123}}=7 vz. \xintIrr{861/123}=7/1`
 See also `\xintFrac` (or `\xintFwOver`) for math mode. As is exemplified above the `\xint-Irr` 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.

27.6 `\xintNumerator`

$\frac{\text{Frac}}{f}$ ★ 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]}=178000000000000000000000
\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`.

27.7 `\xintDenominator`

$\frac{\text{Frac}}{f}$ ★ This returns the denominator corresponding to the internal representation of the fraction:⁵⁶

```
\xintDenominator {178000/25600000[17]}=25600000
\xintDenominator {312.289001/20198.27}=20198270000
\xintDenominator {178000e-3/256e5}=25600000000
\xintDenominator {178.000/25600000}=25600000000
```

As shown by the examples, no simplification of the input is done. The denominator looks wrong in the last example, but the numerator was tacitly multiplied by 1000 through the removal of the decimal point. For a result uniquely associated to the value of the fraction first apply `\xintIrr`.

27.8 `\xintRawWithZeros`

$\frac{\text{Frac}}{f}$ ★ This macro ‘prints’ the fraction *f* (after its parsing and expansion) in A/B form, with A as returned by `\xintNumerator{f}` and B as returned by `\xintDenominator{f}`.

```
\xintRawWithZeros{\the\numexpr 571*987\relax.123e-10/\the\numexpr -201+59\relax e-7}=
-563577123/142000000
```

27.9 `\xintREZ`

$\frac{\text{Frac}}{f}$ ★ This command normalizes a fraction by removing the powers of ten from its numerator and denominator:

```
\xintREZ {178000/25600000[17]}=178/256[15]
\xintREZ {17800000000000e30/2560000000000e15}=178/256[15]
```

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

27.10 `\xintFrac`

$\frac{\text{Frac}}{f}$ ★ This is a \LaTeX only command, to be used in math mode only. It will print a fraction, internally represented as something equivalent to $A/B[n]$ as `\frac {A}{B}10^n`. The power of ten is omitted when $n=0$, the denominator is omitted when it has value one, the number being separated from the power of ten by a `\cdot`. $\text{\$}\xintFrac {178.000/25600000}\text{\$}$ gives $\frac{178000}{25600000}10^{-3}$, $\text{\$}\xintFrac {178.000/1}\text{\$}$ gives $178000 \cdot 10^{-3}$, $\text{\$}\xintFrac {3.5/5.7}\text{\$}$ gives $\frac{35}{57}$, and $\text{\$}\xintFrac {\xintNum {\xintFac{10}}}$

⁵⁶recall that the `[]` construct excludes presence of a decimal point.

`/\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.)

27.11 `\xintSignedFrac`

$\frac{\text{Frac}}{f}$ ★ This is as `\xintFrac` except that a negative fraction has the sign put in front, not in the numerator.

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

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

27.12 `\xintFwOver`

$\frac{\text{Frac}}{f}$ ★ This does the same as `\xintFrac` except that the `\over` primitive is used for the fraction (in case the denominator is not one; and a pair of braces contains the `A\over B` part). `\xintFwOver {178.000/256000000}` gives $\frac{178000}{256000000}10^{-3}$, `\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.

27.13 `\xintSignedFwOver`

$\frac{\text{Frac}}{f}$ ★ This is as `\xintFwOver` except that a negative fraction has the sign put in front, not in the numerator.

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

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

27.14 `\xintIrr`

$\frac{\text{Frac}}{f}$ ★ This puts the fraction into its unique irreducible form:

$$\xintIrr {178.256/256.178}=6856/9853 = \frac{6856}{9853}$$

Note that the current implementation does not cleverly first factor powers of 2 and 5, so input such as `\xintIrr {2/3[100]}` will make **xintfrac** do the Euclidean division of $2 \cdot 10^{100}$ 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}}`.

27.15 `\xintJrr`

$\frac{\text{Frac}}{f}$ ★ This also puts the fraction into its unique irreducible form:

$$\xintJrr {178.256/256.178}=6856/9853$$

This is faster than `\xintIrr` for fractions having some big common factor in the numerator and the denominator.

```
\xintJrr {\xintiPow{\xintFac {15}}{3}/\xintiiPrdExpr {\xintFac{10}}{
\xintFac{30}}{\xintFac{5}}\relax }=1001/51705840
```

But to notice the difference one would need computations with much bigger numbers than in this example. Starting with release 1.08, `\xintJrr` does not remove the trailing /1 when the output is an integer.

27.16 `\xintTrunc`

$\text{num}_x^{\text{Frac}} \star$

`\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}}=-0.0000000000003
\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.⁵⁷

27.17 `\xintiTrunc`

$\text{num}_x^{\text{Frac}} \star$

`\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.)

27.18 `\xintRound`

$\text{num}_x^{\text{Frac}} \star$

`\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}}=-0.0000000000004
\xintRound {12}{\xintAdd {-1/3}{3/9}}=0
```

⁵⁷Recall that `-\macro` is not valid as argument to any package macro, one must use `\xintOpp{\macro}` or `\xintiOpp{\macro}`, except inside `\xinttheexpr...\relax`.

The identity $\backslash\mathrm{xintRound}\{x\}{-f} = -\backslash\mathrm{xintRound}\{x\}{f}$ holds. And regarding $(-11)^{-11}$ here is some more of its expansion:

-0.000000000000350493899481392497604003313162598556370...

27.19 $\backslash\mathrm{xintiRound}$

$\frac{\mathrm{num}}{x} \frac{\mathrm{Frac}}{f}$ ★ $\backslash\mathrm{xintiRound}\{x\}{f}$ returns the integer equal to 10^x times what $\backslash\mathrm{xintRound}\{x\}{f}$ would return.

$\backslash\mathrm{xintiRound}\{16\}{-803.2028/20905.298} = -384210165289201$
 $\backslash\mathrm{xintiRound}\{10\}{\backslash\mathrm{xintPow}\{-11\}{-11}} = 0$

Differences between $\backslash\mathrm{xintRound}\{0\}{f}$ and $\backslash\mathrm{xintiRound}\{0\}{f}$: 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.)

27.20 $\backslash\mathrm{xintFloor}$

$\frac{\mathrm{Frac}}{f}$ ★ $\backslash\mathrm{xintFloor}\{f\}$ returns the largest relative integer N with $N \leq f$.
 $\backslash\mathrm{xintFloor}\{-2.13\} = -3$, $\backslash\mathrm{xintFloor}\{-2\} = -2$, $\backslash\mathrm{xintFloor}\{2.13\} = 2$

27.21 $\backslash\mathrm{xintCeil}$

$\frac{\mathrm{Frac}}{f}$ ★ $\backslash\mathrm{xintCeil}\{f\}$ returns the smallest relative integer N with $N > f$.
 $\backslash\mathrm{xintCeil}\{-2.13\} = -2$, $\backslash\mathrm{xintCeil}\{-2\} = -2$, $\backslash\mathrm{xintCeil}\{2.13\} = 3$

27.22 $\backslash\mathrm{xintE}$

$\frac{\mathrm{Frac}}{f} \frac{\mathrm{num}}{x}$ ★ $\backslash\mathrm{xintE}\{f\}{x}$ multiplies the fraction f by 10^x . The *second* argument x must obey the $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ bounds. Example:

$\backslash\mathrm{count}\ 255\ 123456789\ \backslash\mathrm{xintE}\{10\}{\backslash\mathrm{count}\ 255} \rightarrow 10/1[123456789]$

Be careful that for obvious reasons such gigantic numbers should not be given to $\backslash\mathrm{xint-Num}$, 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:

$\backslash\mathrm{xintFloatAdd}\{1\mathrm{e}1234567890\}{1} = 1.0000000000000000\mathrm{e}1234567890$

27.23 $\backslash\mathrm{xintDigits}$, $\backslash\mathrm{xinttheDigits}$

The syntax $\backslash\mathrm{xintDigits} := D$; (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 $\backslash\mathrm{xinttheDigits}$ serves to print the current value.

27.24 $\backslash\mathrm{xintFloat}$

$\frac{\mathrm{num}}{[x]} \frac{\mathrm{Frac}}{f}$ ★ The macro $\backslash\mathrm{xintFloat}[P]{f}$ has an optional argument P which replaces the current value of $\backslash\mathrm{xintDigits}$. 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 $10.0\dots0\mathrm{e}N$ (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`-expression, 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`.

27.25 `\xintAdd`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★

The original macro is extended to accept fractions on input. Its output will now always be in the form `A/B[n]`. The original is available as `\xintiAdd`.

27.26 `\xintFloatAdd`

$\left[\frac{\text{num}}{x}\right] \frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★

`\xintFloatAdd [P]{f}{g}` first replaces `f` and `g` with their float approximations, with 2 safety digits. It then adds exactly and outputs in float format with precision `P` (which is optional) or `\xintDigits` if `P` was absent, the result of this computation.

27.27 `\xintSub`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★

The original macro is extended to accept fractions on input. Its output will now always be in the form `A/B[n]`. The original is available as `\xintiSub`.

27.28 `\xintFloatSub`

$\left[\frac{\text{num}}{x}\right] \frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★

`\xintFloatSub [P]{f}{g}` first replaces `f` and `g` with their float approximations, with 2 safety digits. It then subtracts exactly and outputs in float format with precision `P` (which is optional), or `\xintDigits` if `P` was absent, the result of this computation.

27.29 `\xintMul`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★

The original macro is extended to accept fractions on input. Its output will now always be in the form `A/B[n]`. The original, only for big integers, and outputting a big integer, is available as `\xintiMul`.

27.30 `\xintFloatMul`

$\left[\frac{\text{num}}{x}\right] \frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★

`\xintFloatMul [P]{f}{g}` first replaces `f` and `g` with their float approximations, with 2 safety digits. It then multiplies exactly and outputs in float format with precision `P` (which is optional), or `\xintDigits` if `P` was absent, the result of this computation.

27.31 `\xintSqr`

$\frac{\text{Frac}}{f}$ ★

The original macro is extended to accept a fraction on input. Its output will now always be in the form `A/B[n]`. The original which outputs only big integers is available as `\xintiSqr`.

27.32 \xintDiv

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★ `\xintDiv{f}{g}` computes the fraction f/g . As with all other computation macros, no simplification is done on the output, which is in the form $A/B[n]$.

27.33 \xintFloatDiv

$\left[\frac{\text{num}}{x}\right] \frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★ `\xintFloatDiv [P]{f}{g}` first replaces f and g with their float approximations, with 2 safety digits. It then divides exactly and outputs in float format with precision P (which is optional), or `\xintDigits` if P was absent, the result of this computation.

27.34 \xintFac

$\frac{\text{Num}}{f}$ ★ The original is extended to allow a fraction on input but this fraction f must simplify to a integer n (non negative and at most 999999, but already 100000! is prohibitively time-costly). On output $n!$ (no trailing $/1[0]$). The original macro (which has less overhead) is still available as `\xintiFac`.

27.35 \xintPow

$\frac{\text{Frac}}{f} \frac{\text{Num}}{f}$ ★ `\xintPow{f}{g}`: the original macro is extended to accept fractions on input. The output will now always be in the form $A/B[n]$ (even when the exponent vanishes: `\xintPow{2/3}{0}=1/1[0]`). The original is available as `\xintiPow`.

The exponent is allowed to be input as a fraction but it must simplify to an integer: `\xintPow{2/3}{10/2}=32/243[0]`. This integer will be checked to not exceed 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.

27.36 \xintFloatPow

$\left[\frac{\text{num}}{x}\right] \frac{\text{Frac}}{f} \frac{\text{num}}{x}$ ★ `\xintFloatPow [P]{f}{x}` uses either the optional argument P or the value of `\xintDigits`. It computes a floating approximation to f^x .

The 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 $\text{T}_{\text{E}}\text{X}$ 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`

27.37 \xintFloatPower

$\left[\frac{\text{num}}{x}\right] \frac{\text{Frac}}{f} \frac{\text{Num}}{f}$ ★ `\xintFloatPower[P]{f}{g}` computes a floating point value f^g where the exponent g is not constrained to be at most the $\text{T}_{\text{E}}\text{X}$ bound 2147483647. It may even be a fraction A/B but must simplify to a (possibly big) integer.

`\xintFloatPower [8]{1.0000000000001}{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.

27.38 `\xintFloatSqrt`

$\left[\begin{smallmatrix} \text{num} \\ x \end{smallmatrix}\right] \overset{\text{Frac}}{f} \star$

`\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
```

27.39 `\xintSum`

$f \rightarrow * \overset{\text{Frac}}{f} \star$

The original command is extended to accept fractions on input and produce fractions on output. The output will now always be in the form `A/B[n]`. The original, for big integers only (in strict format), is available as `\xintiSum`.

27.40 `\xintPrd`

$f \rightarrow * \overset{\text{Frac}}{f} \star$

The original is extended to accept fractions on input and produce fractions on output. The output will now always be in the form `A/B[n]`. The original, for big integers only (in strict format), is available as `\xintiPrd`.

27.41 `\xintCmp`

$\overset{\text{Frac}}{f} \overset{\text{Frac}}{f} \star$

The macro is extended to fractions. Its output is still either `-1`, `0`, or `1` with no forward slash nor trailing `[n]`.

For choosing branches according to the result of comparing `f` and `g`, the following syntax is recommended: `\xintSgnFork{\xintCmp{f}{g}}{code for f<g}{code for f=g}{code for f>g}`.

27.42 `\xintIsOne`

$\overset{\text{Frac}}{f} \star$

See `\xintIsOne` (subsection 26.17).

27.43 `\xintGeq`

$\overset{\text{Frac}}{f} \overset{\text{Frac}}{f} \star$

The macro is extended to fractions. Beware that the comparison is on the *absolute values* of the fractions. Can be used as: `\xintSgnFork{\xintGeq{f}{g}}{code for |f|<|g|}{code for |f|≥|g|}`

27.44 \xintMax

$\frac{\frac{f}{f}}$ ★ The macro is extended to fractions. But now `\xintMax {2}{3}` returns 3/1 [0]. The original, for use with (possibly big) integers only, is available as `\xintiMax`: `\xintiMax {2}{3}=3`.

27.45 \xintMaxof

$f \rightarrow *^{\text{Frac}} f$ ★ See `\xintMaxof` (subsection 26.26).

27.46 \xintMin

$\frac{f}{f}$ ★ The macro is extended to fractions. The original, for (big) integers only, is available as `\xintiMin`.

27.47 \xintMinof

$f \rightarrow * f^{\text{Frac}}$ ★ See `\xintMinof` (subsection 26.28).

27.48 \xintAbs

$\frac{\text{Frac}}{f}$ ★ The macro is extended to fractions. The original, for (big) integers only, is available as `\xintiAbs`. Note that `\xintiAbs {-2}=2/1 [0]` whereas `\xintiAbs {-2}=2`.

27.49 \xintSgn

$\frac{\text{Frac}}{f}$ ★ The macro is extended to fractions. Naturally, its output is still either -1, 0, or 1 with no forward slash nor trailing [n].

27.50 \xint0pp

$\frac{\text{Frac}}{f}$ ★ The macro is extended to fractions. The original is available as `\xintiOpp`. Note that `\xintiOpp {3}` now outputs `-3/1[0]` whereas `\xintiOpp {3}` returns `-3`.

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

$\frac{f}{f}$ ★	These macros accept a fraction on input if this fraction in fact reduces to an integer (if not an <code>\xintError:NotAnInteger</code> will be raised). There is no difference in the format of the outputs, which are still (possibly big) integers without fraction slash nor trailing [n], the sole difference is in the extended range of accepted inputs.
or $\frac{f}{f}$ ★	

All have variants whose names start with `xintii` rather than `xint`; these variants accept on input only integers in the strict format (they do not use `\xintNum`). They thus have less overhead, and may be used when one is dealing exclusively with (big) integers. These variants are already available in `xint`, there is no need for `xintfrac` to be loaded.

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

[illegible]

28 Expandable expressions with the **xintexpr** package

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

The syntax is described in [section 23](#) and [section 24](#).

Contents

.1	The <code>\xintexpr</code> expressions	81	.9	<code>\xintifboolexpr</code>	86
.2	<code>\numexpr</code> expressions, count and dimension registers	82	.10	<code>\xintifboolfloatexpr</code>	86
.3	Catcodes and spaces	82	.11	<code>\xintfloatexpr</code> , <code>\xintthe- floatexpr</code>	86
.4	Expandability	83	.12	<code>\xintNewFloatExpr</code>	87
.5	Memory considerations	83	.13	<code>\xintNewNumExpr</code>	87
.6	The <code>\xintNewExpr</code> command	84	.14	<code>\xintNewBoolExpr</code>	87
.7	<code>\xintnumexpr</code> , <code>\xintthenumexpr</code> . .	86	.15	Technicalities	88
.8	<code>\xintboolexpr</code> , <code>\xintthebool- expr</code>	86	.16	Acknowledgements	88

28.1 The `\xintexpr` expressions

x ★ An **xintexpression** is a construct `\xintexpr⟨expandable_expression⟩\relax` where the expandable expression is read and completely expanded from left to right.

During this parsing, braced sub-content `{⟨expandable⟩}` may be serving as a macro parameter, or a branch of the `?` two-way and `:` three-way operators; else it is treated in a special manner:

1. it is allowed to occur only at the spots where numbers are legal,
2. the `⟨expandable⟩` contents is then completely expanded as if it were put in a `\csname . . \endcsname`,⁵⁸ thus it escapes entirely the parser scope and infix notations will not be recognized except if the expanded macros know how to handle them by themselves,
3. and this complete expansion *must* produce a number or a fraction, possibly with decimal mark and trailing `[n]`, the scientific notation is *not* authorized.

Braces are the only way to input some number or fraction with a trailing `[n]` as square brackets are *not* accepted in a `\xintexpr . . \relax` if not within such braces.

An `\xintexpr . . \relax` *must* end in a `\relax` (which will be absorbed). Like a `\numexpr` expression, it is not printable as is, nor can it be directly employed as argument to the other package macros. For this one must use one of the two equivalent forms:

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

⁵⁸well, actually it is put in such a `\csname . . \endcsname`.

The computations are done *exactly*, and with no simplification of the result. The output format for the result can be coded inside the expression through the use of one of the functions `round`, `trunc`, `float`, `reduce`.⁵⁹ Here are some examples

```
\xinttheexpr 1/5!-1/7!-1/9!\relax=1784764800/219469824000[0]
\xinttheexpr round(1/5!-1/7!-1/9!,18)\relax=0.008132164902998236
\xinttheexpr float(1/5!-1/7!-1/9!,18)\relax=813216490299823633[-20]
\xinttheexpr reduce(1/5!-1/7!-1/9!)\relax=2951/362880
\xinttheexpr 1.99^-2 - 2.01^-2 \relax=800/1599920001[4]
\xinttheexpr round(1.99^-2 - 2.01^-2, 10)\relax=0.0050002500
```

- the expression may contain arbitrarily many levels of nested parenthesized sub-expressions,
- sub-contents giving numbers of fractions should be either
 1. parenthesized,
 2. a sub-expression `\xintexpr...\relax`,
 3. or within braces.
- 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` (or `\xinttheexpr...\relax`) but simply `\xintexpr...\relax`,
- each **xintexpr** expression is completely expandable and obtains its result in two expansion steps.

28.2 `\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.

28.3 Catcodes and spaces

28.3.1 `\xintexprSafeCatcodes`

Active characters will interfere with `\xintexpr`-essions. One may prefix them with `\string` or use the command `\xintexprSafeCatcodes` before the `\xintexpr`-essions. This sets (not globally) the catcodes of the relevant characters to safe values. The command `\xintNewExpr` does it by itself internally (restoring the catcodes on exit).

⁵⁹In `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.

28.3.2 `\xintexprRestoreCatcodes`

Restores the catcodes to the earlier state.

Unbraced spaces inside an `\xinttheexpr... \relax` should mostly be innocuous (except inside macro arguments).

`\xintexpr` and `\xinttheexpr` are very agnostic regarding catcodes: (unbraced) 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 `+, -, *, /, ^, !, &, |, ?, :, <, >, =, (,)` and the comma should not be active as everything is expanded along the way. If one of them is active, it can be prefixed with `\string`.

Digits, slash, square brackets, minus sign, in the output from an `\xinttheexpr` are all of catcode 12. For `\xintthefloatexpr` the ‘e’ in output is of catcode 11.

A macro with arguments will expand and grab its arguments before the parser may get a chance to see them, so the situation with catcodes and spaces is not the same within such macro arguments (or within braces used to protect square brackets).

28.4 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.

28.5 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 plots⁶⁰, this may cause a problem.

There is a solution.⁶¹

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

⁶⁰this 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.

⁶¹which convinced me that I could stick with the parser implementation despite its potential impact on the hash-table.

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.

28.6 The `\xintNewExpr` command

The command is used as:

`\xintNewExpr{myformula}[n]{stuff}`, where

- `stuff` will be inserted inside `\xinttheexpr . . . \relax`,
- `n` is an integer between zero and nine, inclusive, and tells how many parameters will `myformula` have (it is *mandatory* even if `n=0`⁶²)
- the placeholders `#1`, `#2`, ..., `#n` are used inside `stuff` in their usual rôle.

The macro `\myformula` is defined without checking if it already exists, \LaTeX users might prefer to do first `\newcommand*myformula {}` to get a reasonable error message in case `\myformula` already exists.

The definition of `\myformula` made by `\xintNewExpr` is global. The protection against active characters is done automatically.

It will be a completely expandable macro entirely built-up using `\xintAdd`, `\xintSub`, `\xintMul`, `\xintDiv`, `\xintPow`, etc... as corresponds to the expression written with the infix operators.

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:

⁶²there is some use for `\xintNewExpr[0]` compared to an `\edef` as `\xintNewExpr` has some built-in catcode protection.

```
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}}}{\xint
Mul {\xintMul {#2}{#4}}{#9}}}{\xintMul {\xintMul {#3}{#5}}{#7}}
```

This is why `\printnumber` was used, to have breaks across lines.

28.6.1 Use of conditional operators

The 1.09a conditional operators `?` and `:` cannot be parsed by `\xintNewExpr` when they contain macro parameters `#1, ..., #9` within their scope. However replacing them with the functions `if` and, respectively `ifsgn`, the parsing should succeed. And the created macro will *not evaluate the branches to be skipped*, thus behaving exactly like `?` and `:` would have in the `\xintexpr`.

```
\xintNewExpr\Formula [3]
{ if((#1>#2) & (#2>#3), sqrt(#1-#2)*sqrt(#2-#3), #1^2+#3/#2) }
\meaning\Formula:macro:#1#2#3->\romannumeral-'0\xintifNotZero{\xintAND{
\xintGt{#1}{#2}}{\xintGt{#2}{#3}}}{\xintMul{\XINTinFloatSqrt[\XINTdigit
s]{\xintSub{#1}{#2}}}{\XINTinFloatSqrt[\XINTdigits]{\xintSub{#2}{#3}}}}
{\xintAdd{\xintPow{#1}{2}}{\xintDiv{#3}{#2}}}
```

This formula (with `\xintifNotZero`) will gobble the false branch.

Remark: this `\XINTinFloatSqrt` macro is a non-user package macro used internally within `\xintexpr`-essions, it produces the result in `A[n]` form rather than in scientific notation, and for reasons of the inner workings of `\xintexpr`-essions, this is necessary; a hand-made macro would have used instead the equivalent `\xintFloatSqrt`.

Another example

```
\xintNewExpr\myformula [3]
{ ifsgn(#1,#2/#3,#2-#3,#2*#3) }
macro:#1#2#3->\romannumeral-'0\xintifSgn{#1}{\xintDiv{#2}{#3}}{\xintSub
{#2}{#3}}{\xintMul{#2}{#3}}
```

Again, this macro gobbles the false branches, as would have the operator `:` inside an `\xintexpr`-ession.

28.6.2 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}}
```

28.7 **\xintnumexpr**, **\xintthenumexpr**

- x ★ 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.

28.8 **\xintboolexpr**, **\xinttheboolexpr**

- x ★ Equivalent to doing `\xintexpr ... \relax` and returning 1 if the result does not vanish, and 0 if 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)).

28.9 **\xintifboolexpr**

- x n n ★ `\xintifboolexpr{<expr>}{YES}{NO}` does `\xinttheexpr <expr>\relax` and then executes the YES or the NO branch depending on whether the outcome was non-zero or zero. 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.

28.10 **\xintifboolfloatexpr**

- x n n ★ `\xintifboolfloatexpr{<expr>}{YES}{NO}` does `\xintthefloatexpr <expr>\relax` and then executes the YES or the NO branch depending on whether the outcome was non zero or zero. This will crash if used on an expression which is a comma separated list.

28.11 **\xintfloatexpr**, **\xintthefloatexpr**

- x ★ `\xintfloatexpr ... \relax` is exactly like `\xintexpr ... \relax` but with the four binary operations and the power function mapped to `\xintFloatAdd`, `\xintFloatSub`, `\xintFloatMul`, `\xintFloatDiv` and `\xintFloatPower`. The precision is from the current setting of `\xintDigits` (it can not be given as an optional parameter).

Currently, the factorial function hasn't yet a float version; so inside `\xintthefloatexpr ... \relax`, `n!` will be computed exactly. Perhaps this will be improved in a future release.

Note that `1.0000000001` 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 1.0000000001 , 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 1.0000000001-1\relax=1.00000000e-9
For the fun of it: \xintDigits:=20;
\xintthefloatexpr (1+1e-7)^1e7\relax=2.7182816925449662712e0
\xintDigits:=36;
\xintthefloatexpr ((1/13+1/121)*(1/179-1/173))/(1/19-1/18)\relax
5.64487459334466559166166079096852897e-3
\xintFloat{\xinttheexpr ((1/13+1/121)*(1/179-1/173))/(1/19-1/18)\relax}
5.64487459334466559166166079096852912e-3
```

The latter result is the rounding of the exact result. The previous one has rounding errors coming from the various roundings done for each sub-expression. It was a bit funny to discover that maple, configured with Digits:=36; and with decimal dots everywhere to let it input the numbers as floats, gives exactly the same result with the same rounding errors as does `\xintthefloatexpr`!

Using `\xintthefloatexpr` only pays off compared to using `\xinttheexpr` followed with `\xintFloat` if the computations turn out to involve hundreds of digits. For elementary calculations with hand written numbers (not using the scientific notation with exponents differing greatly) it will generally be more efficient to use `\xinttheexpr`. The situation is quickly otherwise if one starts using the Power function. Then, `\xintthefloat` is often useful; and sometimes indispensable to achieve the (approximate) computation in reasonable time.

We can try some crazy things:

```
\xintDigits:=12;\xintthefloatexpr 1.0000000000000001^1e15\relax
2.71828182846e0
```

Contrarily to some professional computing software which are our concurrents on this market, the 1.0000000000000001 wasn't rounded to 1 despite the setting of `\xintDigits`; it would have been if we had input it as $(1+1e-15)$.

28.12 `\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`.

28.13 `\xintNewNumExpr`

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

28.14 `\xintNewBoolExpr`

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

28.15 Technicalities

As already mentioned `\xintNewExpr\myformula[n]` does not check the prior existence of a macro `\myformula`. And the number of parameters `n` given as mandatory argument withing square brackets should be (at least) equal to the number of parameters in the expression.

Obviously I should mention that `\xintNewExpr` itself can not be used in an expansion-only context, as it creates a macro.

The 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 (presumably undefined, but it does not matter) 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.

Syntax errors in the input such as using a one-argument function with two arguments will generate low-level \TeX processing unrecoverable errors, with cryptic accompanying message.

Some other problems will give rise to ‘error messages’ macros giving some indication on the location and nature of the problem. Mainly, an attempt has been made to handle gracefully missing or extraneous parentheses.

When the scanner is looking for a number and finds something else not otherwise treated, it assumes it is the start of the function name and will expand forward in the hope of hitting an opening parenthesis; if none is found at least it should stop when encountering the `\relax` marking the end of the expressions.

Note that `\relax` is mandatory (contrarily to a `\numexpr`).

28.16 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.

29 Commands of the **xintbinhex** package

This package was first included in the 1.08 release of **xint**. It provides expandable conversions of arbitrarily long numbers to and from binary and hexadecimal.

The argument is first *f*-expanded. It then may start with an optional minus sign (unique, of category code other), followed with optional leading 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 uppcased.

Contents

.1	<code>\xintDecToHex</code>	89	.5	<code>\xintBinToHex</code>	90
.2	<code>\xintDecToBin</code>	89	.6	<code>\xintHexToBin</code>	90
.3	<code>\xintHexToDec</code>	89	.7	<code>\xintCHexToBin</code>	90
.4	<code>\xintBinToDec</code>	89			

29.1 `\xintDecToHex`

f ★ Converts from decimal to hexadecimal.

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

29.2 `\xintDecToBin`

f ★ Converts from decimal to binary.

```
\xintDecToBin{2718281828459045235360287471352662497757247093699959574
966967627724076630353547594571382178525166427427466391932003}
->100011010100100111001011111000110011010010100100110101001011100000
10100011111011111010000101010000001011110010001010011100011111000001
01100010111110001000001101100010001110001001000101110101110111100101
01101010111011000001011101100111000110100100111001011110100011011011
10011100100011011000110000000110010100100110110101111110011011111011
0101100100100011000100000010100110001100011
```

29.3 `\xintHexToDec`

f ★ Converts from hexadecimal to decimal.

```
\xintHexToDec{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B576
0BB38D272F46DCE46C6032936BF37DAC918814C63}
->271828182845904523536028747135266249775724709369995957496696762772
4076630353547594571382178525166427427466391932003
```

29.4 `\xintBinToDec`

f ★ Converts from binary to decimal.

```
\xintBinToDec{1000110101001001110010111110001100110100101001001101010
010111000000101000111110111110100001010100000010111100100010100111000111
11000001011000101111100010000011011000100011100010010001011101011101111
00101011010101110110000010111011001110001101001001110010111101000110110
11100111001000110110001100000001100101001001101101011111100110111110110
101100100100011000100000010100110001100011}
```

```
->271828182845904523536028747135266249775724709369995957496696762772
4076630353547594571382178525166427427466391932003
```

29.5 **\xintBinToHex**

f★ Converts from binary to hexadecimal.

```
\xintBinToHex{1000110101001001110010111110001100110100101001001101010
010111000000101000111110111101000001010100000010111100100010100111000111
1100000010110001011111000100000011011000100011100010010001011101011101111
001010110101011101100000010111011001110001101001001110010111101000110110
111001110010001101100011000000001100101001001101101011111100110111110110
1011001001000110001000000010100110001100011}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F4
6DCE46C6032936BF37DAC918814C63
```

29.6 **\xintHexToBin**

f★ Converts from hexadecimal to binary.

```
\xintHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B576
0BB38D272F46DCE46C6032936BF37DAC918814C63}
->100011010100100111001011111000110011010010100100110101001011100000
10100011111011111010000101010000001011110010001010011100011111000001
01100010111110001000001101100010001110001001000101110101110111100101
01101010111011000001011101100111000110100100111001011110100011011011
10011100100011011000110000000110010100100110110101111110011011111011
0101100100100011000100000010100110001100011
```

29.7 **\xintCHexToBin**

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

```
\xintCHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B57
60BB38D272F46DCE46C6032936BF37DAC918814C63}
->100011010100100111001011111000110011010010100100110101001011100000
10100011111011111010000101010000001011110010001010011100011111000001
01100010111110001000001101100010001110001001000101110101110111100101
01101010111011000001011101100111000110100100111001011110100011011011
10011100100011011000110000000110010100100110110101111110011011111011
0101100100100011000100000010100110001100011
```

30 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

.1	<code>\xintGCD</code>	91	.6	<code>\xintEuclideanAlgorithm</code>	91
.2	<code>\xintGCDof</code>	91	.7	<code>\xintBezoutAlgorithm</code>	92
.3	<code>\xintLCM</code>	91	.8	<code>\xintTypesetEuclideanAlgorithm</code>	
.4	<code>\xintLCMof</code>	91		92
.5	<code>\xintBezout</code>	91	.9	<code>\xintTypesetBezoutAlgorithm</code>	92

30.1 `\xintGCD`

$\overset{\text{Num}}{f} \overset{\text{Num}}{f} \star$ `\xintGCD{N}{M}` computes the greatest common divisor. It is positive, except when both N and M vanish, in which case the macro returns zero.

```
\xintGCD{10000}{1113}=1
\xintGCD{123456789012345}{9876543210321}=3
```

30.2 `\xintGCDof`

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

30.3 `\xintLCM`

$\overset{\text{Num}}{f} \overset{\text{Num}}{f} \star$ `\xintGCD{N}{M}` computes the least common multiple. It is 0 if one of the two integers vanishes.

30.4 `\xintLCMof`

$f \rightarrow * \overset{\text{Num}}{f} \star$ `\xintLCMof{{a}{b}{c}...}` computes the least common multiple of all integers a, b, ... The list argument may be a macro, it is *f*-expanded first and must contain at least one item.

30.5 `\xintBezout`

$\overset{\text{Num}}{f} \overset{\text{Num}}{f} \star$ `\xintBezout{N}{M}` returns five numbers A, B, U, V, D within braces. A is the first (expanded, as usual) input number, B the second, D is the GCD, and $UA - VB = D$.

```
\xintAssign {\xintBezout {10000}{1113}}\to\X
\meaning\X: macro:->{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.
```

30.6 `\xintEuclideanAlgorithm`

$\overset{\text{Num}}{f} \overset{\text{Num}}{f} \star$ `\xintEuclideanAlgorithm{N}{M}` applies the Euclidean algorithm and keeps a copy of all quotients and remainders.

```
\xintAssign {\xintEuclideanAlgorithm {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.

30.7 `\xintBezoutAlgorithm`

Num Num
f f ★

`\xintBezoutAlgorithm{N}{M}` applies the Euclidean 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 {\xintEuclideanAlgorithm {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.

30.8 `\xintTypesetEuclideanAlgorithm`

Num Num
f f

This macro is just an example of how to organize the data returned by `\xintEuclideanAlgorithm`. Copy the source code to a new macro and modify it to what is needed.

`\xintTypesetEuclideanAlgorithm {123456789012345}{9876543210321}`

$$123456789012345 = 12 \times 9876543210321 + 4938270488493$$

$$9876543210321 = 2 \times 4938270488493 + 2233335$$

$$4938270488493 = 2211164 \times 2233335 + 536553$$

$$2233335 = 4 \times 536553 + 87123$$

$$536553 = 6 \times 87123 + 13815$$

$$87123 = 6 \times 13815 + 4233$$

$$13815 = 3 \times 4233 + 1116$$

$$4233 = 3 \times 1116 + 885$$

$$1116 = 1 \times 885 + 231$$

$$885 = 3 \times 231 + 192$$

$$231 = 1 \times 192 + 39$$

$$192 = 4 \times 39 + 36$$

$$39 = 1 \times 36 + 3$$

$$36 = 12 \times 3 + 0$$

30.9 `\xintTypesetBezoutAlgorithm`

Num Num
f f

This macro is just an example of how to organize the data returned by `\xintBezoutAlgorithm`. Copy the source code to a new macro and modify it to what is needed.

`\xintTypesetBezoutAlgorithm {10000}{1113}`

$$10000 = 8 \times 1113 + 1096$$

$$8 = 8 \times 1 + 0$$

$$1 = 8 \times 0 + 1$$

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

31 Commands of the **xintseries** package

Some arguments to the package commands are macros which are expanded only later, when given their parameters. The arguments serving as indices are systematically given to a `\numexpr` expressions (new with 1.06!), hence *f*-expanded, they may be count registers, etc...

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

We use $\overset{\text{Frac}}{f}$ for the expansion type of various macro arguments, but if only **xint** and not **xintfrac** is loaded this should be more appropriately $\overset{\text{Num}}{f}$. The macro `\xintiSeries` is special and expects summing big integers obeying the strict format, even if **xintfrac** is loaded.

Contents

.1	<code>\xintSeries</code>	93	.7	<code>\xintFxFtPowerSeries</code>	103
.2	<code>\xintiSeries</code>	95	.8	<code>\xintFxFtPowerSeriesX</code>	104
.3	<code>\xintRationalSeries</code>	96	.9	<code>\xintFloatPowerSeries</code>	105
.4	<code>\xintRationalSeriesX</code>	99	.10	<code>\xintFloatPowerSeriesX</code>	106
.5	<code>\xintPowerSeries</code>	101	.11	Computing $\log 2$ and π	106
.6	<code>\xintPowerSeriesX</code>	103			

31.1 `\xintSeries`

$\overset{\text{num}}{x} \overset{\text{num}}{x} \overset{\text{Frac}}{f}$ ★

`\xintSeries{A}{B}{\coeff}` computes $\sum_{n=A}^{n=B} \coeff{n}$. The initial and final indices must obey the `\numexpr` constraint of expanding to numbers at most $2^{31}-1$. The `\coeff` macro must be a one-parameter *f*-expandable command, taking on input an explicit number *n* and producing some number or fraction `\coeff{n}`; it is expanded at the time it is needed.⁶³

⁶³`\xintiMON` is like `\xintMON` but does not parse its argument through `\xintNum`, for efficiency; other macros of this type are `\xintiiAdd`, `\xintiiMul`, `\xintiiSum`, `\xintiiPrd`, `\xintiiMMON`, `\xint-iiLDg`, `\xintiiFDg`, `\xintiiOdd`, ...

```

\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.
\[ \sum_{n=0}^{50} \frac{(-1)^n}{n + \frac{1}{2}} = \xintFrac\z \]

```

$$\sum_{n=0}^{50} \frac{(-1)^n}{n + \frac{1}{2}} = \frac{173909338287370940432112792101626602278714}{110027467159390003025279917226039729050575}$$

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...	9. 0.745634920634...	17. 0.721695379783...
2. 0.500000000000...	10. 0.645634920634...	18. 0.666139824228...
3. 0.833333333333...	11. 0.736544011544...	19. 0.718771403175...
4. 0.583333333333...	12. 0.653210678210...	20. 0.668771403175...
5. 0.783333333333...	13. 0.730133755133...	21. 0.716390450794...
6. 0.616666666666...	14. 0.658705183705...	22. 0.670935905339...
7. 0.759523809523...	15. 0.725371850371...	23. 0.714414166209...
8. 0.634523809523...	16. 0.662871850371...	24. 0.672747499542...

25. 0.712747499542... 27. 0.711322998118... 29. 0.710091471024...
 26. 0.674285961081... 28. 0.675608712404... 30. 0.676758137691...

31.2 `\xintiSeries`

$\frac{\text{num}}{x} \frac{\text{num}}{x} f \star$

`\xintiSeries{A}{B}{\coeff}` computes $\sum_{n=A}^{n=B} \text{\coeff{n}}$ where `\coeff{n}` must *f*-expand to a (possibly long) integer in the strict format.
`\def\coeff #1{\xintiTrunc {40}{\xintMON{#1}/#1.5}}%`
 % better:
`\def\coeff #1{\xintiTrunc {40}`
`{\the\numexpr 2*\xintiiMON{#1}\relax/\the\numexpr 2*#1+1\relax [0]}}`
 % better still:
`\def\coeff #1{\xintiTrunc {40}`
`{\the\numexpr \ifodd #1 -2\else 2\fi\relax/\the\numexpr 2*#1+1\relax [0]}}`
 % $(-1)^n/(n+1/2)$ times 10^{40} , truncated to an integer.
`\[\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}} \approx`
`\xintTrunc {40}{\xintiSeries {0}{50}{\coeff}[-40]}\dots\]`

The #1.5 trick to define the `\coeff` macro was neat, but $1/3.5$, for example, turns internally into $10/35$ whereas it would be more efficient to have $2/7$. The second way of coding the wanted coefficient avoids a superfluous factor of five and leads to a faster evaluation. The third way is faster, after all there is no need to use `\xintMON` (or rather `\xintiiMON` which has less parsing overhead) on integers obeying the \TeX bound. The denominator having no sign, we have added the `[0]` as this speeds up (infinitesimally) the parsing.

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n + \frac{1}{2}} \approx 1.5805993064935250412367895069567264144810$$

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\else 2\fi\relax/\the\numexpr 2*#1+1\relax [0]}}`
 % $(-1)^n/(n+1/2)$ times 10^{40} , rounded to an integer.
`\[\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}} \approx`
`\xintTrunc {40}{\xintiSeries {0}{50}{\coeff}[-40]}\]`
`\def\exactcoeff #1%`
`{\the\numexpr \ifodd #1 -2\else 2\fi\relax/\the\numexpr 2*#1+1\relax [0]}}`
`\[\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}}`
`= \xintTrunc {50}{\xintSeries {0}{50}{\exactcoeff}}\dots\]`

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n + \frac{1}{2}} \approx 1.5805993064935250412367895069567264144804$$

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n + \frac{1}{2}} = 1.58059930649352504123678950695672641448068680288367 \dots$$

This shows indeed that our sum of truncated terms estimated wrongly the 39th and 40th digits of the exact result⁶⁴ and that the sum of rounded terms fared a bit better.

31.3 `\xintRationalSeries`

`\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

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


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


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


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


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


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


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


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


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


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


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


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


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


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


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


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


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


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


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


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

```

⁶⁴as the series is alternating, we can roughly expect an error of $\sqrt{40}$ and the last two digits are off by 4 units, which is not contradictory to our expectations.

$$\sum_{n=0}^{20} \frac{2^n}{n!} = 7.389056098930 \dots = \frac{17976849421618118656}{2432902008176640000} = \frac{6857623833199}{9280784638125}$$

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$\sum_{n=0}^{\the\cnta} \frac{(-1)^n}{n!}=
\xintTrunc{20}\z\dotso=\xintFrac{\z}=\xintFrac{\xintIrr\z}$
\vtop to 5pt{}\endgraf
\ifnum\cnta<20 \advance\cnta 1 \repeat

$$\sum_{n=0}^0 \frac{(-1)^n}{n!} = 1.00000000000000000000 \dots = 1 = 1$$


$$\sum_{n=0}^1 \frac{(-1)^n}{n!} = 0 \dots = 0 = 0$$


$$\sum_{n=0}^2 \frac{(-1)^n}{n!} = 0.50000000000000000000 \dots = \frac{1}{2} = \frac{1}{2}$$


$$\sum_{n=0}^3 \frac{(-1)^n}{n!} = 0.33333333333333333333 \dots = \frac{2}{6} = \frac{1}{3}$$


$$\sum_{n=0}^4 \frac{(-1)^n}{n!} = 0.37500000000000000000 \dots = \frac{9}{24} = \frac{3}{8}$$


$$\sum_{n=0}^5 \frac{(-1)^n}{n!} = 0.36666666666666666666 \dots = \frac{44}{120} = \frac{11}{30}$$


$$\sum_{n=0}^6 \frac{(-1)^n}{n!} = 0.36805555555555555555 \dots = \frac{265}{720} = \frac{53}{144}$$


$$\sum_{n=0}^7 \frac{(-1)^n}{n!} = 0.36785714285714285714 \dots = \frac{1854}{5040} = \frac{103}{280}$$


$$\sum_{n=0}^8 \frac{(-1)^n}{n!} = 0.36788194444444444444 \dots = \frac{14833}{40320} = \frac{2119}{5760}$$


$$\sum_{n=0}^9 \frac{(-1)^n}{n!} = 0.36787918871252204585 \dots = \frac{133496}{362880} = \frac{16687}{45360}$$


$$\sum_{n=0}^{10} \frac{(-1)^n}{n!} = 0.36787946428571428571 \dots = \frac{1334961}{3628800} = \frac{16481}{44800}$$


$$\sum_{n=0}^{11} \frac{(-1)^n}{n!} = 0.36787943923360590027 \dots = \frac{14684570}{39916800} = \frac{1468457}{3991680}$$


$$\sum_{n=0}^{12} \frac{(-1)^n}{n!} = 0.36787944132128159905 \dots = \frac{176214841}{479001600} = \frac{16019531}{43545600}$$


$$\sum_{n=0}^{13} \frac{(-1)^n}{n!} = 0.36787944116069116069 \dots = \frac{2290792932}{6227020800} = \frac{63633137}{172972800}$$


$$\sum_{n=0}^{14} \frac{(-1)^n}{n!} = 0.36787944117216190628 \dots = \frac{32071101049}{87178291200} = \frac{2467007773}{6706022400}$$


$$\sum_{n=0}^{15} \frac{(-1)^n}{n!} = 0.36787944117139718991 \dots = \frac{481066515734}{1307674368000} = \frac{34361893981}{93405312000}$$


$$\sum_{n=0}^{16} \frac{(-1)^n}{n!} = 0.36787944117144498468 \dots = \frac{7697064251745}{20922789888000} = \frac{15549624751}{42268262400}$$


$$\sum_{n=0}^{17} \frac{(-1)^n}{n!} = 0.36787944117144217323 \dots = \frac{130850092279664}{355687428096000} = \frac{8178130767479}{22230464256000}$$


$$\sum_{n=0}^{18} \frac{(-1)^n}{n!} = 0.36787944117144232942 \dots = \frac{2355301661033953}{6402373705728000} = \frac{138547156531409}{376610217984000}$$


$$\sum_{n=0}^{19} \frac{(-1)^n}{n!} = 0.36787944117144232120 \dots = \frac{44750731559645106}{121645100408832000} = \frac{92079694567171}{250298560512000}$$


$$\sum_{n=0}^{20} \frac{(-1)^n}{n!} = 0.36787944117144232161 \dots = \frac{895014631192902121}{2432902008176640000} = \frac{4282366656425369}{11640679464960000}$$

```

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
```

[illegible]

Observe that in this last example the `x` was directly inserted; if it had been a more complicated explicit fraction it would have been worthwhile 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

$$\sum_{n=\text{the}\cnta}^{\text{the}\numexpr 2*\cnta-1\relax} \frac{\text{the}\cnta^n \{n!\}}{\sum_{n=0}^{\text{the}\numexpr 2*\cnta-1\relax} \frac{\text{the}\cnta^n \{n!\}}{\xintTrunc{8}{\xintDiv{z}{w}}\dots} \text{vtop to 5pt}} =$$

\ifnum\cnta<20 \advance\cnta 1 \repeat

```

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

31.4 `\xintRationalSeriesX`

$\frac{\text{num}}{x} \frac{\text{num}}{x} \frac{\text{Frac}}{f} \frac{\text{Frac}}{f} f \star$

`\xintRationalSeriesX{A}{B}{\first}{\ratio}{\g}` is a parametrized version of `\xintRationalSeries` where `\first` is now a one-parameter macro such that `\first{\g}` gives the initial term and `\ratio` is a two-parameter macro such that `\ratio{n}{\g}` represents the ratio of one term to the previous one. The parameter `\g` is evaluated only once at the beginning of the computation, and can thus itself be the yet unevaluated result of a previous computation.

Let `\ratio` be such a two-parameter macro; note the subtle differences between

```
\xintRationalSeries {A}{B}{\first}{\ratio{\g}}
```

```
and \xintRationalSeriesX {A}{B}{\first}{\ratio}{\g}.
```

First the location of braces differ... then, in the former case `\first` is a *no-parameter* macro expanding to a fractional number, and in the latter, it is a *one-parameter* macro which will use `\g`. Furthermore the X variant will expand `\g` at the very beginning whereas the former non-X former variant will evaluate it each time it needs it (which is bad if this evaluation is time-costly, but good if `\g` is a big explicit fraction encapsulated in a macro).

The example will use the macro `\xintPowerSeries` which computes efficiently exact partial sums of power series, and is discussed in the next section.

```
\def\firstterm #1{1[0]}% first term of the exponential series
% although it is the constant 1, here it must be defined as a
% one-parameter macro. Next comes the ratio function for exp:
\def\ratioexp #1#2{\xintDiv {#1}{#2}}% x/n
% These are the (-1)^(n-1)/n of the log(1+h) series:
\def\coefflog #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}%
% Let L(h) be the first 10 terms of the log(1+h) series and
% let E(t) be the first 10 terms of the exp(t) series.
% The following computes E(L(a/10)) for a=1,...,12.
\cnta 0
\loop
\noindent\xintTrunc {18}{%
  \xintRationalSeriesX {0}{9}{\firstterm}{\ratioexp}
  {\xintPowerSeries{1}{10}{\coefflog}{\the\cnta[-1]}}\dots
}\endgraf
\ifnum\cnta < 12 \advance \cnta 1 \repeat
```

```
1.0999999999999083906... 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
56308534427154141471013807206588202981046013155342233701289165089056
83005693656447898877952000000000/39594086612242519324387557078266845
776303882240000000000000000000[-90] (length of numerator: 155)
```

```
E(L(12[-2]))=443453770054417465442109252347264824711893599160411729
60388258419808415322610807070750589009628030597103713328020346412371
55887714188380658982959014134632946402759999397422009303463626532643
```

5417048639843167445553122713679545984140443648000000000/395940866122
425193243875570782668457763038822400000000000000000000[-180] (length of
numerator: 245)

E(L(123[-3]))=44464159265194177715425414884885486619895497155261639
00742959135317921138508647797623508008144169817627741486630524932175
66759754097977420731516373336789722730765496139079185229545102248282
39119962102923779381174012211091973543316113275716895586401771088185
05853950798598438316179662071953915678034718321474363029365556301004
8000000000/395940866122425193243875570782668457763038822400000000000
00000000[-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))=518138516117322604916074833164833344883840590133006168125
12534667430913353255394804713669158571590044976892591448945234186435
1924224000000000/453371201621089791788096627821377652892232653817581
52546654836095087089601022689942796465342115407786358809263904208715
7760000000000000000000[0] (length of numerator: 141; length of denominator: 141)

E(L(1/71))=16479948917721955649802595580610709825615810175620936986
46571522821497800830677980391753251868507166092934678546038421637547
16919123274624394132188208895310089982001627351524910000588238596565
3808879162861533474038814343168000000000/162510607383091507102283159
26583043448560635097998286551792304600401711584442548604911127392639
47128502616674265101594835449174751466360330459637981998261154868149
5538153647264137927630891689041426777132144944742400000000000000000
0[0] (length of numerator: 232; length of denominator: 232)

E(L(1/712))=2096231738801631206754816378972162002839689022482032389
43136902264182865559717266406341976325767001357109452980607391271438
07919507395930152825400608790815688812956752026901171545996915468879
90896257382714338565353779187008849807986411970218551170786297803168
35353043067415753497212012899985019017494798220551782400000000/2093
29172233767379973271986231161997566292788454774484652603429574146596
81775830937864120504809583013570752212138965469030119839610806057249
0342602456343055829220334691330984419090140201839416227006587667057
5550330002721292096217682473000829618103432600036119035084894266166
64834303221920647163859173376000000000000000000000 [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.⁶⁵ 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

⁶⁵putting 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 `\xintFxFtPowerSeries` for fixed-point computations.

Update: release 1.08a of *xintseries* now includes a tentative naive `\xintFloatPowerSeries`.

31.5 `\xintPowerSeries`

`\xintPowerSeries{A}{B}{\coeff}{f}` evaluates the sum $\sum_{n=A}^{n=B} \text{\coeff}\{n\} \cdot f^n$. 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.⁶⁶

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

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

⁶⁶with powers f^k , from $k=0$ to N , a denominator d of f became $d^{1+2+\dots+N}$, 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\fr {1/2[0]}%
\[ \log 2 \approx \sum_{n=1}^{20} \frac{1}{n \cdot 2^n}
= \xintFrac {\xintIrr {\xintPowerSeries {1}{20}{\coefflog}{\fr}}}{\fr}
\[ \log 2 \approx \sum_{n=1}^{50} \frac{1}{n \cdot 2^n}
= \xintFrac {\xintIrr {\xintPowerSeries {1}{50}{\coefflog}{\fr}}}{\fr}

```

$$\log 2 \approx \sum_{n=1}^{20} \frac{1}{n \cdot 2^n} = \frac{42299423848079}{61025172848640}$$

$$\log 2 \approx \sum_{n=1}^{50} \frac{1}{n \cdot 2^n} = \frac{60463469751752265663579884559739219}{87230347965792839223946208178339840}$$

```

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

```

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

31.6 `\xintPowerSeriesX`

$$\frac{\frac{\text{num } x}{x} \frac{\text{num } x}{x}}{f} \frac{\text{Frac } f}{f}$$

This is the same as `\xintPowerSeries` apart from the fact that the last parameter `f` is expanded once and for all before being then used repeatedly. If the `f` parameter is to be an explicit big fraction with many (dozens) digits, rather than using it directly it is slightly better to have some macro `\g` defined to expand to the explicit fraction and then use `\xintPowerSeries` with `\g`; but if `f` has not yet been evaluated and will be the output of a complicated expansion of some `\f`, and if, due to an expanding only context, doing `\edef \g{\f}` is no option, then `\xintPowerSeriesX` should be used with `\f` as last parameter.

```
\def\xratioexp #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...
0.1999999995263443554...	0.593980619762352217...	-12.648937932093322763...
0.299999338075041781...	0.645144282733914916...	-66.259639046914679687...
0.399974460740121112...	0.398118280111436442...	-304.768437445462801227...

31.7 `\xintFxFtPowerSeries`

$$\frac{\frac{\text{num } x}{f} \frac{\text{num } x}{f}}{\frac{\text{Frac } f}{f}} \frac{\text{Frac } f}{f} \frac{\text{num } x}{x} \star$$

`\xintFxFtPowerSeries{A}{B}{\coeff}{f}{D}` computes $\sum_{n=A}^{n=B} \text{coeff}\{n\} \cdot f^n$ with each term of the series truncated to `D` digits after the decimal point. As usual, `A` and `B` are completely expanded through their inclusion in a `\numexpr` expression. Regarding `D` it will be similarly be expanded each time it is used inside an `\xintTrunc`. The one-parameter macro `\coeff` is similarly expanded at the time it is used inside the computations. Idem for `f`. If `f` itself is some complicated macro it is thus better to use the variant `\xintFxFtPowerSeriesX` 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 $\text{coeff}\{n\} \cdot f^n$ is obtained from that by multiplying by $\text{coeff}\{n\}$ (untruncated) and then truncating. Finally the sum is computed exactly. Apart from that `\xintFxFtPowerSeries` (where `FxFt` 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 `\xintFxFtPowerSeries` 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{\xintFac {#1}[0]}% 1/n!
```

```
\def\ f {-1/2[0]}% [0] for faster input parsing
```

```
\cnta 0 % previously declared \count register
```

```
\noindent\loop
```

```
\xintFxFtPowerSeries {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:

```
\xintFxFtPowerSeries {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:

$$\begin{aligned} \text{\xintPowerSeries } \{0\}{19}\{\text{\coeffexp}\}\{\text{\ f}\} &= \frac{38682746160036397317757}{63777066403145711616000} \\ &= 0.606530659712633423603799152126\dots \end{aligned}$$

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{N}
 to get the number k of digits possibly of dubious significance.

31.8 **\xintFxFtPowerSeriesX**

$\frac{\frac{x}{f}}{\frac{x}{f}}$ ★

\xintFxFtPowerSeriesX{A}{B}{\coeff}{\ f}{D} computes, exactly as **\xintFxFt-**
PowerSeries, the sum of $\text{\coeff}\{n\} \cdot \text{\ f}^n$ 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 {\xintFxFtPowerSeriesX {1}{10}{\coefflog}{\the\cnta [-2]}}{5}}
        {\xintFxFtPowerSeriesX {1}{10}{\coefflog}
          {\xintFxFtPowerSeriesX {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 {\xintFxFtPowerSeriesX {1}{15}{\coefflog}{\the\cnta [-2]}}{6}}
        {\xintFxFtPowerSeriesX {1}{15}{\coefflog}
          {\xintRound {4}{\xintFxFtPowerSeriesX {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 `\xintFxFtPowerSeriesX` 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.

31.9 `\xintFloatPowerSeries`

$\left[\begin{smallmatrix} \text{num} \\ x \\ \text{Frac} \end{smallmatrix} \right] \begin{smallmatrix} \text{num} \\ x \\ \text{Frac} \end{smallmatrix} \begin{smallmatrix} \text{num} \\ x \\ \text{Frac} \end{smallmatrix}$
 $f \quad f \quad \star$

`\xintFloatPowerSeries[P]{A}{B}{\coeff}{f}` computes $\sum_{n=A}^{n=B} \text{coeff}\{n\} \cdot f^n$ with a floating point precision given by the optional parameter P or by the current setting of `\xintDigits`.

In the current, preliminary, version, no attempt has been made to try to guarantee to the final result the precision P. Rather, P is used for all intermediate floating point evaluations. So rounding errors will make some of the last printed digits invalid. The operations done are first the evaluation of f^A using `\xintFloatPow`, then each successive power

is obtained from this first one by multiplication by f using `\xintFloatMul`, then again with `\xintFloatMul` this is multiplied with `\coeff{n}`, and the sum is done adding one term at a time with `\xintFloatAdd`. To sum up, this is just the naive transformation of `\xintFxFtPowerSeries` 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
```

31.10 `\xintFloatPowerSeriesX`

$\left[\begin{smallmatrix} \text{num} & \text{num} & \text{num} \\ x & \text{Frac} & \text{Frac} \\ f & f & \end{smallmatrix} \right] x$ ★

`\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
```

31.11 Computing $\log 2$ and π

In this final section, the use of `\xintFxFtPowerSeries` (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 fortieth 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 `\xintFxFtPowerSeries`: this is worthwhile 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}{\xintFxFtPowerSeries {1}{#2}{\coefflog}{\xa}{#3}}}
{\xintMul {5}{\xintFxFtPowerSeries {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
 $\log 2 \approx 0.693147180559945309417232121458176568075500134360255254120484\dots$ 
 $\approx 0.693147180559945309417232121458176568075500134360255254120680$ 
 $00711\dots$ 
 $\approx 0.693147180559945309417232121458176568075500134360255254120680$ 
 $0094933723\dots$ 

```

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 `\xintFxFtPowerSeries`.

```

\def\LogTwo #1% get  $\log(2) = -2\log(1-13/256) - 5\log(1-1/9)$ 
{%
\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}{\xintFxFtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}}
    {\xintMul {4/239}{\xintFxFtPowerSeries {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

```

	3.141592653589793
3.	3.1415926535897932
3.1	3.14159265358979323
3.14	3.141592653589793238
3.141	3.1415926535897932384
3.1415	3.14159265358979323846
3.14159	3.141592653589793238462
3.141592	3.1415926535897932384626
3.1415926	3.14159265358979323846264
3.14159265	3.141592653589793238462643
3.141592653	3.1415926535897932384626433
3.1415926535	3.14159265358979323846264338
3.14159265358	3.141592653589793238462643383
3.141592653589	3.1415926535897932384626433832
3.1415926535897	3.14159265358979323846264338327
3.14159265358979	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 `\xintFxFtPowerSeries` and `\xintFxFtPowerSeriesX`? To round at D digits, and excluding a rewrite or cloning of the division algo-

rithm 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 negligible 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.

32 Commands of the **xintcfrac** package

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

Contents

.1	Package overview	110	.13	<code>\xintCstoGC</code>	121
.2	<code>\xintCFrac</code>	117	.14	<code>\xintGctoF</code>	121
.3	<code>\xintGCfrac</code>	118	.15	<code>\xintGctoCv</code>	121
.4	<code>\xintGctoGCx</code>	118	.16	<code>\xintCntoF</code>	122
.5	<code>\xintFtoCs</code>	118	.17	<code>\xintGCntoF</code>	122
.6	<code>\xintFtoCx</code>	118	.18	<code>\xintCntoCs</code>	122
.7	<code>\xintFtoGC</code>	119	.19	<code>\xintCntoGC</code>	123
.8	<code>\xintFtoCC</code>	119	.20	<code>\xintGCntoGC</code>	123
.9	<code>\xintFtoCv</code>	119	.21	<code>\xintiCstoF</code> , <code>\xintiGctoF</code> ,	
.10	<code>\xintFtoCCv</code>	119		<code>\xintiCstoCv</code> , <code>\xintiGctoCv</code>	124
.11	<code>\xintCstoF</code>	120	.22	<code>\xintGctoGC</code>	124
.12	<code>\xintCstoCv</code>	120			

32.1 Package overview

A *simple* continued fraction has coefficients $[c_0, c_1, \dots, c_N]$ (usually called partial quotients, but I really dislike this entrenched terminology), where c_0 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 $c:n \rightarrow c_n$. Note that the index then starts at zero as indi-

cated. With the `amsmath` macro `\cfrac` one can display such a continued fraction as

$$c_0 + \frac{1}{c_1 + \frac{1}{c_2 + \frac{1}{c_3 + \frac{1}{\ddots}}}}$$

Here is a concrete example:

$$\frac{208341}{66317} = 3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1 + \frac{1}{292 + \frac{1}{2}}}}}$$

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:

$$\frac{915286}{188421} = 5 - \frac{1}{7 + \frac{1}{39 - \frac{1}{53 - \frac{1}{13}}}} = 4 + \frac{1}{1 + \frac{1}{6 + \frac{1}{38 + \frac{1}{1 + \frac{1}{51 + \frac{1}{1 + \frac{1}{12}}}}}}}$$

```
\[ \xintFrac {915286/188421}=\xintGCFrac {\xintFtoCC {915286/188421}} \]
```

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 $\frac{915286}{188421}$. 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:

$$a_0 + b_0/a_1 + b_1/a_2 + b_2/\dots/a_{n-1} + b_{n-1}/a_n$$

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}}`

$$\frac{1907}{1902} = 1 - \frac{1}{57 - \frac{2187}{5}}$$

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}}`

$$\frac{-28077}{4108} = -7 + \frac{1}{6 + \frac{1}{19 + \frac{1}{1 + \frac{1}{33}}}}$$

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`:

$$\begin{aligned} 2721/1001 &= \text{\xintFtoCC } \{2721/1001\} \\ 2721/1001 &= 3 + -1/4 + -1/2 + 1/5 + -1/2 + 1/7 + -1/2 \end{aligned}$$

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:

$$\text{\xintGctoF } \{143+1/2+\dots+-1/9\}=2897319801297630107/20197107104701740$$

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+\dots+-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}}}\$`

$$\frac{915286}{188421} \rightarrow 4, 5, \frac{34}{7}, \frac{1297}{267}, \frac{1331}{274}, \frac{69178}{14241}, \frac{70509}{14515}, \frac{915286}{188421}$$

`$$\xintFrac{915286/188421}\to \xintListWithSep {,}%
{\xintApply{\xintFrac}{\xintFtoCCv{915286/188421}}}\$`

$$\frac{915286}{188421} \rightarrow 5, \frac{34}{7}, \frac{1331}{274}, \frac{70509}{14515}, \frac{915286}{188421}$$

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:

$\frac{49171}{18089} \rightarrow 2 = [2], 3 = [3], \frac{8}{3} = [2, 1, 2], \frac{11}{4} = [2, 1, 3], \frac{19}{7} = [2, 1, 2, 2], \frac{87}{32} = [2, 1, 2, 1, 1, 4],$
 $\frac{106}{39} = [2, 1, 2, 1, 1, 5], \frac{193}{71} = [2, 1, 2, 1, 1, 4, 2], \frac{1264}{465} = [2, 1, 2, 1, 1, 4, 1, 1, 6], \frac{1457}{536} =$
 $[2, 1, 2, 1, 1, 4, 1, 1, 7], \frac{2721}{1001} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 2], \frac{23225}{8544} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8],$
 $\frac{49171}{18089} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 2].$

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
\[\xintFrac{\xintCnToF {6}{\cn}}=\xintCFrac [1]{\xintCnToF {6}{\cn}}\]
```

$$\frac{3541373}{2449193} = 1 + \frac{1}{2 + \frac{1}{4 + \frac{1}{8 + \frac{1}{16 + \frac{1}{32 + \frac{1}{64}}}}}}$$

Notice the use of the optional argument [1] to `\xintCFrac`. Other possibilities are [r] and (default) [c].

```
\def\cn #1{\xintiPow {2}{-#1}}% 1/2^n
\[\xintFrac{\xintCnToF {6}{\cn}} = \xintGCFrac [r]{\xintCnToGC {6}{\cn}}
= [\xintFtoCs {\xintCnToF {6}{\cn}}]\]
```

$$\frac{3159019}{2465449} = 1 + \frac{1}{\frac{1}{2} + \frac{1}{\frac{1}{4} + \frac{1}{\frac{1}{8} + \frac{1}{\frac{1}{16} + \frac{1}{\frac{1}{32} + \frac{1}{64}}}}}} = [1, 3, 1, 1, 4, 14, 1, 1, 1, 1, 79, 2, 1, 1, 2]$$

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 :

$$\frac{92736}{29520} = \frac{4}{1 + \frac{1}{3 + \frac{4}{5 + \frac{9}{7 + \frac{16}{9 + \frac{25}{11}}}}}} = 3.1414634146\dots$$

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\}}

```

$$\frac{208341}{66317} = 3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1 + \frac{1}{292 + \frac{1}{1 + \frac{1}{1}}}}}} = 3.1415926534\dots$$

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. $2.00000000000000000000000000000000 \dots = 2$
2. $3.00000000000000000000000000000000 \dots = 3$
3. $2.66666666666666666666666666666666 \dots = \frac{8}{3}$
4. $2.75000000000000000000000000000000 \dots = \frac{11}{4}$
5. $2.714285714285714285714285714285 \dots = \frac{19}{7}$
6. $2.71875000000000000000000000000000 \dots = \frac{87}{32}$
7. $2.717948717948717948717948717948 \dots = \frac{106}{39}$
8. $2.718309859154929577464788732394 \dots = \frac{193}{71}$
9. $2.718279569892473118279569892473 \dots = \frac{1264}{465}$
10. $2.718283582089552238805970149253 \dots = \frac{1457}{536}$
11. $2.718281718281718281718281718281 \dots = \frac{2721}{1001}$
12. $2.718281835205992509363295880149 \dots = \frac{23225}{8544}$
13. $2.718281822943949711891042430591 \dots = \frac{25946}{9545}$
14. $2.718281828735695726684725523798 \dots = \frac{49171}{18089}$
15. $2.718281828445401318035025074172 \dots = \frac{517656}{190435}$
16. $2.718281828470583721777828930962 \dots = \frac{566827}{208524}$
17. $2.718281828458563411277850606202 \dots = \frac{1084483}{398959}$
18. $2.718281828459065114074529546648 \dots = \frac{13580623}{4996032}$
19. $2.718281828459028013207065591026 \dots = \frac{14665106}{5394991}$
20. $2.718281828459045851404621084949 \dots = \frac{28245729}{10391023}$
21. $2.718281828459045213521983758221 \dots = \frac{410105312}{150869313}$
22. $2.718281828459045254624795027092 \dots = \frac{438351041}{161260336}$
23. $2.718281828459045234757560631479 \dots = \frac{848456353}{312129649}$
24. $2.718281828459045235379013372772 \dots = \frac{14013652689}{5155334720}$
25. $2.718281828459045235343535532787 \dots = \frac{14862109042}{5467464369}$
26. $2.718281828459045235360753230188 \dots = \frac{28875761731}{10622799089}$

27. $2.718281828459045235360274593941 \dots = \frac{534625820200}{196677847971}$
28. $2.718281828459045235360299120911 \dots = \frac{563501581931}{207300647060}$
29. $2.718281828459045235360287179900 \dots = \frac{1098127402131}{403978495031}$
30. $2.718281828459045235360287478611 \dots = \frac{22526049624551}{8286870547680}$
31. $2.718281828459045235360287464726 \dots = \frac{23624177026682}{8690849042711}$
32. $2.718281828459045235360287471503 \dots = \frac{46150226651233}{16977719590391}$
33. $2.718281828459045235360287471349 \dots = \frac{1038929163353808}{382200680031313}$
34. $2.718281828459045235360287471355 \dots = \frac{1085079390005041}{399178399621704}$
35. $2.718281828459045235360287471352 \dots = \frac{212400855358849}{781379079653017}$
36. $2.718281828459045235360287471352 \dots = \frac{52061284670617417}{19152276311294112}$

The actual computation of the list of all 36 convergents accounts for only 8% of the total time (total time equal to about 5 hundredths of a second in my testing, on my laptop): another 80% is occupied with the computation of the truncated decimal expansions (and the addition of 1 to everything as the formula gives the continued fraction of $e - 1$). One can with no problem compute much bigger convergents. Let's get the 200th convergent. It turns out to have the same first 268 digits after the decimal point as $e - 1$. Higher convergents get more and more digits in proportion to their index: the 500th convergent already gets 799 digits correct! To allow speedy compilation of the source of this document when the need arises, I limit here to the 200th convergent (getting the 500th took about 1.2s on my laptop last time I tried, and the 200th convergent is obtained ten times faster).

```
\edef\z {\xintCtoF {199}{\cn}}%
\begingroup\parindent 0pt \leftskip 2.5cm
\indent\llap {Numerator = }\printnumber{\xintNumerator\z}\par
\indent\llap {Denominator = }\printnumber{\xintDenominator\z}\par
\indent\llap {Expansion = }\printnumber{\xintTrunc{268}\z}\dots
\par\endgroup
Numerator = 56896403887189626759752389231580787529388901766791744605
72320245471922969611182301752438601749953108177313670124
1708609749634329382906
Denominator = 33112381766973761930625636081635675336546882372931443815
62056154632466597285818654613376920631489160195506145705
9255337661142645217223
Expansion = 1.718281828459045235360287471352662497757247093699959574
96696762772407663035354759457138217852516642742746639193
20030599218174135966290435729003342952605956307381323286
27943490763233829880753195251019011573834187930702154089
1499348841675092447614606680822648001684774118...
```

One can also use a centered continued fraction: we get more digits but there are also more computations as the numerators may be either 1 or -1 .

32.2 `\xintCFrac`

$\frac{f}{f}$ ★

`\xintCFrac{f}` is a math-mode only, \LaTeX with `amsmath` only, macro which first com-

puts 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 `[1]`, `[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 **xintcf** package.

32.3 `\xintGCFrac`

f ★ `\xintGCFrac{a+b/c+d/e+f/g+h/...}` uses similarly `\cfrac` to typeset a generalized continued fraction in inline format. It admits the same optional argument as `\xintCFrac`.

`\[\xintGCFrac {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}}\]`

$$1 + \frac{3375 \cdot 10^{-3}}{\frac{1}{7} - \frac{\frac{3}{5}}{720}}$$

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.

32.4 `\xintGctoGCx`

nnf ★ `\xintGctoGCx{sepa}{sepb}{a+b/c+d/e+f/...+x/y}` returns the list of the coefficients of the generalized continued fraction of `f`, each one within a pair of braces, and separated with the help of `sepa` and `sepb`. Thus

`\xintGctoGCx :;{1+2/3+4/5+6/7}` gives `1:2;3:4;5:6;7`

Plain \TeX +`amstex` users may be interested in:

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

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

32.5 `\xintFtoCs`

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

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

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

32.6 `\xintFtoCx`

n Frac f ★ `\xintFtoCx{sep}{f}` returns the list of the coefficients of the simple continued fraction of `f`, withing group braces and separated with the help of `sep`.

`$$\xintFtoCx {+\cfrac1\\ }{f}\endcfrac$$`

will display the continued fraction in `\cfrac` format, with Plain \TeX and `amstex`.

32.7 \xintFtoGC

$\frac{f}{f}$ ★ `\xintFtoGC{f}` does the same as `\xintFtoCx{+1/}{f}`. Its output may thus be used in the package macros expecting such an ‘inline format’. This continued fraction is a *simple* one, not a *generalized* one, but as it is produced in the format used for user input of generalized continued fractions, the macro was called `\xintFtoGC` rather than `\xintFtoC` for example.

```
566827/208524=\xintFtoGC {566827/208524}
566827/208524=2+1/1+1/2+1/1+1/1+1/4+1/1+1/1+1/6+1/1+1/1+1/8+1/1+1/1+1/11
```

32.8 \xintFtoCC

$\frac{f}{f}$ ★ `\xintFtoCC{f}` returns the ‘centered’ continued fraction of f , in ‘inline format’.

```
566827/208524=\xintFtoCC {566827/208524}
566827/208524=3+-1/4+-1/2+1/5+-1/2+1/7+-1/2+1/9+-1/2+1/11
\[\xintFrac{566827/208524} = \xintGCfrac{\xintFtoCC{566827/208524}}{\}
```

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

32.9 \xintFtoCv

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

```
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintFtoCv{5211/3748}}{\}]
1 \rightarrow \frac{3}{2} \rightarrow \frac{4}{3} \rightarrow \frac{7}{5} \rightarrow \frac{25}{18} \rightarrow \frac{32}{23} \rightarrow \frac{57}{41} \rightarrow \frac{317}{228} \rightarrow \frac{374}{269} \rightarrow \frac{691}{497} \rightarrow \frac{5211}{3748}
```

32.10 \xintFtoCCv

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

```
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintFtoCCv{5211/3748}}{\}]
1 \rightarrow \frac{4}{3} \rightarrow \frac{7}{5} \rightarrow \frac{32}{23} \rightarrow \frac{57}{41} \rightarrow \frac{374}{269} \rightarrow \frac{691}{497} \rightarrow \frac{5211}{3748}
```

32.11 \xintCstoF

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

```
\[\xintGCFrac {-1+1/3+1/-5+1/7+1/-9+1/11+1/-13}
=\xintSignedFrac{\xintCstoF {-1,3,-5,7,-9,11,-13}}
=\xintSignedFrac{\xintGctoF {-1+1/3+1/-5+1/7+1/-9+1/11+1/-13}}\]
```

$$-1 + \frac{1}{3 + \frac{1}{-5 + \frac{1}{7 + \frac{1}{-9 + \frac{1}{11 + \frac{1}{-13}}}}}} = -\frac{75887}{118187} = -\frac{75887}{118187}$$

```
\xintGCFrac{{1/2}+1/{1/3}+1/{1/4}+1/{1/5}}=
\xintFrac{\xintCstoF {1/2,1/3,1/4,1/5}}
```

$$\frac{1}{2} + \frac{1}{\frac{1}{\frac{1}{\frac{1}{3} + \frac{1}{\frac{1}{\frac{1}{4} + \frac{1}{\frac{1}{5}}}}}}} = \frac{159}{66}$$

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).

32.12 \xintCstoCv

f★ `\xintCstoCv{a,b,c,d,...,z}` returns the list of the corresponding convergents. It is allowed to use fractions as coefficients (the computed convergents have then no reason to be the real convergents of the final fraction). When the coefficients are integers, the convergents are irreducible fractions, but otherwise it is not necessarily the case.

```
\xintListWithSep:{\xintCstoCv{1,2,3,4,5,6}}
1/1:3/2:10/7:43/30:225/157:1393/972
\xintListWithSep:{\xintCstoCv{1,1/2,1/3,1/4,1/5,1/6}}
1/1:3/1:9/7:45/19:225/159:1575/729
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintCstoCv
{\xintPow {-3}{-5},7.3/4.57,\xintCstoF{3/4,9,-1/3}}}}\]
```

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

32.13 \xintCstoGC

f ★ `\xintCstoGC{a,b,...,z}` transforms a comma separated list (or something expanding to such a list) into an ‘inline format’ continued fraction $\{a\}+1/\{b\}+1/\dots+1/\{z\}$. The coefficients are just copied and put within braces, without expansion. The output can then be used in `\xintGCFrac` for example.

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

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

32.14 \xintGctoF

f ★ `\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.

```
\[\xintGCFrac {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}} =
\xintFrac{\xintGctoF {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}}} =
\xintFrac{\xintIrr{\xintGctoF
{1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}}}}\]
```

$$1 + \frac{3375 \cdot 10^{-3}}{\frac{1}{7} - \frac{\frac{3}{5}}{720}} = \frac{88629000}{3579000} = \frac{29543}{1193}$$

```
\[ \xintGCFrac{{1/2}+{2/3}/{4/5}+{1/2}/{1/5}+{3/2}/{5/3}} =
\xintFrac{\xintGctoF {{1/2}+{2/3}/{4/5}+{1/2}/{1/5}+{3/2}/{5/3}}} \]
```

$$\frac{1}{2} + \frac{\frac{2}{3}}{\frac{\frac{4}{5} + \frac{\frac{1}{2}}{\frac{3}{\frac{1}{5} + \frac{2}{3}}}}}{4140} = \frac{4270}{4140}$$

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.

32.15 \xintGctoCv

f ★ `\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`.

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

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

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

32.16 `\xintCntoF`

$\overset{\text{num}}{x} f \star$ `\xintCntoF{N}{\macro}` computes the fraction f having coefficients $c(j)=\macro{j}$ for $j=0, 1, \dots, N$. 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]
```

32.17 `\xintGCntoF`

$\overset{\text{num}}{x} ff \star$ `\xintGCntoF{N}{\macroA}{\macroB}` returns the fraction f corresponding to the inline generalized continued fraction $a_0+b_0/a_1+b_1/a_2+\dots+b(N-1)/a_N$, with $a(j)=\macroA{j}$ and $b(j)=\macroB{j}$. The N parameter is given to a `\numexpr`.

$$1 + \frac{1}{2 - \frac{1}{3 + \frac{1}{1 - \frac{1}{2 + \frac{1}{3 - \frac{1}{1}}}}}} = \frac{39}{25}$$

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

32.18 `\xintCntoCs`

$\overset{\text{num}}{x} f \star$ `\xintCntoCs{N}{\macro}` produces the comma separated list of the corresponding coefficients, from $n=0$ to $n=N$. The N is given to a `\numexpr`.

```
\def\macro #1{\the\numexpr 1+#1*#1\relax}
```

```
\xintCtoCs {5}{\macro}->1,2,5,10,17,26
\[\xintFrac{\xintCtoF {5}{\macro}}=\xintCFrac{\xintCtoF {5}{\macro}}\]
```

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

32.19 \xintCtoGC

$\overset{\text{num}}{x}f \star$ `\xintCtoGC{N}{\macro}` evaluates the $c(j)=\macro{j}$ from $j=0$ to $j=N$ and returns a continued fraction written in inline format: $\{c(0)\}+1/\{c(1)\}+1/\dots+1/\{c(N)\}$. 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{\xintCtoGC {5}{\macro}}\meaning\x
macro:->\{1/1\}+1/\{-2/2\}+1/\{3/5\}+1/\{-4/10\}+1/\{5/17\}+1/\{-6/26\}
\[\xintGCFrac{\xintCtoGC {5}{\macro}}\]
```

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

32.20 \xintGCtoGC

$\overset{\text{num}}{x}ff \star$ `\xintGCtoGC{N}{\macroA}{\macroB}` evaluates the coefficients and then returns the corresponding $\{a_0\}+\{b_0\}/\{a_1\}+\{b_1\}/\{a_2\}+\dots+\{b_{(N-1)}\}/\{a_N\}$ inline generalized fraction. N is given to a `\numexpr`. As shown, the coefficients are enclosed into added pairs of braces, and may thus be fractions.

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

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

32.21 `\xintiCstoF`, `\xintiGctoF`, `\xintiCstoCv`, `\xintiGctoCv`

The same as the corresponding macros without the ‘i’, but for integer-only input. Infinitely faster; to notice the higher efficiency one would need to use them with an input having (at least) hundreds of coefficients.

32.22 `\xintGctoGC`

f ★ `\xintGctoGC{a+b/c+d/e+f/g+.....+v/w+x/y}` expands (with the usual meaning) each one of the coefficients and returns an inline continued fraction of the same type, each expanded coefficient being enclosed withing braces.

`{6}+\xintCstoF {2,-7,-5}/16}}` `\meaning\x`
`macro:->{1}+{3375/1[-3]}/{1/7}+{-3/5}/{720}+{67/36}/{16}`

To be honest I have, it seems, forgotten why I wrote this macro in the first place.

33 Package **xinttools** implementation

Release 1.09g splits off `xinttools.sty` from `xint.sty`.

Contents

.1	Catcodes, ε -TeX and reload detection ..	125
.2	Package identification	128
.3	Token management, constants	128
.4	<code>\xintReverseOrder</code>	129
.5	<code>\xintRevWithBraces</code>	129
.6	<code>\xintLength</code>	130
.7	<code>\xintZapFirstSpaces</code>	131
.8	<code>\xintZapLastSpaces</code>	133
.9	<code>\xintZapSpaces</code>	134
.10	<code>\xintZapSpacesB</code>	134
.11	<code>\xintCSVtoList</code> , <code>\xintCSVtoList-</code> NonStripped	135
.12	<code>\xintListWithSep</code>	136
.13	<code>\xintNthElt</code>	137
.14	<code>\xintApply</code>	138
.15	<code>\xintApplyUnbraced</code>	139
.16	<code>\xintSeq</code>	139
.17	<code>\xintloop</code> , <code>\xintbreakloop</code> , <code>\xint-</code> <code>breakloopanddo</code> , <code>\xintloopskiptonext</code>	142
.18	<code>\xintiloop</code> , <code>\xintiloopindex</code> , <code>\xintouteriloopindex</code> , <code>\xint-</code> <code>breakiloop</code> , <code>\xintbreakiloopanddo</code> , <code>\xintloopskiptonext</code> , <code>\xintiloop-</code> <code>skipandredo</code>	142
.19	<code>\XINT_xflet</code>	143
.20	<code>\xintApplyInline</code>	144
.21	<code>\xintFor</code> , <code>\xintFor*</code> , <code>\xintBreak-</code> <code>For</code> , <code>\xintBreakForAndDo</code>	145
.22	<code>\XINT_forever</code> , <code>\xintintegers</code> , <code>\xintdimensions</code> , <code>\xintrationals</code>	148
.23	<code>\xintForpair</code> , <code>\xintForthree</code> , <code>\xintForfour</code>	150
.24	<code>\xintAssign</code> , <code>\xintAssignArray</code> , <code>\xintDigitsOf</code>	151

33.1 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 L^AT_EX3.

Release 1.09b is more economical: some macros are defined already in `xint.sty` (now `xinttools.sty`) and re-used in other modules. All catcode changes have been unified and `\XINT_storecatcodes` will be used by each module to redefine `\XINT_restorecatcodes_endinput` in case catcodes have changed in-between the loading of `xint.sty` (now `xinttools.sty`) and the module (not very probable but...).

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2 \catcode13=5 % ^^M
3 \endlinechar=13 %
4 \catcode123=1 % {
5 \catcode125=2 % }
```

33 Package *xinttools* implementation

```

6 \catcode64=11 % @
7 \catcode95=11 % _
8 \catcode35=6 % #
9 \catcode44=12 % ,
10 \catcode45=12 % -
11 \catcode46=12 % .
12 \catcode58=12 % :
13 \expandafter\let\expandafter\x\csname ver@xint.sty\endcsname
14 \expandafter
15 \ifx\csname PackageInfo\endcsname\relax
16 \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
17 \else
18 \def\y#1#2{\PackageInfo{#1}{#2}}%
19 \fi
20 \expandafter
21 \ifx\csname numexpr\endcsname\relax
22 \y{xinttools}{\numexpr not available, aborting input}%
23 \aftergroup\endinput
24 \else
25 \ifx\x\relax % plain-TeX, first loading
26 \else
27 \def\empty {}%
28 \ifx\x\empty % LaTeX, first loading,
29 % variable is initialized, but \ProvidesPackage not yet seen
30 \else
31 \y{xinttools}{I was already loaded, aborting input}%
32 \aftergroup\endinput
33 \fi
34 \fi
35 \fi
36 \def\ChangeCatcodesIfInputNotAborted
37 {%
38 \endgroup
39 \def\XINT_storecatcodes
40 {% takes care of all, to allow more economical code in modules
41 \catcode63=\the\catcode63 % ? xintexpr
42 \catcode124=\the\catcode124 % | xintexpr
43 \catcode38=\the\catcode38 % & xintexpr
44 \catcode64=\the\catcode64 % @ xintexpr
45 \catcode33=\the\catcode33 % ! xintexpr
46 \catcode93=\the\catcode93 % ] -, xintfrac, xintcfrac
47 \catcode91=\the\catcode91 % [ -, xintfrac, xintseries, xintcfrac
48 \catcode36=\the\catcode36 % $ xintgcd only
49 \catcode94=\the\catcode94 % ^
50 \catcode96=\the\catcode96 % '
51 \catcode47=\the\catcode47 % /
52 \catcode41=\the\catcode41 % )
53 \catcode40=\the\catcode40 % (
54 \catcode42=\the\catcode42 % *

```

33 Package *xinttools* implementation

```

55      \catcode43=\the\catcode43    % +
56      \catcode62=\the\catcode62    % >
57      \catcode60=\the\catcode60    % <
58      \catcode58=\the\catcode58    % :
59      \catcode46=\the\catcode46    % .
60      \catcode45=\the\catcode45    % -
61      \catcode44=\the\catcode44    % ,
62      \catcode35=\the\catcode35    % #
63      \catcode95=\the\catcode95    % _
64      \catcode125=\the\catcode125  % }
65      \catcode123=\the\catcode123  % {
66      \endlinechar=\the\endlinechar
67      \catcode13=\the\catcode13    % ^^M
68      \catcode32=\the\catcode32    %
69      \catcode61=\the\catcode61\relax % =
70  }%
71  \edef\XINT_restorecatcodes_endinput
72  {%
73      \XINT_storecatcodes\noexpand\endinput %
74  }%
75  \def\XINT_setcatcodes
76  {%
77      \catcode61=12    % =
78      \catcode32=10    % space
79      \catcode13=5     % ^^M
80      \endlinechar=13  %
81      \catcode123=1    % {
82      \catcode125=2    % }
83      \catcode95=11    % _ (replaces @ everywhere, starting with 1.06b)
84      \catcode35=6     % #
85      \catcode44=12    % ,
86      \catcode45=12    % -
87      \catcode46=12    % .
88      \catcode58=11    % : (made letter for error cs)
89      \catcode60=12    % <
90      \catcode62=12    % >
91      \catcode43=12    % +
92      \catcode42=12    % *
93      \catcode40=12    % (
94      \catcode41=12    % )
95      \catcode47=12    % /
96      \catcode96=12    % '
97      \catcode94=11    % ^
98      \catcode36=3     % $
99      \catcode91=12    % [
100     \catcode93=12    % ]
101     \catcode33=11    % !
102     \catcode64=11    % @
103     \catcode38=12    % &

```

```

104      \catcode124=12 % |
105      \catcode63=11 % ?
106      }%
107      \XINT_setcatcodes
108  }%
109 \ChangeCatcodesIfInputNotAborted
110 \def\XINTsetupcatcodes {% for use by other modules
111      \edef\XINT_restorecatcodes_endinput
112      {%
113          \XINT_storecatcodes\noexpand\endinput %
114      }%
115      \XINT_setcatcodes
116 }%

```

33.2 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 H0 for some reason escaping me (compatibility with LaTeX 2.09 or other things ??) seems to set extra precautions.

1.09c uses e-TeX `\ifdefined`.

```

117 \ifdefined\ProvidesPackage
118   \let\XINT_providespackage\relax
119 \else
120   \def\XINT_providespackage #1#2[#3]%
121       {\immediate\write-1{Package: #2 #3}%
122        \expandafter\xdef\csname ver@#2.sty\endcsname{#3}}%
123 \fi
124 \XINT_providespackage
125 \ProvidesPackage {xinttools}%
126 [2013/11/28 v1.09h Expandable and non-expandable utilities (jfb)]%

```

33.3 Token management, constants

In 1.09e `\xint_undef` replaced everywhere by `\xint_bye`. Release 1.09h makes most everything `\long`.

```

127 \long\def\xint_gobble_      {}%
128 \long\def\xint_gobble_i    #1{}%
129 \long\def\xint_gobble_ii   #1#2{}%
130 \long\def\xint_gobble_iii  #1#2#3{}%
131 \long\def\xint_gobble_iv   #1#2#3#4{}%
132 \long\def\xint_gobble_v    #1#2#3#4#5{}%
133 \long\def\xint_gobble_vi   #1#2#3#4#5#6{}%
134 \long\def\xint_gobble_vii  #1#2#3#4#5#6#7{}%
135 \long\def\xint_gobble_viii #1#2#3#4#5#6#7#8{}%
136 \long\def\xint_firstofone  #1{#1}%
137 \xint_firstofone{\let\XINT_sptoken= } %<- space here!

```

```

138 \long\def\xint_firstoftwo #1#2{#1}%
139 \long\def\xint_secondoftwo #1#2{#2}%
140 \long\def\xint_firstoftwo_andstop #1#2{ #1}%
141 \long\def\xint_secondoftwo_andstop #1#2{ #2}%
142 \def\xint_minus_andstop { -}%
143 \long\def\xint_gob_til_R #1\R {}%
144 \long\def\xint_gob_til_W #1\W {}%
145 \long\def\xint_gob_til_Z #1\Z {}%
146 \long\def\xint_bye #1\xint_bye {}%
147 \let\xint_relax\relax
148 \def\xint_brelax {\xint_relax }%
149 \long\def\xint_gob_til_xint_relax #1\xint_relax {}%
150 \long\def\xint_afterfi #1#2\fi {\fi #1}%
151 \chardef\xint_c_ 0
152 \chardef\xint_c_viii 8
153 \newtoks\XINT_toks

```

33.4 \xintReverseOrder

\xintReverseOrder: does NOT expand its argument.

```

154 \def\xintReverseOrder {\romannumeral0\xintreverseorder }%
155 \long\def\xintreverseorder #1%
156 {%
157   \XINT_rord_main {}#1%
158   \xint_relax
159   \xint_bye\xint_bye\xint_bye\xint_bye
160   \xint_bye\xint_bye\xint_bye\xint_bye
161   \xint_relax
162 }%
163 \long\def\XINT_rord_main #1#2#3#4#5#6#7#8#9%
164 {%
165   \xint_bye #9\XINT_rord_cleanup\xint_bye
166   \XINT_rord_main {#9#8#7#6#5#4#3#2#1}%
167 }%
168 \long\def\XINT_rord_cleanup\xint_bye\XINT_rord_main #1#2\xint_relax
169 {%
170   \expandafter\space\xint_gob_til_xint_relax #1%
171 }%

```

33.5 \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).

As in some other places, 1.09e replaces \Z by \xint_bye, although here it is just for coherence of notation as \Z would be perfectly safe. The reason for \xint_relax, here and in other locations, is in case #1 expands to nothing, the \romannumeral-‘0 must be stopped

```

172 \def\xintRevWithBraces {\romannumeral0\xintrevwithbraces }%
173 \def\xintRevWithBracesNoExpand {\romannumeral0\xintrevwithbracesnoexpand }%
174 \long\def\xintrevwithbraces #1%
175 {%
176     \expandafter\XINT_revwbr_loop\expandafter{\expandafter}%
177     \romannumeral-‘0#1\xint_relax\xint_relax\xint_relax\xint_relax
178         \xint_relax\xint_relax\xint_relax\xint_relax\xint_bye
179 }%
180 \long\def\xintrevwithbracesnoexpand #1%
181 {%
182     \XINT_revwbr_loop }%
183     #1\xint_relax\xint_relax\xint_relax\xint_relax
184     \xint_relax\xint_relax\xint_relax\xint_relax\xint_bye
185 }%
186 \long\def\XINT_revwbr_loop #1#2#3#4#5#6#7#8#9%
187 {%
188     \xint_gob_til_xint_relax #9\XINT_revwbr_finish_a\xint_relax
189     \XINT_revwbr_loop {{#9}{#8}{#7}{#6}{#5}{#4}{#3}{#2}#1}%
190 }%
191 \long\def\XINT_revwbr_finish_a\xint_relax\XINT_revwbr_loop #1#2\xint_bye
192 {%
193     \XINT_revwbr_finish_b #2\R\R\R\R\R\R\R\R #1%
194 }%
195 \def\XINT_revwbr_finish_b #1#2#3#4#5#6#7#8\Z
196 {%
197     \xint_gob_til_R
198         #1\XINT_revwbr_finish_c 8%
199         #2\XINT_revwbr_finish_c 7%
200         #3\XINT_revwbr_finish_c 6%
201         #4\XINT_revwbr_finish_c 5%
202         #5\XINT_revwbr_finish_c 4%
203         #6\XINT_revwbr_finish_c 3%
204         #7\XINT_revwbr_finish_c 2%
205         \R\XINT_revwbr_finish_c 1\Z
206 }%
207 \def\XINT_revwbr_finish_c #1#2\Z
208 {%
209     \expandafter\expandafter\expandafter
210         \space
211     \csname xint_gobble_\romannumeral #1\endcsname
212 }%

```

33.6 \xintLength

`\xintLength` does NOT expand its argument.

1.09g adds the missing `\xintlength`, which was previously called `\XINT_length`, and suppresses `\XINT_Length`

1.06: improved code is roughly 20% faster than the one from earlier versions.

1.09a, \xintnum inserted. 1.09e: \Z->\xint_bye as this is called from \xint-

33 Package *xinttools* implementation

NthElt, and there it was necessary not to use \Z. Later use of \Z and \W perfectly safe here.

```
213 \def\xintLength {\romannumeral0\xintlength}%
214 \long\def\xintlength #1%
215 {%
216   \XINT_length_loop
217   {0}#1\xint_relax\xint_relax\xint_relax\xint_relax
218   \xint_relax\xint_relax\xint_relax\xint_relax\xint_bye
219}%
220 \long\def\XINT_length_loop #1#2#3#4#5#6#7#8#9%
221 {%
222   \xint_gob_til_xint_relax #9\XINT_length_finish_a\xint_relax
223   \expandafter\XINT_length_loop\expandafter {\the\numexpr #1+8\relax}%
224}%
225 \def\XINT_length_finish_a\xint_relax
226   \expandafter\XINT_length_loop\expandafter #1#2\xint_bye
227 {%
228   \XINT_length_finish_b #2\W\W\W\W\W\W\W\Z {#1}%
229}%
230 \def\XINT_length_finish_b #1#2#3#4#5#6#7#8\Z
231 {%
232   \xint_gob_til_W
233   #1\XINT_length_finish_c 8%
234   #2\XINT_length_finish_c 7%
235   #3\XINT_length_finish_c 6%
236   #4\XINT_length_finish_c 5%
237   #5\XINT_length_finish_c 4%
238   #6\XINT_length_finish_c 3%
239   #7\XINT_length_finish_c 2%
240   \W\XINT_length_finish_c 1\Z
241}%
242 \def\XINT_length_finish_c #1#2\Z #3{\expandafter\space\the\numexpr #3-#1\relax}%
```

33.7 \xintZapFirstSpaces

1.09f, written [2013/11/01].

```
243 \def\xintZapFirstSpaces {\romannumeral0\xintzapfirstspaces}%
244 \long\edef\xintzapfirstspaces #1%
245   {\noexpand\XINT_zapbsp_a \space #1\space\space\noexpand\xint_bye\xint_relax}%
246 \xint_firstofone {\long\def\XINT_zapbsp_a #1 }%<- space token here
247 {%
```

If the original #1 started with a space, here #1 will be in fact empty, so the effect will be to remove precisely one space from the original, because the first two

space tokens are matched to the ones of the macro parameter text. If the original #1 did not start with a space then the #1 will be this original #1, with its added first space, up to the first <sp><sp> found. The added initial space will stop later the \romannumeral0. And in \xintZapLastSpaces we also carried along a space in order to be able to mix the two codes in \xintZapSpaces. Testing for \emptiness of #1 is NOT done with an \if test because #1 may contain \if, \fi things (one could use a \detokenize method), and also because xint.sty has a style of its own for doing these things...

```
248 \XINT_zapbsp_again? #1\xint_bye\XINT_zapbsp_b {#1}%
```

The #1 above is thus either empty, or it starts with a (char 32) space token followed with a non (char 32) space token and at any rate #1 is protected from brace stripping. It is assumed that the initial input does not contain space tokens of other than 32 as character code.

```
249 }%
```

```
250 \long\def\XINT_zapbsp_again? #1{\xint_bye #1\XINT_zapbsp_again }%
```

In the "empty" situation above, here #1=\xint_bye, else #1 could be some brace things, but unbracing will anyhow not reveal any \xint_bye. When we do below \XINT_zapbsp_again we recall that we have stripped two spaces out of <sp><original #1>, so we have one <sp> less in #1, and when we loop we better not forget to re-insert one initial <sp>.

```
251 \edef\XINT_zapbsp_again\XINT_zapbsp_b #1{\noexpand\XINT_zapbsp_a\space }%
```

We now have now gotten rid of the initial spaces, but #1 perhaps extend only to some initial chunk which was delimited by <sp><sp>.

```
252 \long\def\XINT_zapbsp_b #1#2\xint_relax
```

```
253 { \XINT_zapbsp_end? #2\XINT_zapbsp_e\empty #2{#1}}%
```

If the initial chunk up to <sp><sp> (after stripping away the first spaces) was maximal, then #2 above is some spaces followed by \xint_bye, so in the \XINT_zapbsp_end? below it appears as \xint_bye, else the #1 below will not be nor give rise after brace removal to \xint_bye. And then the original \xint_bye in #2 will have the effect that all is swallowed and we continue with \XINT_zapbsp_e. If the chunk was maximal, then the #2 above contains as many space tokens as there were originally at the end.

```
254 \long\def\XINT_zapbsp_end? #1{\xint_bye #1\XINT_zapbsp_end }%
```

The #2 starts with a space which stops the \romannumeral. The #1 contains the same number of space tokens there was originally.

```
255 \long\def\XINT_zapbsp_end\XINT_zapbsp_e\empty #1\xint_bye #2{#2#1}%
```

Here the initial chunk was not maximal. So we need to get a second piece all the way up to \xint_bye, we take this opportunity to remove the two initially added ending

33 Package *xinttools* implementation

space tokens. We inserted an `\empty` to prevent brace removal. The `\expandafter` get rid of the `\empty`.

```
256 \xint_firstofone{\long\def\XINT_zapbsp_e #1 } \xint_bye
257   {\expandafter\XINT_zapbsp_f \expandafter{#1}}%
```

Let's not forget when we glue to reinsert the two intermediate space tokens.

```
258 \long\edef\XINT_zapbsp_f #1#2{#2\space\space #1}%
```

33.8 `\xintZapLastSpaces`

1.09f, written [2013/11/01].

```
259 \def\xintZapLastSpaces {\romannumeral0\xintzaplastspaces }%
```

Next macro is defined via an `\edef` for the space tokens.

```
260 \long\edef\xintzaplastspaces #1{\noexpand\XINT_zapesp_a {\space}\noexpand\empty
261   #1\space\space\noexpand\xint_bye \xint_relax}%
```

This creates a delimited macro with two space tokens:

```
262 \xint_firstofone {\long\def\XINT_zapesp_a #1#2 } %<- second space here
263   {\expandafter\XINT_zapesp_b\expandafter{#2}{#1}}%
```

The `\empty` from `\xintzaplastspaces` is to prevent brace removal in the `#2` above. The `\expandafter` chain removes it.

```
264 \long\def\XINT_zapesp_b #1#2#3\xint_relax
265   {\XINT_zapesp_end? #3\XINT_zapesp_e {#2#1}\empty #3\xint_relax }%
```

When we have reached the ending space tokens, `#3` is a bunch of spaces followed by `\xint_bye`. So the `#1` below will be `\xint_bye`. In all other cases `#1` can not be `\xint_bye` nor can it give birth to it via brace stripping.

```
266 \long\def\XINT_zapesp_end? #1{\xint_bye #1\XINT_zapesp_end }%
```

We are done. The `#1` here has accumulated all the previous material. It started with a space token which stops the `\romannumeral0`. The reason for the space is the recycling of this code in `\xintZapSpaces`.

```
267 \long\def\XINT_zapesp_end\XINT_zapesp_e #1#2\xint_relax {#1}%
```

We haven't yet reached the end, so we need to re-inject two space tokens after what we have gotten so far. Then we loop. We might wonder why in `\XINT_zapesp_b` we scooped everything up to the end, rather than trying to test if the next thing was a bunch of spaces followed by `\xint_bye\xint_relax`. But how can we expandably examine what comes next? if we pick up something as undelimited parameter token we risk brace removal and we will never know about it so we cannot reinsert correctly; the only way is to gather a delimited macro parameter and be sure some token will be

inside to forbid brace removal. I do not see (so far) any other way than scooping everything up to the end. Anyhow, 99% of the use cases will NOT have <sp><sp> inside!.

```
268 \long\edef\XINT_zapesp_e #1{\noexpand \XINT_zapesp_a {#1\space\space}}%
```

33.9 \xintZapSpaces

1.09f, written [2013/11/01].

```
269 \def\xintZapSpaces {\romannumeral0\xintzapspaces }%
```

We start like \xintZapStartSpaces.

```
270 \long\edef\xintzapspaces #1%
271 {\noexpand\XINT_zapsp_a \space #1\space\space\noexpand\xint_bye\xint_relax}%
```

Once the loop stripping the starting spaces is done, we plug into the \xintZapLastSpaces code via \XINT_zapesp_b. As our #1 will always have an initial space, this is why we arranged code of \xintZapLastSpaces to do the same.

```
272 \xint_firstofone {\long\def\XINT_zapsp_a #1 } %<- space token here
273 {%
274 \XINT_zapsp_again? #1\xint_bye\XINT_zapesp_b {#1}}%
275}%
276 \long\def\XINT_zapsp_again? #1{\xint_bye #1\XINT_zapsp_again}%
277 \long\edef\XINT_zapsp_again\XINT_zapesp_b #1#2{\noexpand\XINT_zapsp_a\space}%

```

33.10 \xintZapSpacesB

1.09f, written [2013/11/01].

```
278 \def\xintZapSpacesB {\romannumeral0\xintzapspacesb}%
279 \long\def\xintzapspacesb #1{\XINT_zapspb_one? #1\xint_relax\xint_relax
280 \xint_bye\xintzapspaces {#1}}%
281 \long\def\XINT_zapspb_one? #1#2%
282 {\xint_gob_til_xint_relax #1\XINT_zapspb_onlyspaces\xint_relax
283 \xint_gob_til_xint_relax #2\XINT_zapspb_bracedorone\xint_relax
284 \xint_bye {#1}}%
285 \def\XINT_zapspb_onlyspaces\xint_relax
286 \xint_gob_til_xint_relax\xint_relax\XINT_zapspb_bracedorone\xint_relax
287 \xint_bye #1\xint_bye\xintzapspaces #2{ }%
288 \long\def\XINT_zapspb_bracedorone\xint_relax
289 \xint_bye #1\xint_relax\xint_bye\xintzapspaces #2{ #1}%

```

33.11 \xintCSVtoList, \xintCSVtoListNonStripped

\xintCSVtoList transforms `a,b,...,z` into `{a}{b}...{z}`. The comma separated list may be a macro which is first expanded (protect the first item with a space if it is not to be expanded). First included in release 1.06. Here, use of \Z (and \R) perfectly safe.

[2013/11/02]: Starting with 1.09f, automatically filters items through \xintZapSpacesB to strip off all spaces around commas, and spaces at the start and end of the list. The original is kept as \xintCSVtoListNonStripped, and is faster. But ... it doesn't strip spaces.

```

290 \def\xintCSVtoList {\romannumeral0\xintcsvtolist }%
291 \long\def\xintcsvtolist #1{\expandafter\xintApply
292     \expandafter\xintzapspacesb
293     \expandafter{\romannumeral0\xintcsvtolistnonstripped{#1}}}%
294 \def\xintCSVtoListNoExpand {\romannumeral0\xintcsvtolistnoexpand }%
295 \long\def\xintcsvtolistnoexpand #1{\expandafter\xintApply
296     \expandafter\xintzapspacesb
297     \expandafter{\romannumeral0\xintcsvtolistnonstrippednoexpand{#1}}}%
298 \def\xintCSVtoListNonStripped {\romannumeral0\xintcsvtolistnonstripped }%
299 \def\xintCSVtoListNonStrippedNoExpand
300     {\romannumeral0\xintcsvtolistnonstrippednoexpand }%
301 \long\def\xintcsvtolistnonstripped #1%
302 {%
303     \expandafter\XINT_csvtol_loop_a\expandafter
304     {\expandafter}\romannumeral-'\0#1%
305     ,\xint_bye,\xint_bye,\xint_bye,\xint_bye
306     ,\xint_bye,\xint_bye,\xint_bye,\xint_bye,\Z
307 }%
308 \long\def\xintcsvtolistnonstrippednoexpand #1%
309 {%
310     \XINT_csvtol_loop_a
311     {#1,\xint_bye,\xint_bye,\xint_bye,\xint_bye
312     ,\xint_bye,\xint_bye,\xint_bye,\xint_bye,\Z
313 }%
314 \long\def\XINT_csvtol_loop_a #1#2,#3,#4,#5,#6,#7,#8,#9,%
315 {%
316     \xint_bye #9\XINT_csvtol_finish_a\xint_bye
317     \XINT_csvtol_loop_b {#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}}%
318 }%
319 \long\def\XINT_csvtol_loop_b #1#2{\XINT_csvtol_loop_a {#1#2}}%
320 \long\def\XINT_csvtol_finish_a\xint_bye\XINT_csvtol_loop_b #1#2#3\Z
321 {%
322     \XINT_csvtol_finish_b #3\R,\R,\R,\R,\R,\R,\R,\Z #2{#1}%
323 }%
324 \def\XINT_csvtol_finish_b #1,#2,#3,#4,#5,#6,#7,#8\Z
325 {%
326     \xint_gob_til_R
327     #1\XINT_csvtol_finish_c 8%

```

```

328      #2\XINT_csvtol_finish_c 7%
329      #3\XINT_csvtol_finish_c 6%
330      #4\XINT_csvtol_finish_c 5%
331      #5\XINT_csvtol_finish_c 4%
332      #6\XINT_csvtol_finish_c 3%
333      #7\XINT_csvtol_finish_c 2%
334      \R\XINT_csvtol_finish_c 1\Z
335 }%
336 \def\XINT_csvtol_finish_c #1#2\Z
337 {%
338   \csname XINT_csvtol_finish_d\romannumeral #1\endcsname
339 }%
340 \long\def\XINT_csvtol_finish_dviii #1#2#3#4#5#6#7#8#9{ #9}%
341 \long\def\XINT_csvtol_finish_dvii #1#2#3#4#5#6#7#8#9{ #9{#1}}%
342 \long\def\XINT_csvtol_finish_dvi #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}}%
343 \long\def\XINT_csvtol_finish_dv #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}}%
344 \long\def\XINT_csvtol_finish_div #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}}%
345 \long\def\XINT_csvtol_finish_diii #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}{#5}}%
346 \long\def\XINT_csvtol_finish_dii #1#2#3#4#5#6#7#8#9%
347      { #9{#1}{#2}{#3}{#4}{#5}{#6}}%
348 \long\def\XINT_csvtol_finish_di #1#2#3#4#5#6#7#8#9%
349      { #9{#1}{#2}{#3}{#4}{#5}{#6}{#7}}%

```

33.12 \xintListWithSep

\xintListWithSep {\sep}{a}{b}...{z} returns a \sep b \sep ... \sep z
Included in release 1.04. The 'sep' can be \par's: the macro xintlistwithsep
etc... are all declared long. 'sep' does not have to be a single token. It is not
expanded. The list may be a macro and it is expanded. 1.06 modifies the 'feature'
of returning sep if the list is empty: the output is now empty in that case. (sep
was not used for a one element list, but strangely it was for a zero-element list).

Use of \Z as delimiter was objectively an error, which I fix here in 1.09e, now
the code uses \xint_bye.

```

350 \def\xintListWithSep {\romannumeral0\xintlistwithsep}%
351 \def\xintListWithSepNoExpand {\romannumeral0\xintlistwithsepnoexpand}%
352 \long\def\xintlistwithsep #1#2%
353   {\expandafter\XINT_lws\expandafter {\romannumeral-0#2}{#1}}%
354 \long\def\XINT_lws #1#2{\XINT_lws_start {#2}#1\xint_bye}%
355 \long\def\xintlistwithsepnoexpand #1#2{\XINT_lws_start {#1}#2\xint_bye}%
356 \long\def\XINT_lws_start #1#2%
357 {%
358   \xint_bye #2\XINT_lws_dont\xint_bye
359   \XINT_lws_loop_a {#2}{#1}%
360 }%
361 \long\def\XINT_lws_dont\xint_bye\XINT_lws_loop_a #1#2{ }%
362 \long\def\XINT_lws_loop_a #1#2#3%
363 {%
364   \xint_bye #3\XINT_lws_end\xint_bye

```

```

365 \XINT_lws_loop_b {#1}{#2#3}{#2}%
366 }%
367 \long\def\XINT_lws_loop_b #1#2{\XINT_lws_loop_a {#1#2}}%
368 \long\def\XINT_lws_end\xint_bye\XINT_lws_loop_b #1#2#3{ #1}%

```

33.13 \xintNthElt

\xintNthElt {i}{{a}{b}...{z}} (or ‘tokens’ abcd...z) returns the *i* th element (one pair of braces removed). The list is first expanded. First included in release 1.06. With 1.06a, a value of *i* = 0 (or negative) makes the macro return the length. This is different from \xintLen which is for numbers (checks sign) and different from \xintLength which does not first expand its argument. With 1.09b, only *i*=0 gives the length, negative values return the *i* th element from the end. 1.09c has some slightly less quick initial preparation (if #2 is very long, not good to have it twice), I wanted to respect the noexpand directive in all cases, and the alternative would be to define more macros.

At some point I turned the \W’s into \xint_relax’s but forgot to modify accordingly \XINT_nthelt_finish. So in case the index is larger than the number of items the macro returned was an \xint_relax token rather than nothing. Fixed in 1.09e. I also take the opportunity of this fix to replace uses of \Z by \xint_bye. (and as a result I must do the change also in \XINT_length_loop and related macros).

```

369 \def\xintNthElt          {\romannumeral0\xintnthelt }%
370 \def\xintNthEltNoExpand {\romannumeral0\xintntheltnoexpand }%
371 \def\xintnthelt #1%
372 {%
373   \expandafter\XINT_nthelt_a\expandafter {\the\numexpr #1}%
374 }%
375 \def\xintntheltnoexpand #1%
376 {%
377   \expandafter\XINT_ntheltnoexpand_a\expandafter {\the\numexpr #1}%
378 }%
379 \long\def\XINT_nthelt_a #1#2%
380 {%
381   \ifnum #1<0
382     \xint_afterfi{\expandafter\XINT_nthelt_c\expandafter
383                  {\romannumeral0\xintrevwithbraces {#2}}{-#1}}%
384   \else
385     \xint_afterfi{\expandafter\XINT_nthelt_c\expandafter
386                  {\romannumeral-‘0#2}{#1}}%
387   \fi
388 }%
389 \long\def\XINT_ntheltnoexpand_a #1#2%
390 {%
391   \ifnum #1<0
392     \xint_afterfi{\expandafter\XINT_nthelt_c\expandafter
393                  {\romannumeral0\xintrevwithbracesnoexpand {#2}}{-#1}}%
394   \else
395     \xint_afterfi{\expandafter\XINT_nthelt_c\expandafter

```



```

396             {#2}{#1}}%
397     \fi
398 }%
399 \long\def\XINT_nthelt_c #1#2%
400 {%
401     \ifnum #2>\xint_c_
402         \expandafter\XINT_nthelt_loop_a
403     \else
404         \expandafter\XINT_length_loop
405     \fi {#2}{#1}\xint_relax\xint_relax\xint_relax\xint_relax
406         \xint_relax\xint_relax\xint_relax\xint_relax\xint_bye
407 }%
408 \def\XINT_nthelt_loop_a #1%
409 {%
410     \ifnum #1>\xint_c_viii
411         \expandafter\XINT_nthelt_loop_b
412     \else
413         \expandafter\XINT_nthelt_getit
414     \fi
415     {#1}%
416 }%
417 \long\def\XINT_nthelt_loop_b #1#2#3#4#5#6#7#8#9%
418 {%
419     \xint_gob_til_xint_relax #9\XINT_nthelt_silentend\xint_relax
420     \expandafter\XINT_nthelt_loop_a\expandafter{\the\numexpr #1-8}%
421 }%
422 \def\XINT_nthelt_silentend #1\xint_bye { }%
423 \def\XINT_nthelt_getit #1%
424 {%
425     \expandafter\expandafter\expandafter\XINT_nthelt_finish
426     \csname xint_gobble_\romannumeral\numexpr#1-1\endcsname
427 }%
428 \long\def\XINT_nthelt_finish #1#2\xint_bye
429     {\xint_gob_til_xint_relax #1\expandafter\space
430         \xint_gobble_iii\xint_relax\space #1}%

```

33.14 \xintApply

`\xintApply {\macro}{a}{b}...{z}` returns `{\macro{a}}...{\macro{b}}` where each instance of `\macro` is ff-expanded. The list is first expanded and may thus be a macro. Introduced with release 1.04.

Modified in 1.09e to not use `\Z` but rather `\xint_bye`.

```

431 \def\xintApply          {\romannumeral0\xintapply }%
432 \def\xintApplyNoExpand {\romannumeral0\xintapplynoexpand }%
433 \long\def\xintapply #1#2%
434 {%
435     \expandafter\XINT_apply\expandafter {\romannumeral-‘0#2}%
436     {#1}%

```

```

437 }%
438 \long\def\XINT_apply #1#2{\XINT_apply_loop_a {}{#2}#1\xint_bye }%
439 \long\def\xintapplynoexpand #1#2{\XINT_apply_loop_a {}{#1}#2\xint_bye }%
440 \long\def\XINT_apply_loop_a #1#2#3%
441 {%
442   \xint_bye #3\XINT_apply_end\xint_bye
443   \expandafter
444   \XINT_apply_loop_b
445   \expandafter {\romannumeral-‘0#2{#3}}{#1}{#2}%
446 }%
447 \long\def\XINT_apply_loop_b #1#2{\XINT_apply_loop_a {#2}#1}%
448 \long\def\XINT_apply_end\xint_bye\expandafter\XINT_apply_loop_b
449   \expandafter #1#2#3{ #2}%

```

33.15 \xintApplyUnbraced

\xintApplyUnbraced {\macro}{a}{b}...{z} returns \macro{a}...\macro{z} where each instance of \macro is expanded using \romannumeral-‘0. The second argument may be a macro as it is first expanded itself (fully). No braces are added: this allows for example a non-expandable \def in \macro, without having to do \gdef. The list is first expanded. Introduced with release 1.06b. Define \macro to start with a space if it is not expandable or its execution should be delayed only when all of \macro{a}...\macro{z} is ready.

Modified in 1.09e to use \xint_bye rather than \Z.

```

450 \def\xintApplyUnbraced {\romannumeral0\xintapplyunbraced }%
451 \def\xintApplyUnbracedNoExpand {\romannumeral0\xintapplyunbracednoexpand }%
452 \long\def\xintapplyunbraced #1#2%
453 {%
454   \expandafter\XINT_applyunbr\expandafter {\romannumeral-‘0#2}%
455   {#1}%
456 }%
457 \long\def\XINT_applyunbr #1#2{\XINT_applyunbr_loop_a {}{#2}#1\xint_bye }%
458 \long\def\xintapplyunbracednoexpand #1#2%
459   {\XINT_applyunbr_loop_a {}{#1}#2\xint_bye }%
460 \long\def\XINT_applyunbr_loop_a #1#2#3%
461 {%
462   \xint_bye #3\XINT_applyunbr_end\xint_bye
463   \expandafter\XINT_applyunbr_loop_b
464   \expandafter {\romannumeral-‘0#2{#3}}{#1}{#2}%
465 }%
466 \long\def\XINT_applyunbr_loop_b #1#2{\XINT_applyunbr_loop_a {#2}#1}%
467 \long\def\XINT_applyunbr_end\xint_bye\expandafter\XINT_applyunbr_loop_b
468   \expandafter #1#2#3{ #2}%

```

33.16 \xintSeq

1.09c. Without the optional argument puts stress on the input stack, should not be used to generated thousands of terms then. Here also, let's use \xint_bye rather

33 Package *xinttools* implementation

than \Z as delimiter (1.09e; necessary change as #1 is used prior to being expanded, thus \Z might very well arise here as a macro).

```

469 \def\xintSeq {\romannumeral0\xintseq }%
470 \def\xintseq #1{\XINT_seq_chkopt #1\xint_bye }%
471 \def\XINT_seq_chkopt #1%
472 {%
473   \ifx [#1\expandafter\XINT_seq_opt
474     \else\expandafter\XINT_seq_noopt
475   \fi #1%
476 }%
477 \def\XINT_seq_noopt #1\xint_bye #2%
478 {%
479   \expandafter\XINT_seq\expandafter
480     {\the\numexpr#1\expandafter}\expandafter{\the\numexpr #2}%
481 }%
482 \def\XINT_seq #1#2%
483 {%
484   \ifcase\ifnum #1=#2 0\else\ifnum #2>#1 1\else -1\fi\fi\space
485     \expandafter\xint_firstoftwo_andstop
486   \or
487     \expandafter\XINT_seq_p
488   \else
489     \expandafter\XINT_seq_n
490   \fi
491   {#2}{#1}%
492 }%
493 \def\XINT_seq_p #1#2%
494 {%
495   \ifnum #1>#2
496     \xint_afterfi{\expandafter\XINT_seq_p}%
497   \else
498     \expandafter\XINT_seq_e
499   \fi
500   \expandafter{\the\numexpr #1-1}{#2}{#1}%
501 }%
502 \def\XINT_seq_n #1#2%
503 {%
504   \ifnum #1<#2
505     \xint_afterfi{\expandafter\XINT_seq_n}%
506   \else
507     \expandafter\XINT_seq_e
508   \fi
509   \expandafter{\the\numexpr #1+1}{#2}{#1}%
510 }%
511 \def\XINT_seq_e #1#2#3{ }%
512 \def\XINT_seq_opt [\xint_bye #1]#2#3%
513 {%
514   \expandafter\XINT_seqo\expandafter

```

33 Package *xinttools* implementation

```

515     {\the\numexpr #2\expandafter}\expandafter
516     {\the\numexpr #3\expandafter}\expandafter
517     {\the\numexpr #1}%
518 }%
519 \def\XINT_seqo #1#2%
520 {%
521     \ifcase\ifnum #1=#2 0\else\ifnum #2>#1 1\else -1\fi\fi\space
522         \expandafter\XINT_seqo_a
523     \or
524         \expandafter\XINT_seqo_pa
525     \else
526         \expandafter\XINT_seqo_na
527     \fi
528     {#1}{#2}%
529 }%
530 \def\XINT_seqo_a #1#2#3{ {#1}}%
531 \def\XINT_seqo_o #1#2#3#4{ #4}%
532 \def\XINT_seqo_pa #1#2#3%
533 {%
534     \ifcase\ifnum #3=\xint_c_ 0\else\ifnum #3>\xint_c_ 1\else -1\fi\fi\space
535         \expandafter\XINT_seqo_o
536     \or
537         \expandafter\XINT_seqo_pb
538     \else
539         \xint_afterfi{\expandafter\space\xint_gobble_iv}%
540     \fi
541     {#1}{#2}{#3}{#1}%
542 }%
543 \def\XINT_seqo_pb #1#2#3%
544 {%
545     \expandafter\XINT_seqo_pc\expandafter{\the\numexpr #1+#3}{#2}{#3}%
546 }%
547 \def\XINT_seqo_pc #1#2%
548 {%
549     \ifnum #1>#2
550         \expandafter\XINT_seqo_o
551     \else
552         \expandafter\XINT_seqo_pd
553     \fi
554     {#1}{#2}%
555 }%
556 \def\XINT_seqo_pd #1#2#3#4{\XINT_seqo_pb {#1}{#2}{#3}{#4{#1}}}%
557 \def\XINT_seqo_na #1#2#3%
558 {%
559     \ifcase\ifnum #3=\xint_c_ 0\else\ifnum #3>\xint_c_ 1\else -1\fi\fi\space
560         \expandafter\XINT_seqo_o
561     \or
562         \xint_afterfi{\expandafter\space\xint_gobble_iv}%
563     \else

```

```

564      \expandafter\XINT_sequo_nb
565      \fi
566      {#1}{#2}{#3}{#1}}%
567 }%
568 \def\XINT_sequo_nb #1#2#3%
569 {%
570      \expandafter\XINT_sequo_nc\expandafter{\the\numexpr #1+#3}{#2}{#3}%
571 }%
572 \def\XINT_sequo_nc #1#2%
573 {%
574      \ifnum #1<#2
575          \expandafter\XINT_sequo_o
576      \else
577          \expandafter\XINT_sequo_nd
578      \fi
579      {#1}{#2}%
580 }%
581 \def\XINT_sequo_nd #1#2#3#4{\XINT_sequo_nb {#1}{#2}{#3}{#4{#1}}}%

```

33.17 \xintloop, \xintbreakloop, \xintbreakloopanddo, \xintloopskiptonext

1.09g [2013/11/22]. Made long with 1.09h.

```

582 \long\def\xintloop #1#2\repeat {#1#2\xintloop_again\fi\xint_gobble_i {#1#2}}%
583 \long\def\xintloop_again\fi\xint_gobble_i #1{\fi
584      #1\xintloop_again\fi\xint_gobble_i {#1}}%
585 \long\def\xintbreakloop #1\xintloop_again\fi\xint_gobble_i #2{%
586 \long\def\xintbreakloopanddo #1#2\xintloop_again\fi\xint_gobble_i #3{#1}%
587 \long\def\xintloopskiptonext #1\xintloop_again\fi\xint_gobble_i #2{%
588      #2\xintloop_again\fi\xint_gobble_i {#2}}%

```

33.18 \xintilloop, \xintilloopindex, \xintouterilloopindex, \xintbreakilloop, \xintbreakilloopanddo, \xintilloopskiptonext, \xintilloopskipandredo

1.09g [2013/11/22]. Made long with 1.09h.

```

589 \def\xintilloop [#1+#2]{%
590      \expandafter\xintilloop_a\the\numexpr #1\expandafter.\the\numexpr #2.%
591 \long\def\xintilloop_a #1.#2.#3#4\repeat{%
592      #3#4\xintilloop_again\fi\xint_gobble_iii {#1}{#2}{#3#4}}%
593 \def\xintilloop_again\fi\xint_gobble_iii #1#2{%
594      \fi\expandafter\xintilloop_again_b\the\numexpr#1+#2.#2.%
595 \long\def\xintilloop_again_b #1.#2.#3{%
596      #3\xintilloop_again\fi\xint_gobble_iii {#1}{#2}{#3}}%
597 \long\def\xintbreakilloop #1\xintilloop_again\fi\xint_gobble_iii #2#3#4{%
598 \long\def\xintbreakilloopanddo
599      #1.#2\xintilloop_again\fi\xint_gobble_iii #3#4#5{#1}%
600 \long\def\xintilloopindex #1\xintilloop_again\fi\xint_gobble_iii #2%

```

```

601      {#2#1\xintilooop_again\fi\xint_gobble_iii {#2}}%
602 \long\def\xintouterilooopindex #1\xintilooop_again
603      #2\xintilooop_again\fi\xint_gobble_iii #3%
604      {#3#1\xintilooop_again #2\xintilooop_again\fi\xint_gobble_iii {#3}}%
605 \long\def\xintilooopskiptonext #1\xintilooop_again\fi\xint_gobble_iii #2#3{%
606      \expandafter\xintilooop_again_b \the\numexpr#2+#3.#3.}%
607 \long\def\xintilooopskipandredo #1\xintilooop_again\fi\xint_gobble_iii #2#3#4{%
608      #4\xintilooop_again\fi\xint_gobble_iii {#2}{#3}{#4}}%

```

33.19 \XINT_xflet

1.09e [2013/10/29]: we expand fully unbraced tokens and swallow arising space tokens until the dust settles. For treating cases {<blank>\x<blank>y...}, with guaranteed expansion of the \x (which may itself give space tokens), a simpler approach is possible with doubled \romannumeral-‘0, this is what I first did, but it had the feature that <sptoken><sptoken>\x would not expand the \x. At any rate, <sptoken>’s before the list terminator z were all correctly moved out of the way, hence the stuff was robust for use in (the then current versions of) \xintApplyInline and \xintFor. Although *two* space tokens would need devilishly prepared input, nevertheless I decided to also survive that, so here the method is a bit more complicated. But it simplifies things on the caller side.

```

609 \def\XINT_xflet #1%
610 {%
611     \def\XINT_xflet_macro {#1}\XINT_xflet_zapsp
612 }%
613 \def\XINT_xflet_zapsp
614 {%
615     \expandafter\futurelet\expandafter\XINT_token
616     \expandafter\XINT_xflet_sp?\romannumeral-‘0%
617 }%
618 \def\XINT_xflet_sp?
619 {%
620     \ifx\XINT_token\XINT_sptoken
621         \expandafter\XINT_xflet_zapsp
622     \else\expandafter\XINT_xflet_zapspB
623     \fi
624 }%
625 \def\XINT_xflet_zapspB
626 {%
627     \expandafter\futurelet\expandafter\XINT_tokenB
628     \expandafter\XINT_xflet_spB?\romannumeral-‘0%
629 }%
630 \def\XINT_xflet_spB?
631 {%
632     \ifx\XINT_tokenB\XINT_sptoken
633         \expandafter\XINT_xflet_zapspB
634     \else\expandafter\XINT_xflet_eq?
635     \fi

```

```

636 }%
637 \def\XINT_xflet_eq?
638 {%
639   \ifx\XINT_token\XINT_tokenB
640     \expandafter\XINT_xflet_macro
641   \else\expandafter\XINT_xflet_zapsp
642   \fi
643 }%

```

33.20 \xintApplyInline

1.09a: \xintApplyInline\macro{{a}{b}...{z}} has the same effect as executing \macro{a} and then applying again \xintApplyInline to the shortened list {{b}...{z}} until nothing is left. This is a non-expandable command which will result in quicker code than using \xintApplyUnbraced. It expands (fully) its second (list) argument first, which may thus be encapsulated in a macro.

Release 1.09c has a new \xintApplyInline: the new version, while not expandable, is designed to survive when the applied macro closes a group, as is the case in alignmentts when it contains a & or \\. It uses catcode 3 Z as list terminator. Don't use it among the list items.

1.09d: the bug which was discovered in \xintFor* regarding space tokens at the very end of the item list also was in \xintApplyInline. The new version will expand unbraced item elements and this is in fact convenient to simulate insertion of lists in others.

1.09e: the applied macro is allowed to be long, with items (or the first fixed arguments of the macro, passed together with it as #1 to \xintApplyInline) containing explicit \par's. (1.09g: some missing \long's added)

1.09f: terminator used to be z, now Z (still catcode 3).

```

644 \catcode'Z 3
645 \long\def\xintApplyInline #1#2%
646 {%
647   \long\expandafter\def\expandafter\XINT_inline_macro
648   \expandafter ##\expandafter 1\expandafter {#1{##1}}%
649   \XINT_xflet\XINT_inline_b #2Z% this Z has catcode 3
650 }%
651 \def\XINT_inline_b
652 {%
653   \ifx\XINT_token Z\expandafter\xint_gobble_i
654   \else\expandafter\XINT_inline_d\fi
655 }%
656 \long\def\XINT_inline_d #1%
657 {%
658   \long\def\XINT_item{{#1}}\XINT_xflet\XINT_inline_e
659 }%
660 \def\XINT_inline_e
661 {%
662   \ifx\XINT_token Z\expandafter\XINT_inline_w
663   \else\expandafter\XINT_inline_f\fi

```



```

664 }%
665 \def\XINT_inline_f
666 {%
667   \expandafter\XINT_inline_g\expandafter{\XINT_inline_macro {##1}}%
668 }%
669 \long\def\XINT_inline_g #1%
670 {%
671   \expandafter\XINT_inline_macro\XINT_item
672   \long\def\XINT_inline_macro ##1{#1}\XINT_inline_d
673 }%
674 \def\XINT_inline_w #1%
675 {%
676   \expandafter\XINT_inline_macro\XINT_item
677 }%

```

33.21 \xintFor, \xintFor*, \xintBreakFor, \xintBreakForAndDo

1.09c [2013/10/09]: a new kind of loop which uses macro parameters #1, #2, #3, #4 rather than macros; while not expandable it survives executing code closing groups, like what happens in an alignment with the & character. When inserted in a macro for later use, the # character must be doubled.

The non-star variant works on a csv list, which it expands once, the star variant works on a token list, expanded fully.

1.09d: [2013/10/22] \xintFor* crashed when a space token was at the very end of the list. It is crucial in this code to not let the ending Z be picked up as a macro parameter without knowing in advance that it is its turn. So, we conscientiously clean out of the way space tokens, but also we ff-expand with \romannumeral-‘0 (unbraced) items, a process which may create new space tokens, so it is iterated. As unbraced items are expanded, it is easy to simulate insertion of a list in another. Unbraced items consecutive to an even (non-zero) number of space tokens will not get expanded.

1.09e: [2013/10/29] does this better, no difference between an even or odd number of explicit consecutive space tokens. Normal situations anyhow only create at most one space token, but well. There was a feature in \xintFor (not \xintFor*) from 1.09c that it treated an empty list as a list with one, empty, item. This feature is kept in 1.09e, knowingly... Also, macros are made long, hence the iterated text may contain \par and also the looped over items. I thought about providing some macro expanding to the loop count, but as the \xintFor is not expandable anyhow, there is no loss of generality if the iterated commands do themselves the bookkeeping using a count or a LaTeX counter, and deal with nesting or other problems. I can't do *everything*!

1.09e adds \XINT_forever with \xintintegers, \xintdimensions, \xintrationals and \xintBreakFor, \xintBreakForAndDo, \xintifForFirst, \xintifForLast. On this occasion \xint_firstoftwo and \xint_secondoftwo are made long.

1.09f: rewrites large parts of \xintFor code in order to filter the comma separated list via \xintCSVtoList which gets rid of spaces. Compatibility with \XINT_forever, the necessity to prevent unwanted brace stripping, and shared code with \xintFor*, make this all a delicate balancing act. The #1 in

\XINT_for_forever? has an initial space token which serves two purposes: preventing brace stripping, and stopping the expansion made by \xintcsvtolist. If the \XINT_forever branch is taken, the added space will not be a problem there.

1.09f rewrites (2013/11/03) the code which now allows all macro parameters from #1 to #9 in \xintFor, \xintFor*, and \XINT_forever.

The 1.09f \xintFor and \xintFor* modified the value of \count 255 which was silly, 1.09g used \XINT_count, but requiring a \count only for that was also silly, 1.09h just uses \numexpr (all of that was only to get rid simply of a possibly space in #2...).

```

678 \def\XINT_tmpa #1#2{\ifnum #2<#1 \xint_afterfi {{#####2}}\fi}%
679 \def\XINT_tmpb #1#2{\ifnum #1<#2 \xint_afterfi {{#####2}}\fi}%
680 \def\XINT_tmpc #1%
681 {%
682   \expandafter\edef \csname XINT_for_left#1\endcsname
683     {\xintApplyUnbraced {\XINT_tmpa #1}{123456789}}%
684   \expandafter\edef \csname XINT_for_right#1\endcsname
685     {\xintApplyUnbraced {\XINT_tmpb #1}{123456789}}%
686 }%
687 \xintApplyInline \XINT_tmpc {123456789}%
688 \long\def\xintBreakFor #1Z{%
689 \long\def\xintBreakForAndDo #1#2Z{#1}%
690 \def\xintFor {\let\xintifForFirst\xint_firstoftwo
691   \futurelet\XINT_token\XINT_for_ifstar }%
692 \def\XINT_for_ifstar {\ifx\XINT_token*\expandafter\XINT_forx
693   \else\expandafter\XINT_for \fi }%
694 \catcode'U 3 % with numexpr
695 \catcode'V 3 % with xintfrac.sty (xint.sty not enough)
696 \catcode'D 3 % with dimexpr
697 % \def\XINT_flet #1%
698 % {%
699 %   \def\XINT_flet_macro {#1}\XINT_flet_zapsp
700 % }%
701 \def\XINT_flet_zapsp
702 {%
703   \futurelet\XINT_token\XINT_flet_sp?
704 }%
705 \def\XINT_flet_sp?
706 {%
707   \ifx\XINT_token\XINT_sptoken
708     \xint_afterfi{\expandafter\XINT_flet_zapsp\romannumeral0}%
709   \else\expandafter\XINT_flet_macro
710   \fi
711 }%
712 \long\def\XINT_for #1#2in#3#4#5%
713 {%
714   \expandafter\XINT_toks\expandafter
715     {\expandafter\XINT_for_d\the\numexpr #2\relax {#5}}%
716   \def\XINT_flet_macro {\expandafter\XINT_for_forever?\space}%

```

33 Package *xinttools* implementation

```

717 \expandafter\XINT_flet_zapsp #3Z%
718 }%
719 \def\XINT_for_forever? #1Z%
720 {%
721 \ifx\XINT_token U\XINT_to_forever\fi
722 \ifx\XINT_token V\XINT_to_forever\fi
723 \ifx\XINT_token D\XINT_to_forever\fi
724 \expandafter\the\expandafter\XINT_toks\romannumeral0\xintcsvtolist {#1}Z%
725 }%
726 \def\XINT_to_forever\fi #1\xintcsvtolist #2{\fi \XINT_forever #2}%
727 \long\def\XINT_forx *#1#2in#3#4#5%
728 {%
729 \expandafter\XINT_toks\expandafter
730 {\expandafter\XINT_forx_d\the\numexpr #2\relax {#5}}%
731 \XINT_xflet\XINT_forx_forever? #3Z%
732 }%
733 \def\XINT_forx_forever?
734 {%
735 \ifx\XINT_token U\XINT_to_forxever\fi
736 \ifx\XINT_token V\XINT_to_forxever\fi
737 \ifx\XINT_token D\XINT_to_forxever\fi
738 \XINT_forx_empty?
739 }%
740 \def\XINT_to_forxever\fi #1\XINT_forx_empty? {\fi \XINT_forever }%
741 \catcode'U 11
742 \catcode'D 11
743 \catcode'V 11
744 \def\XINT_forx_empty?
745 {%
746 \ifx\XINT_token Z\expandafter\xintBreakFor\fi
747 \the\XINT_toks
748 }%
749 \long\def\XINT_for_d #1#2#3%
750 {%
751 \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
752 \XINT_toks {{#3}}%
753 \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
754 \the\XINT_toks \csname XINT_for_right#1\endcsname }%
755 \XINT_toks {\XINT_x\let\xintifForFirst\xint_secondoftwo\XINT_for_d #1{#2}}%
756 \futurelet\XINT_token\XINT_for_last?
757 }%
758 \long\def\XINT_forx_d #1#2#3%
759 {%
760 \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
761 \XINT_toks {{#3}}%
762 \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
763 \the\XINT_toks \csname XINT_for_right#1\endcsname }%
764 \XINT_toks {\XINT_x\let\xintifForFirst\xint_secondoftwo\XINT_forx_d #1{#2}}%
765 \XINT_xflet\XINT_for_last?

```

```

766 }%
767 \def\XINT_for_last?
768 {%
769   \let\xintifForLast\xint_secondoftwo
770   \ifx\XINT_token Z\let\xintifForLast\xint_firstoftwo
771               \xint_afterfi{\xintBreakForAndDo\XINT_x}\fi
772   \the\XINT_toks
773 }%

```

33.22 \XINT_forever, \xintintegers, \xintdimensions, \xintrationals

New with 1.09e. But this used inadvertently \xintiadd/\xintimul which have the unnecessary \xintnum overhead. Changed in 1.09f to use \xintiiadd/\xintiimul which do not have this overhead. Also 1.09f has \xintZapSpacesB which helps getting rid of spaces for the \xintrationals case (the other cases end up inside a \numexpr, or \dimexpr, so not necessary).

```

774 \catcode'U 3
775 \catcode'D 3
776 \catcode'V 3
777 \let\xintegers      U%
778 \let\xintintegers  U%
779 \let\xintdimensions D%
780 \let\xintrationals  V%
781 \def\XINT_forever #1%
782 {%
783   \expandafter\XINT_forever_a
784   \csname XINT_?expr_\ifx#1UU\else\ifx#1DD\else V\fi\fi a\expandafter\endcsname
785   \csname XINT_?expr_\ifx#1UU\else\ifx#1DD\else V\fi\fi i\expandafter\endcsname
786   \csname XINT_?expr_\ifx#1UU\else\ifx#1DD\else V\fi\fi \endcsname
787 }%
788 \catcode'U 11
789 \catcode'D 11
790 \catcode'V 11
791 \def\XINT_?expr_Ua #1#2%
792   {\expandafter{\expandafter\numexpr\the\numexpr #1\expandafter\relax
793               \expandafter\relax\expandafter}%
794   \expandafter{\the\numexpr #2}}%
795 \def\XINT_?expr_Da #1#2%
796   {\expandafter{\expandafter\dimexpr\number\dimexpr #1\expandafter\relax
797               \expandafter s\expandafter p\expandafter\relax\expandafter}%
798   \expandafter{\number\dimexpr #2}}%
799 \catcode'Z 11
800 \def\XINT_?expr_Va #1#2%
801 {%
802   \expandafter\XINT_?expr_Vb\expandafter
803   {\romannumeral-'0\xintrawwithzeros{\xintZapSpacesB{#2}}}%
804   {\romannumeral-'0\xintrawwithzeros{\xintZapSpacesB{#1}}}%
805 }%

```

33 Package *xinttools* implementation

```

806 \catcode'Z 3
807 \def\XINT_?expr_Vb #1#2{\expandafter\XINT_?expr_Vc #2.#1.}%
808 \def\XINT_?expr_Vc #1/#2.#3/#4.%
809 {%
810     \xintifEq {#2}{#4}%
811     {\XINT_?expr_Vf {#3}{#1}{#2}}%
812     {\expandafter\XINT_?expr_Vd\expandafter
813     {\romannumeral0\xintiimul {#2}{#4}}%
814     {\romannumeral0\xintiimul {#1}{#4}}%
815     {\romannumeral0\xintiimul {#2}{#3}}%
816     }%
817 }%
818 \def\XINT_?expr_Vd #1#2#3{\expandafter\XINT_?expr_Ve\expandafter {#2}{#3}{#1}}%
819 \def\XINT_?expr_Ve #1#2{\expandafter\XINT_?expr_Vf\expandafter {#2}{#1}}%
820 \def\XINT_?expr_Vf #1#2#3{{#2/#3}{0}{#1}{#2}{#3}}%
821 \def\XINT_?expr_Ui {{\numexpr 1\relax}{1}}%
822 \def\XINT_?expr_Di {{\dimexpr 0pt\relax}{65536}}%
823 \def\XINT_?expr_Vi {{1/1}{0111}}%
824 \def\XINT_?expr_U #1#2%
825     {\expandafter{\expandafter\numexpr\the\numexpr #1+#2\relax\relax}{#2}}%
826 \def\XINT_?expr_D #1#2%
827     {\expandafter{\expandafter\dimexpr\the\numexpr #1+#2\relax sp\relax}{#2}}%
828 \def\XINT_?expr_V #1#2{\XINT_?expr_Vx #2}%
829 \def\XINT_?expr_Vx #1#2%
830 {%
831     \expandafter\XINT_?expr_Vy\expandafter
832     {\romannumeral0\xintiiaadd {#1}{#2}}{#2}%
833 }%
834 \def\XINT_?expr_Vy #1#2#3#4%
835 {%
836     \expandafter{\romannumeral0\xintiiaadd {#3}{#1}/#4}{#1}{#2}{#3}{#4}}%
837 }%
838 \def\XINT_forever_a #1#2#3#4%
839 {%
840     \ifx #4[\expandafter\XINT_forever_opt_a
841     \else\expandafter\XINT_forever_b
842     \fi #1#2#3#4%
843 }%
844 \def\XINT_forever_b #1#2#3Z{\expandafter\XINT_forever_c\the\XINT_toks #2#3}%
845 \long\def\XINT_forever_c #1#2#3#4#5%
846     {\expandafter\XINT_forever_d\expandafter #2#4#5{#3}Z}%
847 \def\XINT_forever_opt_a #1#2#3[#4+#5]#6Z%
848 {%
849     \expandafter\expandafter\expandafter
850     \XINT_forever_opt_c\expandafter\the\expandafter\XINT_toks
851     \romannumeral-'0#1{#4}{#5}#3%
852 }%
853 \long\def\XINT_forever_opt_c #1#2#3#4#5#6{\XINT_forever_d #2{#4}{#5}#6{#3}Z}%
854 \long\def\XINT_forever_d #1#2#3#4#5%

```

```

855 {%
856 \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#5}%
857 \XINT_toks {{#2}}%
858 \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
859 \the\XINT_toks \csname XINT_for_right#1\endcsname }%
860 \XINT_x
861 \let\xintifForFirst\xint_secondoftwo
862 \expandafter\XINT_forever_d\expandafter #1\romannumeral-‘0#4{#2}{#3}#4{#5}%
863 }%

```

33.23 \xintForpair, \xintForthree, \xintForfour

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

[2013/11/02] 1.09f: I may not have been very motivated in 1.09c, but since then I developed the \xintZapSpaces/\xintZapSpacesB tools (much to my satisfaction). Based on this, and better parameter texts, \xintForpair and its cousins now handle spaces very satisfactorily (this relies partly on the new \xintCSVtoList which makes use of \xintZapSpacesB). Does not share code with \xintFor anymore.

[2013/11/03] 1.09f: \xintForpair extended to accept #1#2, #2#3 etc... up to #8#9, \xintForthree, #1#2#3 up to #7#8#9, \xintForfour id.

```

864 \catcode‘j 3
865 \long\def\xintForpair #1#2#3in#4#5#6%
866 {%
867 \let\xintifForFirst\xint_firstoftwo
868 \XINT_toks {\XINT_forpair_d #2{#6}}%
869 \expandafter\the\expandafter\XINT_toks #4jZ%
870 }%
871 \long\def\XINT_forpair_d #1#2#3(#4)#5%
872 {%
873 \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
874 \XINT_toks \expandafter{\romannumeral0\xintcsvtolist{ #4}}%
875 \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
876 \the\XINT_toks \csname XINT_for_right\the\numexpr#1+1\endcsname}%
877 \let\xintifForLast\xint_secondoftwo
878 \ifx #5j\let\xintifForLast\xint_firstoftwo\expandafter\xintBreakForAndDo\fi
879 \XINT_x\let\xintifForFirst\xint_secondoftwo\XINT_forpair_d #1{#2}%
880 }%
881 \long\def\xintForthree #1#2#3in#4#5#6%
882 {%
883 \let\xintifForFirst\xint_firstoftwo
884 \XINT_toks {\XINT_forthree_d #2{#6}}%
885 \expandafter\the\expandafter\XINT_toks #4jZ%
886 }%
887 \long\def\XINT_forthree_d #1#2#3(#4)#5%
888 {%
889 \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%

```

```

890 \XINT_toks \expandafter{\romannumeral0\xintcsvtolist{ #4}}%
891 \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
892 \the\XINT_toks \csname XINT_for_right\the\numexpr#1+2\endcsname}%
893 \let\xintifForLast\xint_secondoftwo
894 \ifx #5j\let\xintifForLast\xint_firstoftwo\expandafter\xintBreakForAndDo\fi
895 \XINT_x\let\xintifForFirst\xint_secondoftwo\XINT_forthree_d #1{#2}%
896 }%
897 \long\def\xintForfour #1#2#3in#4#5#6%
898 {%
899 \let\xintifForFirst\xint_firstoftwo
900 \XINT_toks {\XINT_forfour_d #2{#6}}%
901 \expandafter\the\expandafter\XINT_toks #4jZ%
902 }%
903 \long\def\XINT_forfour_d #1#2#3(#4)#5%
904 {%
905 \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
906 \XINT_toks \expandafter{\romannumeral0\xintcsvtolist{ #4}}%
907 \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
908 \the\XINT_toks \csname XINT_for_right\the\numexpr#1+3\endcsname}%
909 \let\xintifForLast\xint_secondoftwo
910 \ifx #5j\let\xintifForLast\xint_firstoftwo\expandafter\xintBreakForAndDo\fi
911 \XINT_x\let\xintifForFirst\xint_secondoftwo\XINT_forfour_d #1{#2}%
912 }%
913 \catcode'Z 11
914 \catcode'j 11

```

33.24 \xintAssign, \xintAssignArray, \xintDigitsOf

\xintAssign {a}{b}..{z}\to A\B...Z,
\xintAssignArray {a}{b}..{z}\to U

version 1.01 corrects an oversight in 1.0 related to the value of \escapechar at the time of using \xintAssignArray or \xintRelaxArray These macros are non-expandable.

In version 1.05a I suddenly see some incongruous \expandafter's in (what is called now) \XINT_assignarray_end_c, which I remove.

Release 1.06 modifies the macros created by \xintAssignArray to feed their argument to a \numexpr. Release 1.06a detects an incredible typo in 1.01, (bad copy-paste from \xintRelaxArray) which caused \xintAssignArray to use #1 rather than the #2 as in the correct earlier 1.0 version!!! This went through undetected because \xint_arrayname, although weird, was still usable: the probability to overwrite something was almost zero. The bug got finally revealed doing \xintAssignArray {}{}{} \to \Stuff.

With release 1.06b an empty argument (or expanding to empty) to \xintAssignArray is ok.

1.09h simplifies the coding of \xintAssignArray (no more _end_a, _end_b, etc...), and no use of a \count register anymore, and uses \xintloop in \xintRelaxArray. Furthermore, macros are made long.

```

915 \long\def\xintAssign #1\to

```


33 Package *xinttools* implementation

```

916 {%
917   \expandafter\XINT_assign_a\romannumeral-'0#1{}\to
918 }%
919 \long\def\XINT_assign_a #1% attention to the # at the beginning of next line
920 {%
921   \def\xint_temp {#1}%
922   \ifx\empty\xint_temp
923     \expandafter\XINT_assign_b
924   \else
925     \expandafter\XINT_assign_B
926   \fi
927 }%
928 \long\def\XINT_assign_b #1#2\to #3%
929 {%
930   \edef #3{#1}\def\xint_temp {#2}%
931   \unless\ifx\empty\xint_temp\xint_afterfi{\XINT_assign_a #2\to }\fi
932 }%
933 \def\XINT_assign_B #1\to #2%
934 {%
935   \edef #2{\xint_temp}%
936 }%
937 \def\xintRelaxArray #1%
938 {%
939   \edef\XINT_restoreescapechar {\escapechar\the\escapechar\relax}%
940   \escapechar -1
941   \expandafter\def\expandafter\xint_arrayname\expandafter {\string #1}%
942   \XINT_restoreescapechar
943   \xintilooop [\csname\xint_arrayname 0\endcsname+-1]
944   \global
945   \expandafter\let\csname\xint_arrayname\xintilooopindex\endcsname\relax
946   \ifnum \xintilooopindex > \xint_c_
947   \repeat
948   \global\expandafter\let\csname\xint_arrayname 00\endcsname\relax
949   \global\let #1\relax
950 }%
951 \long\def\xintAssignArray #1\to #2%
952 {%
953   \edef\XINT_restoreescapechar {\escapechar\the\escapechar\relax}%
954   \escapechar -1
955   \expandafter\def\expandafter\xint_arrayname\expandafter {\string #2}%
956   \XINT_restoreescapechar
957   \def\xint_itemcount {0}%
958   \expandafter\XINT_assignarray_loop \romannumeral-'0#1\xint_relax
959   \csname\xint_arrayname 00\expandafter\endcsname
960   \csname\xint_arrayname 0\expandafter\endcsname
961   \expandafter {\xint_arrayname}#2%
962 }%
963 \long\def\XINT_assignarray_loop #1%
964 {%

```

```

965 \def\xint_temp {#1}%
966 \ifx\xint_brelax\xint_temp
967   \expandafter\def\csname\xint_arrayname 0\expandafter\endcsname
968     \expandafter{\the\numexpr\xint_itemcount}%
969   \expandafter\expandafter\expandafter\XINT_assignarray_end
970 \else
971   \expandafter\def\expandafter\xint_itemcount\expandafter
972     {\the\numexpr\xint_itemcount+\xint_c_i}%
973   \expandafter\edef
974     \csname\xint_arrayname\xint_itemcount\endcsname {\xint_temp }%
975   \expandafter\XINT_assignarray_loop
976 \fi
977 }%
978 \def\XINT_assignarray_end #1#2#3#4%
979 {%
980   \def #4##1%
981   {%
982     \romannumeral0\expandafter #1\expandafter{\the\numexpr ##1}%
983   }%
984   \def #1##1%
985   {%
986     \ifnum ##1<\xint_c_
987       \xint_afterfi {\xintError:ArrayIndexIsNegative\space }%
988     \else
989       \xint_afterfi {%
990         \ifnum ##1>#2
991           \xint_afterfi {\xintError:ArrayIndexBeyondLimit\space }%
992         \else\xint_afterfi
993       {\expandafter\expandafter\expandafter\space\csname #3##1\endcsname}%
994       \fi}%
995     \fi
996   }%
997 }%
998 \let\xintDigitsOf\xintAssignArray
999 \XINT_restorecatcodes_endinput%

```

34 Package **xint** implementation

With release 1.09a all macros doing arithmetic operations and a few more apply systematically `\xint-` 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

.1	Catcodes, ε -TeX and reload detection . . 154	.2	Confirmation of xinttools loading . . 155
----	---	----	--

.3	Catcodes	156	.35	\xintORof.....	186
.4	Package identification	156	.36	\xintORof:csv.....	187
.5	Token management, constants	156	.37	\xintXORof.....	187
.6	\xintRev.....	157	.38	\xintXORof:csv.....	187
.7	\xintLen.....	157	.39	\xintGeq.....	188
.8	\XINT_RQ.....	158	.40	\xintMax.....	189
.9	\XINT_cuz.....	160	.41	\xintMaxof.....	191
.10	\xintIsOne.....	161	.42	\xintMin.....	191
.11	\xintNum.....	161	.43	\xintMinof.....	192
.12	\xintSgn.....	162	.44	\xintSum, \xintSumExpr.....	193
.13	\xintBool, \xintToggle.....	163	.45	\xintMul.....	194
.14	\xintSgnFork.....	163	.46	\xintSqr.....	203
.15	\xintifSgn.....	163	.47	\xintPrd, \xintPrdExpr.....	203
.16	\xintifZero, \xintifNotZero.....	164	.48	\xintFac.....	204
.17	\xintifTrueFalse.....	164	.49	\xintPow.....	206
.18	\xintifCmp.....	164	.50	\xintDivision, \xintQuo, \xintRem	210
.19	\xintifEq.....	165	.51	\xintFDg.....	223
.20	\xintifGt.....	165	.52	\xintLDg.....	223
.21	\xintifLt.....	165	.53	\xintMON, \xintMMON.....	224
.22	\xintifOdd.....	165	.54	\xintOdd.....	224
.23	\xintOpp.....	166	.55	\xintDSL.....	225
.24	\xintAbs.....	166	.56	\xintDSR.....	225
.25	\xintAdd.....	175	.57	\xintDSH, \xintDSHr.....	226
.26	\xintSub.....	176	.58	\xintDSx.....	227
.27	\xintCmp.....	182	.59	\xintDecSplit, \xintDecSplitL,	
.28	\xintEq, \xintGt, \xintLt.....	185		\xintDecSplitR.....	230
.29	\xintIsZero, \xintIsNotZero.....	185	.60	\xintDouble.....	233
.30	\xintIsTrue, \xintNot.....	185	.61	\xintHalf.....	234
.31	\xintIsTrue:csv.....	185	.62	\xintDec.....	235
.32	\xintAND, \xintOR, \xintXOR.....	185	.63	\xintInc.....	236
.33	\xintANDof.....	186	.64	\xintiSqrt, \xintiSquareRoot	237
.34	\xintANDof:csv.....	186			

34.1 Catcodes, ε -T_EX 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.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2 \catcode13=5 % ^^M
3 \endlinechar=13 %
4 \catcode123=1 % {
5 \catcode125=2 % }
6 \catcode64=11 % @

```

```

7 \catcode35=6 % #
8 \catcode44=12 % ,
9 \catcode45=12 % -
10 \catcode46=12 % .
11 \catcode58=12 % :
12 \def\space { }%
13 \let\z\endgroup
14 \expandafter\let\expandafter\x\csname ver@xint.sty\endcsname
15 \expandafter\let\expandafter\w\csname ver@xinttools.sty\endcsname
16 \expandafter
17 \ifx\csname PackageInfo\endcsname\relax
18 \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
19 \else
20 \def\y#1#2{\PackageInfo{#1}{#2}}%
21 \fi
22 \expandafter
23 \ifx\csname numexpr\endcsname\relax
24 \y{xint}{\numexpr not available, aborting input}%
25 \aftergroup\endinput
26 \else
27 \ifx\x\relax % plain-TeX, first loading of xint.sty
28 \ifx\w\relax % but xinttools.sty not yet loaded.
29 \y{xint}{Package xinttools is required}%
30 \y{xint}{Will try \string\input\space xinttools.sty}%
31 \def\z{\endgroup\input xinttools.sty\relax}%
32 \fi
33 \else
34 \def\empty {}%
35 \ifx\x\empty % LaTeX, first loading,
36 % variable is initialized, but \ProvidesPackage not yet seen
37 \ifx\w\relax % xinttools.sty not yet loaded.
38 \y{xint}{Package xinttools is required}%
39 \y{xint}{Will try \string\RequirePackage{xinttools}}%
40 \def\z{\endgroup\RequirePackage{xinttools}}%
41 \fi
42 \else
43 \y{xint}{I was already loaded, aborting input}%
44 \aftergroup\endinput
45 \fi
46 \fi
47 \fi
48 \z%

```

34.2 Confirmation of *xinttools* loading

```

49 \begingroup\catcode61\catcode48\catcode32=10\relax%
50 \catcode13=5 % ^^M
51 \endlinechar=13 %
52 \catcode123=1 % {

```

```

53 \catcode125=2 % }
54 \catcode64=11 % @
55 \catcode35=6 % #
56 \catcode44=12 % ,
57 \catcode45=12 % -
58 \catcode46=12 % .
59 \catcode58=12 % :
60 \ifdefined\PackageInfo
61 \def\y#1#2{\PackageInfo{#1}{#2}}%
62 \else
63 \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
64 \fi
65 \def\empty {}%
66 \expandafter\let\expandafter\w\csname ver@xinttools.sty\endcsname
67 \ifx\w\relax % Plain TeX, user gave a file name at the prompt
68 \y{xint}{Loading of package xinttools failed, aborting input}%
69 \aftergroup\endinput
70 \fi
71 \ifx\w\empty % LaTeX, user gave a file name at the prompt
72 \y{xint}{Loading of package xinttools failed, aborting input}%
73 \aftergroup\endinput
74 \fi
75 \endgroup%

```

34.3 Catcodes

```
76 \XINTsetupcatcodes%
```

34.4 Package identification

```

77 \XINT_providespackage
78 \ProvidesPackage{xint}%
79 [2013/11/28 v1.09h Expandable operations on long numbers (jfb)]%

```

34.5 Token management, constants

```

80 \long\def\xint_firstofthree #1#2#3{#1}%
81 \long\def\xint_secondofthree #1#2#3{#2}%
82 \long\def\xint_thirdofthree #1#2#3{#3}%
83 \def\xint_gob_til_zero #10{}%
84 \def\xint_gob_til_zeros_iii #1000{}%
85 \def\xint_gob_til_zeros_iv #10000{}%
86 \def\xint_gob_til_one #11{}%
87 \def\xint_gob_til_G #1G{}%
88 \def\xint_gob_til_minus #1-{}%
89 \def\xint_gob_til_relax #1\relax {}%
90 \def\xint_exchangetwo_keepbraces_andstop #1#2{ {#2}{#1}}%
91 \def\xint_UDzerofork #10\dummy #2#3\krof {#2}%
92 \def\xint_UDsignfork #1-\dummy #2#3\krof {#2}%
93 \def\xint_UDwfork #1W\dummy #2#3\krof {#2}%
94 \def\xint_UDzerosfork #100\dummy #2#3\krof {#2}%

```

```

95 \def\xint_UDonezerofork #1\0\dummy #2#3\krof {#2}%
96 \def\xint_UDzerominusfork #1\0-\dummy #2#3\krof {#2}%
97 \def\xint_UDsignsfork #1--\dummy #2#3\krof {#2}%
98 \chardef\xint_c_i 1 % 0 and 8 in xinttools
99 \chardef\xint_c_ii 2
100 \chardef\xint_c_iii 3
101 \chardef\xint_c_iv 4
102 \chardef\xint_c_v 5
103 \chardef\xint_c_ix 9
104 \chardef\xint_c_x 10
105 \newcount\xint_c_x^viii \xint_c_x^viii 100000000

```

34.6 \xintRev

\xintRev: expands fully its argument \romannumeral-‘0, and checks the sign. However this last aspect does not appear like a very useful thing. And despite the fact that a special check is made for a sign, actually the input is not given to \xintnum, contrarily to \xintLen. This is all a bit incoherent. Should be fixed.

```

106 \def\xintRev {\romannumeral0\xintrev}%
107 \def\xintrev #1%
108 {%
109   \expandafter\XINT_rev_fork
110   \romannumeral-‘0#1\xint_relax % empty #1 ok, \xint_relax stops expansion
111   \xint_bye\xint_bye\xint_bye\xint_bye
112   \xint_bye\xint_bye\xint_bye\xint_bye
113   \xint_relax
114}%
115 \def\XINT_rev_fork #1%
116 {%
117   \xint_UDsignfork
118   #1\dummy {\expandafter\xint_minus_andstop\romannumeral0\XINT_rord_main {}}%
119   -\dummy {\XINT_rord_main {}}#1}%
120   \krof
121}%

```

34.7 \xintLen

\xintLen is ONLY for (possibly long) integers. Gets extended to fractions by xintfrac.sty

```

122 \def\xintLen {\romannumeral0\xintlen}%
123 \def\xintlen #1%
124 {%
125   \expandafter\XINT_len_fork
126   \romannumeral0\xintnum{#1}\xint_relax\xint_relax\xint_relax\xint_relax
127   \xint_relax\xint_relax\xint_relax\xint_relax\xint_bye
128}%
129 \def\XINT_Len #1% variant which does not expand via \xintnum.

```

```

130 {%
131   \romannumeral0\XINT_len_fork
132   #1\xint_relax\xint_relax\xint_relax\xint_relax
133   \xint_relax\xint_relax\xint_relax\xint_relax\xint_bye
134 }%
135 \def\XINT_len_fork #1%
136 {%
137   \expandafter\XINT_length_loop
138   \xint_UDsignfork
139   #1\dummy {{0}}%
140   -\dummy {{0}}#1}%
141   \krof
142 }%

```

34.8 \XINT_RQ

cette macro renverse et ajoute le nombre minimal de zéros à la fin pour que la longueur soit alors multiple de 4

\romannumeral0\XINT_RQ {}<le truc à renverser>\R\R\R\R\R\R\R\R\Z

Attention, ceci n'est utilisé que pour des chaînes de chiffres, et donc le comportement avec des {...} ou autres espaces n'a fait l'objet d'aucune attention

```

143 \def\XINT_RQ #1#2#3#4#5#6#7#8#9%
144 {%
145   \xint_gob_til_R #9\XINT_RQ_end_a\R\XINT_RQ {#9#8#7#6#5#4#3#2#1}%
146 }%
147 \def\XINT_RQ_end_a\R\XINT_RQ #1#2\Z
148 {%
149   \XINT_RQ_end_b #1\Z
150 }%
151 \def\XINT_RQ_end_b #1#2#3#4#5#6#7#8%
152 {%
153   \xint_gob_til_R
154   #8\XINT_RQ_end_viii
155   #7\XINT_RQ_end_vii
156   #6\XINT_RQ_end_vi
157   #5\XINT_RQ_end_v
158   #4\XINT_RQ_end_iv
159   #3\XINT_RQ_end_iii
160   #2\XINT_RQ_end_ii
161   \R\XINT_RQ_end_i
162   \Z #2#3#4#5#6#7#8%
163 }%
164 \def\XINT_RQ_end_viii #1\Z #2#3#4#5#6#7#8#9\Z { #9}%
165 \def\XINT_RQ_end_vii #1\Z #2#3#4#5#6#7#8#9\Z { #8#9000}%
166 \def\XINT_RQ_end_vi #1\Z #2#3#4#5#6#7#8#9\Z { #7#8#900}%
167 \def\XINT_RQ_end_v #1\Z #2#3#4#5#6#7#8#9\Z { #6#7#8#90}%
168 \def\XINT_RQ_end_iv #1\Z #2#3#4#5#6#7#8#9\Z { #5#6#7#8#9}%
169 \def\XINT_RQ_end_iii #1\Z #2#3#4#5#6#7#8#9\Z { #4#5#6#7#8#9000}%

```


34 Package *xint* implementation

```

170 \def\XINT_RQ_end_ii    #1\Z #2#3#4#5#6#7#8#9\Z { #3#4#5#6#7#8#900}%
171 \def\XINT_RQ_end_i     \Z #1#2#3#4#5#6#7#8\Z { #1#2#3#4#5#6#7#80}%
172 \def\XINT_SQ #1#2#3#4#5#6#7#8%
173 {%
174     \xint_gob_til_R #8\XINT_SQ_end_a\R\XINT_SQ {#8#7#6#5#4#3#2#1}%
175}%
176 \def\XINT_SQ_end_a\R\XINT_SQ #1#2\Z
177 {%
178     \XINT_SQ_end_b #1\Z
179}%
180 \def\XINT_SQ_end_b #1#2#3#4#5#6#7%
181 {%
182     \xint_gob_til_R
183         #7\XINT_SQ_end_vii
184         #6\XINT_SQ_end_vi
185         #5\XINT_SQ_end_v
186         #4\XINT_SQ_end_iv
187         #3\XINT_SQ_end_iii
188         #2\XINT_SQ_end_ii
189         \R\XINT_SQ_end_i
190         \Z #2#3#4#5#6#7%
191}%
192 \def\XINT_SQ_end_vii    #1\Z #2#3#4#5#6#7#8\Z { #8}%
193 \def\XINT_SQ_end_vi     #1\Z #2#3#4#5#6#7#8\Z { #7#8000000}%
194 \def\XINT_SQ_end_v      #1\Z #2#3#4#5#6#7#8\Z { #6#7#800000}%
195 \def\XINT_SQ_end_iv     #1\Z #2#3#4#5#6#7#8\Z { #5#6#7#80000}%
196 \def\XINT_SQ_end_iii    #1\Z #2#3#4#5#6#7#8\Z { #4#5#6#7#8000}%
197 \def\XINT_SQ_end_ii     #1\Z #2#3#4#5#6#7#8\Z { #3#4#5#6#7#800}%
198 \def\XINT_SQ_end_i      \Z #1#2#3#4#5#6#7\Z { #1#2#3#4#5#6#70}%
199 \def\XINT_OQ #1#2#3#4#5#6#7#8#9%
200 {%
201     \xint_gob_til_R #9\XINT_OQ_end_a\R\XINT_OQ {#9#8#7#6#5#4#3#2#1}%
202}%
203 \def\XINT_OQ_end_a\R\XINT_OQ #1#2\Z
204 {%
205     \XINT_OQ_end_b #1\Z
206}%
207 \def\XINT_OQ_end_b #1#2#3#4#5#6#7#8%
208 {%
209     \xint_gob_til_R
210         #8\XINT_OQ_end_viii
211         #7\XINT_OQ_end_vii
212         #6\XINT_OQ_end_vi
213         #5\XINT_OQ_end_v
214         #4\XINT_OQ_end_iv
215         #3\XINT_OQ_end_iii
216         #2\XINT_OQ_end_ii
217         \R\XINT_OQ_end_i
218         \Z #2#3#4#5#6#7#8%

```

```

219 }%
220 \def\XINT_OQ_end_viii #1\Z #2#3#4#5#6#7#8#9\Z { #9}%
221 \def\XINT_OQ_end_vii #1\Z #2#3#4#5#6#7#8#9\Z { #8#90000000}%
222 \def\XINT_OQ_end_vi #1\Z #2#3#4#5#6#7#8#9\Z { #7#8#9000000}%
223 \def\XINT_OQ_end_v #1\Z #2#3#4#5#6#7#8#9\Z { #6#7#8#900000}%
224 \def\XINT_OQ_end_iv #1\Z #2#3#4#5#6#7#8#9\Z { #5#6#7#8#90000}%
225 \def\XINT_OQ_end_iii #1\Z #2#3#4#5#6#7#8#9\Z { #4#5#6#7#8#9000}%
226 \def\XINT_OQ_end_ii #1\Z #2#3#4#5#6#7#8#9\Z { #3#4#5#6#7#8#900}%
227 \def\XINT_OQ_end_i \Z #1#2#3#4#5#6#7#8\Z { #1#2#3#4#5#6#7#80}%

```

34.9 \XINT_cuz

```

228 \def\xint_cleanupzeros_andstop #1#2#3#4%
229 {%
230   \expandafter\space\the\numexpr #1#2#3#4\relax
231 }%
232 \def\xint_cleanupzeros_nospace #1#2#3#4%
233 {%
234   \the\numexpr #1#2#3#4\relax
235 }%
236 \def\XINT_rev_andcuz #1%
237 {%
238   \expandafter\xint_cleanupzeros_andstop
239   \romannumeral0\XINT_rord_main {}#1%
240   \xint_relax
241   \xint_bye\xint_bye\xint_bye\xint_bye
242   \xint_bye\xint_bye\xint_bye\xint_bye
243   \xint_relax
244 }%

```

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

```

245 \def\XINT_cuz #1%
246 {%
247   \XINT_cuz_loop #1\W\W\W\W\W\W\W\Z%
248 }%
249 \def\XINT_cuz_loop #1#2#3#4#5#6#7#8%
250 {%
251   \xint_gob_til_W #8\xint_cuz_end_a\W
252   \xint_gob_til_Z #8\xint_cuz_end_A\Z
253   \XINT_cuz_check_a {#1#2#3#4#5#6#7#8}%
254 }%
255 \def\xint_cuz_end_a #1\XINT_cuz_check_a #2%
256 {%
257   \xint_cuz_end_b #2%
258 }%

```

```

259 \def\xint_cuz_end_b #1#2#3#4#5\Z
260 {%
261   \expandafter\space\the\numexpr #1#2#3#4\relax
262 }%
263 \def\xint_cuz_end_A \Z\XINT_cuz_check_a #1{ 0}%
264 \def\XINT_cuz_check_a #1%
265 {%
266   \expandafter\XINT_cuz_check_b\the\numexpr #1\relax
267 }%
268 \def\XINT_cuz_check_b #1%
269 {%
270   \xint_gob_til_zero #1\xint_cuz_backtoloop 0\XINT_cuz_stop #1%
271 }%
272 \def\XINT_cuz_stop #1\W #2\Z{ #1}%
273 \def\xint_cuz_backtoloop 0\XINT_cuz_stop 0{\XINT_cuz_loop }%

```

34.10 \xintIsOne

Added in 1.03. Attention: \XINT_isOne does not do any expansion. Release 1.09a defines \xintIsOne which is more user-friendly. Will be modified if xintfracis loaded.

```

274 \def\xintIsOne {\romannumeral0\xintisone }%
275 \def\xintisone #1{\expandafter\XINT_isone \romannumeral0\xintnum{#1}\W\Z }%
276 \def\XINT_isOne #1{\romannumeral0\XINT_isone #1\W\Z }%
277 \def\XINT_isone #1#2%
278 {%
279   \xint_gob_til_one #1\XINT_isone_b 1%
280   \expandafter\space\expandafter 0\xint_gob_til_Z #2%
281 }%
282 \def\XINT_isone_b #1\xint_gob_til_Z #2%
283 {%
284   \xint_gob_til_W #2\XINT_isone_yes \W
285   \expandafter\space\expandafter 0\xint_gob_til_Z
286 }%
287 \def\XINT_isone_yes #1\Z { 1}%

```

34.11 \xintNum

For example \xintNum {-----0000000000000003}

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).

```

288 \def\xintiNum {\romannumeral0\xintinum }%

```

```

289 \def\xintinum #1%
290 {%
291   \expandafter\XINT_num_loop
292   \romannumeral-‘0#1\xint_relax\xint_relax\xint_relax\xint_relax
293   \xint_relax\xint_relax\xint_relax\xint_relax\Z
294 }%
295 \let\xintNum\xintiNum \let\xintnum\xintinum
296 \def\XINT_num #1%
297 {%
298   \XINT_num_loop #1\xint_relax\xint_relax\xint_relax\xint_relax
299   \xint_relax\xint_relax\xint_relax\xint_relax\Z
300 }%
301 \def\XINT_num_loop #1#2#3#4#5#6#7#8%
302 {%
303   \xint_gob_til_xint_relax #8\XINT_num_end\xint_relax
304   \XINT_num_NumEight #1#2#3#4#5#6#7#8%
305 }%
306 \def\XINT_num_end\xint_relax\XINT_num_NumEight #1\xint_relax #2\Z
307 {%
308   \expandafter\space\the\numexpr #1+0\relax
309 }%
310 \def\XINT_num_NumEight #1#2#3#4#5#6#7#8%
311 {%
312   \ifnum \numexpr #1#2#3#4#5#6#7#8+0= 0
313     \xint_afterfi {\expandafter\XINT_num_keepsign_a
314     \the\numexpr #1#2#3#4#5#6#7#81\relax}%
315   \else
316     \xint_afterfi {\expandafter\XINT_num_finish
317     \the\numexpr #1#2#3#4#5#6#7#8\relax}%
318   \fi
319 }%
320 \def\XINT_num_keepsign_a #1%
321 {%
322   \xint_gob_til_one#1\XINT_num_gobackto loop 1\XINT_num_keepsign_b
323 }%
324 \def\XINT_num_gobackto loop 1\XINT_num_keepsign_b {\XINT_num_loop }%
325 \def\XINT_num_keepsign_b #1{\XINT_num_loop -}%
326 \def\XINT_num_finish #1\xint_relax #2\Z { #1}%

```

34.12 \xintSgn

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

```

327 \def\xintiiSgn {\romannumeral0\xintiisgn }%
328 \def\xintiisgn #1%
329 {%
330   \expandafter\XINT_sgn \romannumeral-‘0#1\Z%
331 }%
332 \def\xintSgn {\romannumeral0\xintsgn }%

```

```

333 \def\xintsgn #1%
334 {%
335   \expandafter\XINT_sgn \romannumeral0\xintnum{#1}\Z%
336 }%
337 \def\XINT_Sgn #1{\romannumeral0\XINT_sgn #1\Z }%
338 \def\XINT_sgn #1#2\Z
339 {%
340   \xint_UDzerominusfork
341   #1-\dummy { 0}%
342   0#1\dummy { -1}%
343   0-\dummy { 1}%
344   \krof
345 }%

```

34.13 \xintBool, \xintToggle

1.09c

```

346 \def\xintBool #1{\romannumeral-‘0%
347   \csname if#1\endcsname\expandafter1\else\expandafter0\fi }%
348 \def\xintToggle #1{\romannumeral-‘0\iftoggle{#1}{1}{0}}%

```

34.14 \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.

```

349 \def\xintSgnFork {\romannumeral0\xintsgnfork }%
350 \def\xintsgnfork #1%
351 {%
352   \ifcase #1 \xint_afterfi{\expandafter\space\xint_secondofthree}%
353   \or\xint_afterfi{\expandafter\space\xint_thirdofthree}%
354   \else\xint_afterfi{\expandafter\space\xint_firstofthree}%
355   \fi
356 }%

```

34.15 \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 for matters related of the transformation of the ternary operator : in \xintNew-Expr

```

357 \def\xintifSgn {\romannumeral0\xintifsgn }%
358 \def\xintifsgn #1%
359 {%
360   \ifcase \romannumeral0\xintsgn{#1}
361     \xint_afterfi{\expandafter\space\xint_secondofthree}%

```

```

362         \or\xint_afterfi{\expandafter\space\xint_thirdofthree}%
363         \else\xint_afterfi{\expandafter\space\xint_firstofthree}%
364     \fi
365 }%

```

34.16 \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).

```

366 \def\xintifZero {\romannumeral0\xintifzero }%
367 \def\xintifzero #1%
368 {%
369     \if\xintSgn{\xintAbs{#1}}0%
370         \xint_afterfi{\expandafter\space\xint_firstoftwo}%
371     \else
372         \xint_afterfi{\expandafter\space\xint_secondoftwo}%
373     \fi
374 }%
375 \def\xintifNotZero {\romannumeral0\xintifnotzero }%
376 \def\xintifnotzero #1%
377 {%
378     \if\xintSgn{\xintAbs{#1}}1%
379         \xint_afterfi{\expandafter\space\xint_firstoftwo}%
380     \else
381         \xint_afterfi{\expandafter\space\xint_secondoftwo}%
382     \fi
383 }%

```

34.17 \xintifTrueFalse

```

384 \let\xintifTrue\xintifNotZero
385 \let\xintifTrueFalse\xintifNotZero

```

34.18 \xintifCmp

1.09e \xintifCmp {n}{m}{if n<m}{if n=m}{if n>m}.

```

386 \def\xintifCmp {\romannumeral0\xintifcmp }%
387 \def\xintifcmp #1#2%
388 {%
389     \ifcase \xintCmp {#1}{#2}
390         \xint_afterfi{\expandafter\space\xint_secondofthree}%
391     \or\xint_afterfi{\expandafter\space\xint_thirdofthree}%
392     \else\xint_afterfi{\expandafter\space\xint_firstofthree}%
393     \fi
394 }%

```

34.19 \xintifEq

1.09a \xintifEq {n}{m}{YES if n=m}{NO if n<>m}.

```

395 \def\xintifEq {\romannumeral0\xintifeq }%
396 \def\xintifeq #1#2%
397 {%
398   \if\xintCmp{#1}{#2}0%
399       \xint_afterfi{\expandafter\space\xint_firstoftwo}%
400       \else\xint_afterfi{\expandafter\space\xint_secondoftwo}%
401   \fi
402 }%
```

34.20 \xintifGt

1.09a \xintifGt {n}{m}{YES if n>m}{NO if n<=m}.

```

403 \def\xintifGt {\romannumeral0\xintifgt }%
404 \def\xintifgt #1#2%
405 {%
406   \if\xintCmp{#1}{#2}1%
407       \xint_afterfi{\expandafter\space\xint_firstoftwo}%
408       \else\xint_afterfi{\expandafter\space\xint_secondoftwo}%
409   \fi
410 }%
```

34.21 \xintifLt

1.09a \xintifLt {n}{m}{YES if n<m}{NO if n>=m}.

```

411 \def\xintifLt {\romannumeral0\xintiflt }%
412 \def\xintiflt #1#2%
413 {%
414   \xintSgnFork{\xintCmp{#1}{#2}}%
415       {\expandafter\space\xint_firstoftwo}%
416       {\expandafter\space\xint_secondoftwo}%
417       {\expandafter\space\xint_secondoftwo}%
418 }%
```

34.22 \xintifOdd

1.09e

```

419 \def\xintifOdd {\romannumeral0\xintifodd }%
420 \def\xintifodd #1%
421 {%
422   \if\xintOdd{#1}1%
423       \xint_afterfi{\expandafter\space\xint_firstoftwo}%

```

```

424 \else
425 \xint_afterfi{\expandafter\space\xint_seconddoftwo}%
426 \fi
427 }%

```

34.23 \xintOpp

\xintnum added in 1.09a

```

428 \def\xintiiOpp {\romannumeral0\xintiiopp }%
429 \def\xintiiopp #1%
430 {%
431 \expandafter\XINT_opp \romannumeral-‘0#1%
432 }%
433 \def\xintiOpp {\romannumeral0\xintiopp }%
434 \def\xintiopp #1%
435 {%
436 \expandafter\XINT_opp \romannumeral0\xintnum{#1}%
437 }%
438 \let\xintOpp\xintiOpp \let\xintopp\xintiopp
439 \def\XINT_Opp #1{\romannumeral0\XINT_opp #1}%
440 \def\XINT_opp #1%
441 {%
442 \xint_UDzerominusfork
443 #1-\dummy { 0}% zero
444 0#1\dummy { }% negative
445 0-\dummy { -#1}% positive
446 \krof
447 }%

```

34.24 \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.

```

448 \def\xintiiAbs {\romannumeral0\xintiiabs }%
449 \def\xintiiabs #1%
450 {%
451 \expandafter\XINT_abs \romannumeral-‘0#1%
452 }%
453 \def\xintiAbs {\romannumeral0\xintiabs }%
454 \def\xintiabs #1%
455 {%
456 \expandafter\XINT_abs \romannumeral0\xintnum{#1}%
457 }%
458 \let\xintAbs\xintiAbs \let\xintabs\xintiabs
459 \def\XINT_Abs #1{\romannumeral0\XINT_abs #1}%

```



```

460 \def\XINT_abs #1%
461 {%
462   \xint_UDsignfork
463   #1\dummy { }%
464   -\dummy { #1}%
465   \krof
466 }%

```

 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.

```

467 \def\XINT_add_A #1#2#3#4#5#6%
468 {%
469   \xint_gob_til_W #3\xint_add_az\W
470   \XINT_add_AB #1{#3#4#5#6}{#2}%
471 }%
472 \def\xint_add_az\W\XINT_add_AB #1#2%
473 {%
474   \XINT_add_AC_checkcarry #1%
475 }%

```

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.

```

476 \def\XINT_add_AB #1#2#3#4\W\X\Y\Z #5#6#7#8%

```

34 Package *xint* implementation

```

477 {%
478   \xint_gob_til_W #5\xint_add_bz\W
479   \XINT_add_ABE #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
480 }%
481 \def\XINT_add_ABE #1#2#3#4#5#6%
482 {%
483   \expandafter\XINT_add_ABEA\the\numexpr #1+10#5#4#3#2+#6.%
484 }%
485 \def\XINT_add_ABEA #1#2#3.#4%
486 {%
487   \XINT_add_A #2{#3#4}%
488 }%

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

489 \def\xint_add_bz\W\XINT_add_ABE #1#2#3#4#5#6%
490 {%
491   \expandafter\XINT_add_CC\the\numexpr #1+10#5#4#3#2.%
492 }%
493 \def\XINT_add_CC #1#2#3.#4%
494 {%
495   \XINT_add_AC_checkcarry #2{#3#4}% on va examiner et \'eliminer #2
496 }%

retenue plus chiffres qui restent de l'un des deux nombres. #2 = résultat partiel
#3#4#5#6 = summand, avec plus significatif à droite

497 \def\XINT_add_AC_checkcarry #1%
498 {%
499   \xint_gob_til_zero #1\xint_add_AC_nocarry 0\XINT_add_C
500 }%
501 \def\xint_add_AC_nocarry 0\XINT_add_C #1#2\W\X\Y\Z
502 {%
503   \expandafter
504   \xint_cleanupzeros_andstop
505   \romannumeral0%
506   \XINT_rord_main {}#2%
507   \xint_relax
508   \xint_bye\xint_bye\xint_bye\xint_bye
509   \xint_bye\xint_bye\xint_bye\xint_bye
510   \xint_relax
511   #1%
512 }%
513 \def\XINT_add_C #1#2#3#4#5%
514 {%
515   \xint_gob_til_W #2\xint_add_cz\W
516   \XINT_add_CD {#5#4#3#2}{#1}%
517 }%
518 \def\XINT_add_CD #1%

```

```

519 {%
520   \expandafter\XINT_addr_CC\the\numexpr 1+10#1.%
521 }%
522 \def\xint_addr_cz\W\XINT_addr_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_addr_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_addr_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*

523 \def\XINT_addr_A #1#2#3#4#5#6%
524 {%
525   \xint_gob_til_W #3\xint_addr_az\W
526   \XINT_addr_B #1{#3#4#5#6}{#2}%
527 }%
528 \def\xint_addr_az\W\XINT_addr_B #1#2%
529 {%
530   \XINT_addr_AC_checkcarry #1%
531 }%
532 \def\XINT_addr_B #1#2#3#4\W\X\Y\Z #5#6#7#8%
533 {%
534   \xint_gob_til_W #5\xint_addr_bz\W
535   \XINT_addr_E #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
536 }%
537 \def\XINT_addr_E #1#2#3#4#5#6%
538 {%
539   \expandafter\XINT_addr_ABEA\the\numexpr #1+10#5#4#3#2+#6\relax
540 }%
541 \def\XINT_addr_ABEA #1#2#3#4#5#6#7%
542 {%
543   \XINT_addr_A #2{#7#6#5#4#3}%
544 }%
545 \def\xint_addr_bz\W\XINT_addr_E #1#2#3#4#5#6%
546 {%
547   \expandafter\XINT_addr_CC\the\numexpr #1+10#5#4#3#2\relax
548 }%
549 \def\XINT_addr_CC #1#2#3#4#5#6#7%
550 {%
551   \XINT_addr_AC_checkcarry #2{#7#6#5#4#3}%
552 }%
553 \def\XINT_addr_AC_checkcarry #1%
554 {%
555   \xint_gob_til_zero #1\xint_addr_AC_nocarry 0\XINT_addr_C

```

```

556 }%
557 \def\xint_addr_AC_nocarry 0\XINT_addr_C #1#2\W\X\Y\Z { #1#2}%
558 \def\XINT_addr_C #1#2#3#4#5%
559 {%
560   \xint_gob_til_W #2\xint_addr_cz\W
561   \XINT_addr_D {#5#4#3#2}{#1}%
562 }%
563 \def\XINT_addr_D #1%
564 {%
565   \expandafter\XINT_addr_CC\the\numexpr 1+10#1\relax
566 }%
567 \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. Util-
  isé par la multiplication.

568 \def\XINT_addm_A #1#2#3#4#5#6%
569 {%
570   \xint_gob_til_W #3\xint_addm_az\W
571   \XINT_addm_AB #1{#3#4#5#6}{#2}%
572 }%
573 \def\xint_addm_az\W\XINT_addm_AB #1#2%
574 {%
575   \XINT_addm_AC_checkcarry #1%
576 }%
577 \def\XINT_addm_AB #1#2#3#4\W\X\Y\Z #5#6#7#8%
578 {%
579   \XINT_addm_ABE #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
580 }%
581 \def\XINT_addm_ABE #1#2#3#4#5#6%
582 {%
583   \expandafter\XINT_addm_ABEA\the\numexpr #1+10#5#4#3#2+#6.%
584 }%
585 \def\XINT_addm_ABEA #1#2#3.#4%
586 {%
587   \XINT_addm_A #2{#3#4}%
588 }%
589 \def\XINT_addm_AC_checkcarry #1%
590 {%
591   \xint_gob_til_zero #1\xint_addm_AC_nocarry 0\XINT_addm_C
592 }%
593 \def\xint_addm_AC_nocarry 0\XINT_addm_C #1#2\W\X\Y\Z
594 {%
595   \expandafter
596   \xint_cleanupzeros_andstop

```

```

597 \romannumeral0%
598 \XINT_rord_main {}#2%
599 \xint_relax
600 \xint_bye\xint_bye\xint_bye\xint_bye
601 \xint_bye\xint_bye\xint_bye\xint_bye
602 \xint_relax
603 #1%
604 }%
605 \def\XINT_addm_C #1#2#3#4#5%
606 {%
607 \xint_gob_til_W
608 #5\xint_addm_cw
609 #4\xint_addm_cx
610 #3\xint_addm_cy
611 #2\xint_addm_cz
612 \W\XINT_addm_CD {#5#4#3#2}{#1}%
613 }%
614 \def\XINT_addm_CD #1%
615 {%
616 \expandafter\XINT_addm_CC\the\numexpr 1+10#1.%
617 }%
618 \def\XINT_addm_CC #1#2#3.#4%
619 {%
620 \XINT_addm_AC_checkcarry #2{#3#4}%
621 }%
622 \def\xint_addm_cw
623 #1\xint_addm_cx
624 #2\xint_addm_cy
625 #3\xint_addm_cz
626 \W\XINT_addm_CD
627 {%
628 \expandafter\XINT_addm_CDw\the\numexpr 1+#1#2#3.%
629 }%
630 \def\XINT_addm_CDw #1.#2#3\X\Y\Z
631 {%
632 \XINT_addm_end #1#3%
633 }%
634 \def\xint_addm_cx
635 #1\xint_addm_cy
636 #2\xint_addm_cz
637 \W\XINT_addm_CD
638 {%
639 \expandafter\XINT_addm_CDx\the\numexpr 1+#1#2.%
640 }%
641 \def\XINT_addm_CDx #1.#2#3\Y\Z
642 {%
643 \XINT_addm_end #1#3%
644 }%
645 \def\xint_addm_cy

```

```

646 #1\xint_addm_cz
647 \W\XINT_addm_CD
648 {%
649 \expandafter\XINT_addm_CDy\the\numexpr 1+#1.%
650}%
651 \def\XINT_addm_CDy #1.#2#3\Z
652 {%
653 \XINT_addm_end #1#3%
654}%
655 \def\xint_addm_cz\W\XINT_addm_CD #1#2#3{\XINT_addm_end #1#3}%
656 \def\XINT_addm_end #1#2#3#4#5%
657 {\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.

658 \def\XINT_addp_A #1#2#3#4#5#6%
659 {%
660 \xint_gob_til_W #3\xint_addp_az\W
661 \XINT_addp_AB #1{#3#4#5#6}{#2}%
662}%
663 \def\xint_addp_az\W\XINT_addp_AB #1#2%
664 {%
665 \XINT_addp_AC_checkcarry #1%
666}%
667 \def\XINT_addp_AC_checkcarry #1%
668 {%
669 \xint_gob_til_zero #1\xint_addp_AC_nocarry 0\XINT_addp_C
670}%
671 \def\xint_addp_AC_nocarry 0\XINT_addp_C
672 {%
673 \XINT_addp_F
674}%
675 \def\XINT_addp_AB #1#2#3#4\W\X\Y\Z #5#6#7#8%
676 {%
677 \XINT_addp_ABE #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
678}%
679 \def\XINT_addp_ABE #1#2#3#4#5#6%
680 {%
681 \expandafter\XINT_addp_ABEA\the\numexpr #1+10#5#4#3#2+#6\relax
682}%
683 \def\XINT_addp_ABEA #1#2#3#4#5#6#7%
684 {%
685 \XINT_addp_A #2{#7#6#5#4#3}%<-- attention on met donc \'a droite

```

```

686 }%
687 \def\XINT_addp_C #1#2#3#4#5%
688 {%
689     \xint_gob_til_W
690     #5\xint_addp_cw
691     #4\xint_addp_cx
692     #3\xint_addp_cy
693     #2\xint_addp_cz
694     \W\XINT_addp_CD    {#5#4#3#2}{#1}%
695 }%
696 \def\XINT_addp_CD #1%
697 {%
698     \expandafter\XINT_addp_CC\the\numexpr 1+10#1\relax
699 }%
700 \def\XINT_addp_CC #1#2#3#4#5#6#7%
701 {%
702     \XINT_addp_AC_checkcarry #2{#7#6#5#4#3}%
703 }%
704 \def\xint_addp_cw
705     #1\xint_addp_cx
706     #2\xint_addp_cy
707     #3\xint_addp_cz
708     \W\XINT_addp_CD
709 {%
710     \expandafter\XINT_addp_CDw\the\numexpr \xint_c_i+10#1#2#3\relax
711 }%
712 \def\XINT_addp_CDw #1#2#3#4#5#6%
713 {%
714     \xint_gob_til_zeros_iv #2#3#4#5\XINT_addp_endDw_zeros
715     0000\XINT_addp_endDw #2#3#4#5%
716 }%
717 \def\XINT_addp_endDw_zeros 0000\XINT_addp_endDw 0000#1\X\Y\Z{ #1}%
718 \def\XINT_addp_endDw #1#2#3#4#5\X\Y\Z{ #5#4#3#2#1}%
719 \def\xint_addp_cx
720     #1\xint_addp_cy
721     #2\xint_addp_cz
722     \W\XINT_addp_CD
723 {%
724     \expandafter\XINT_addp_CDx\the\numexpr \xint_c_i+100#1#2\relax
725 }%
726 \def\XINT_addp_CDx #1#2#3#4#5#6%
727 {%
728     \xint_gob_til_zeros_iv #2#3#4#5\XINT_addp_endDx_zeros
729     0000\XINT_addp_endDx #2#3#4#5%
730 }%
731 \def\XINT_addp_endDx_zeros 0000\XINT_addp_endDx 0000#1\Y\Z{ #1}%
732 \def\XINT_addp_endDx #1#2#3#4#5\Y\Z{ #5#4#3#2#1}%
733 \def\xint_addp_cy #1\xint_addp_cz\W\XINT_addp_CD
734 {%

```

34 Package *xint* implementation

```

735 \expandafter\XINT_addp_CDy\the\numexpr \xint_c_i+1000#1\relax
736 }%
737 \def\XINT_addp_CDy #1#2#3#4#5#6%
738 {%
739 \xint_gob_til_zeros_iv #2#3#4#5\XINT_addp_endDy_zeros
740 0000\XINT_addp_endDy #2#3#4#5%
741 }%
742 \def\XINT_addp_endDy_zeros 0000\XINT_addp_endDy 0000#1\Z{ #1}%
743 \def\XINT_addp_endDy #1#2#3#4#5\Z{ #5#4#3#2#1}%
744 \def\xint_addp_cz\W\XINT_addp_CD #1#2{ #21000}%
745 \def\XINT_addp_F #1#2#3#4#5%
746 {%
747 \xint_gob_til_W
748 #5\xint_addp_Gw
749 #4\xint_addp_Gx
750 #3\xint_addp_Gy
751 #2\xint_addp_Gz
752 \W\XINT_addp_G {#2#3#4#5}{#1}%
753 }%
754 \def\XINT_addp_G #1#2%
755 {%
756 \XINT_addp_F {#2#1}%
757 }%
758 \def\xint_addp_Gw
759 #1\xint_addp_Gx
760 #2\xint_addp_Gy
761 #3\xint_addp_Gz
762 \W\XINT_addp_G #4%
763 {%
764 \xint_gob_til_zeros_iv #3#2#10\XINT_addp_endGw_zeros
765 0000\XINT_addp_endGw #3#2#10%
766 }%
767 \def\XINT_addp_endGw_zeros 0000\XINT_addp_endGw 0000#1\X\Y\Z{ #1}%
768 \def\XINT_addp_endGw #1#2#3#4#5\X\Y\Z{ #5#1#2#3#4}%
769 \def\xint_addp_Gx
770 #1\xint_addp_Gy
771 #2\xint_addp_Gz
772 \W\XINT_addp_G #3%
773 {%
774 \xint_gob_til_zeros_iv #2#100\XINT_addp_endGx_zeros
775 0000\XINT_addp_endGx #2#100%
776 }%
777 \def\XINT_addp_endGx_zeros 0000\XINT_addp_endGx 0000#1\Y\Z{ #1}%
778 \def\XINT_addp_endGx #1#2#3#4#5\Y\Z{ #5#1#2#3#4}%
779 \def\xint_addp_Gy
780 #1\xint_addp_Gz
781 \W\XINT_addp_G #2%
782 {%
783 \xint_gob_til_zeros_iv #1000\XINT_addp_endGy_zeros

```


34 Package *xint* implementation

```

784          0000\XINT_addp_endGy #1000%
785 }%
786 \def\XINT_addp_endGy_zeros 0000\XINT_addp_endGy 0000#1\Z{ #1}%
787 \def\XINT_addp_endGy #1#2#3#4#5\Z{ #5#1#2#3#4}%
788 \def\xint_addp_Gz\W\XINT_addp_G #1#2{ #2}%

```

34.25 \xintAdd

Release 1.09a has \xintnum added into \xintiAdd.

```

789 \def\xintiiAdd {\romannumeral0\xintiiadd }%
790 \def\xintiiadd #1%
791 {%
792   \expandafter\xint_iiadd\expandafter{\romannumeral-‘0#1}%
793 }%
794 \def\xint_iiadd #1#2%
795 {%
796   \expandafter\XINT_add_fork \romannumeral-‘0#2\Z #1\Z
797 }%
798 \def\xintiAdd {\romannumeral0\xintiadd }%
799 \def\xintiadd #1%
800 {%
801   \expandafter\xint_add\expandafter{\romannumeral0\xintnum{#1}}%
802 }%
803 \def\xint_add #1#2%
804 {%
805   \expandafter\XINT_add_fork \romannumeral0\xintnum{#2}\Z #1\Z
806 }%
807 \let\xintAdd\xintiAdd \let\xintadd\xintiadd
808 \def\XINT_Add #1#2{\romannumeral0\XINT_add_fork #2\Z #1\Z }%
809 \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]

```

810 \def\XINT_add_fork #1#2\Z #3#4\Z
811 {%
812   \xint_UDzerofork
813     #1\dummy \XINT_add_secondiszero
814     #3\dummy \XINT_add_firstiszero
815     0\dummy
816     {\xint_UDsignsfork
817       #1#3\dummy \XINT_add_minusminus      % #1 = #3 = -
818       #1-\dummy \XINT_add_minusplus       % #1 = -
819       #3-\dummy \XINT_add_plusminus      % #3 = -
820       --\dummy \XINT_add_plusplus
821     \krof }%
822 \krof
823 {#2}{#4}#1#3%

```

```

824 }%
825 \def\XINT_add_secondiszero #1#2#3#4{ #4#2}%
826 \def\XINT_add_firstiszero #1#2#3#4{ #3#1}%

#1 vient du *deuxième* et #2 vient du *premier*

827 \def\XINT_add_minusminus #1#2#3#4%
828 {%
829   \expandafter\xint_minus_andstop%
830   \romannumeral0\XINT_add_pre {#2}{#1}%
831 }%
832 \def\XINT_add_minusplus #1#2#3#4%
833 {%
834   \XINT_sub_pre {#4#2}{#1}%
835 }%
836 \def\XINT_add_plusminus #1#2#3#4%
837 {%
838   \XINT_sub_pre {#3#1}{#2}%
839 }%
840 \def\XINT_add_plusplus #1#2#3#4%
841 {%
842   \XINT_add_pre {#4#2}{#3#1}%
843 }%
844 \def\XINT_add_pre #1%
845 {%
846   \expandafter\XINT_add_pre_b\expandafter
847   {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z }%
848 }%
849 \def\XINT_add_pre_b #1#2%
850 {%
851   \expandafter\XINT_add_A
852   \expandafter0\expandafter{\expandafter}%
853   \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
854   \W\X\Y\Z #1\W\X\Y\Z
855 }%

```

34.26 \xintSub

Release 1.09a has \xintnum added into \xintiSub.

```

856 \def\xintiiSub {\romannumeral0\xintiisub }%
857 \def\xintiisub #1%
858 {%
859   \expandafter\xint_iisub\expandafter{\romannumeral-‘0#1}%
860 }%
861 \def\xint_iisub #1#2%
862 {%
863   \expandafter\XINT_sub_fork \romannumeral-‘0#2\Z #1\Z
864 }%

```

```

865 \def\xintiSub {\romannumeral0\xintisub }%
866 \def\xintisub #1%
867 {%
868   \expandafter\xint_sub\expandafter{\romannumeral0\xintnum{#1}}%
869 }%
870 \def\xint_sub #1#2%
871 {%
872   \expandafter\XINT_sub_fork \romannumeral0\xintnum{#2}\Z #1\Z
873 }%
874 \def\XINT_Sub #1#2{\romannumeral0\XINT_sub_fork #2\Z #1\Z }%
875 \def\XINT_sub #1#2{\XINT_sub_fork #2\Z #1\Z }%
876 \let\xintSub\xintiSub \let\xintsub\xintisub

SOUSTRACTION #3#4-#1#2: #3#4 vient du *premier* #1#2 vient du *second*

877 \def\XINT_sub_fork #1#2\Z #3#4\Z
878 {%
879   \xint_UDsignsfork
880     #1#3\dummy \XINT_sub_minusminus
881     #1-\dummy \XINT_sub_minusplus % attention, #3=0 possible
882     #3-\dummy \XINT_sub_plusminus % attention, #1=0 possible
883     --\dummy {\xint_UDzerofork
884       #1\dummy \XINT_sub_secondiszero
885       #3\dummy \XINT_sub_firstiszero
886       0\dummy \XINT_sub_plusplus
887       \krof }%
888   \krof
889   {#2}{#4}#1#3%
890 }%
891 \def\XINT_sub_secondiszero #1#2#3#4{ #4#2}%
892 \def\XINT_sub_firstiszero #1#2#3#4{ -#3#1}%
893 \def\XINT_sub_plusplus #1#2#3#4%
894 {%
895   \XINT_sub_pre {#4#2}{#3#1}%
896 }%
897 \def\XINT_sub_minusminus #1#2#3#4%
898 {%
899   \XINT_sub_pre {#1}{#2}%
900 }%
901 \def\XINT_sub_minusplus #1#2#3#4%
902 {%
903   \xint_gob_til_zero #4\xint_sub_mp0\XINT_add_pre {#4#2}{#1}%
904 }%
905 \def\xint_sub_mp0\XINT_add_pre #1#2{ #2}%
906 \def\XINT_sub_plusminus #1#2#3#4%
907 {%
908   \xint_gob_til_zero #3\xint_sub_pm0\expandafter\xint_minus_andstop%
909   \romannumeral0\XINT_add_pre {#2}{#3#1}%
910 }%
911 \def\xint_sub_pm #1\XINT_add_pre #2#3{ -#2}%

```

```

912 \def\XINT_sub_pre #1%
913 {%
914   \expandafter\XINT_sub_pre_b\expandafter
915   {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z }%
916 }%
917 \def\XINT_sub_pre_b #1#2%
918 {%
919   \expandafter\XINT_sub_A
920   \expandafter1\expandafter{\expandafter}%
921   \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
922   \W\X\Y\Z #1 \W\X\Y\Z
923 }%

\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.

924 \def\XINT_sub_A #1#2#3\W\X\Y\Z #4#5#6#7%
925 {%
926   \xint_gob_til_W
927   #4\xint_sub_az
928   \W\XINT_sub_B #1{#4#5#6#7}{#2}#3\W\X\Y\Z
929 }%
930 \def\XINT_sub_B #1#2#3#4#5#6#7%
931 {%
932   \xint_gob_til_W
933   #4\xint_sub_bz
934   \W\XINT_sub_onestep #1#2{#7#6#5#4}{#3}%
935 }%

d'abord la branche principale #6 = 4 chiffres de N1, plus significatif en *pre-
*mier*, #2#3#4#5 chiffres de N2, plus significatif en *dernier* On veut N2 - N1.

936 \def\XINT_sub_onestep #1#2#3#4#5#6%
937 {%
938   \expandafter\XINT_sub_backtoA\the\numexpr 11#5#4#3#2-#6+#1-\xint_c_i.%
939 }%

ON PRODUIT LE RÉSULTAT DANS LE BON ORDRE

940 \def\XINT_sub_backtoA #1#2#3.#4%
941 {%
942   \XINT_sub_A #2{#3#4}%
943 }%
944 \def\xint_sub_bz
945   \W\XINT_sub_onestep #1#2#3#4#5#6#7%
946 {%

```

```

947 \xint_UDzerofork
948   #1\dummy \XINT_sub_C % une retenue
949   0\dummy \XINT_sub_D % pas de retenue
950 \krof
951 {#7}#2#3#4#5%
952 }%
953 \def\XINT_sub_D #1#2\W\X\Y\Z
954 {%
955   \expandafter
956   \xint_cleanupzeros_andstop
957   \romannumeral0%
958   \XINT_rord_main {}#2%
959   \xint_relax
960   \xint_bye\xint_bye\xint_bye\xint_bye
961   \xint_bye\xint_bye\xint_bye\xint_bye
962   \xint_relax
963   #1%
964 }%
965 \def\XINT_sub_C #1#2#3#4#5%
966 {%
967   \xint_gob_til_W
968   #2\xint_sub_cz
969   \W\XINT_sub_AC_onestep {#5#4#3#2}{#1}%
970 }%
971 \def\XINT_sub_AC_onestep #1%
972 {%
973   \expandafter\XINT_sub_backtoC\the\numexpr 11#1-\xint_c_i.%
974 }%
975 \def\XINT_sub_backtoC #1#2#3.#4%
976 {%
977   \XINT_sub_AC_checkcarry #2{#3#4}% la retenue va \^etre examin\'ee
978 }%
979 \def\XINT_sub_AC_checkcarry #1%
980 {%
981   \xint_gob_til_one #1\xint_sub_AC_nocarry 1\XINT_sub_C
982 }%
983 \def\xint_sub_AC_nocarry 1\XINT_sub_C #1#2\W\X\Y\Z
984 {%
985   \expandafter
986   \XINT_cuz_loop
987   \romannumeral0%
988   \XINT_rord_main {}#2%
989   \xint_relax
990   \xint_bye\xint_bye\xint_bye\xint_bye
991   \xint_bye\xint_bye\xint_bye\xint_bye
992   \xint_relax
993   #1\W\W\W\W\W\W\W\Z
994 }%
995 \def\xint_sub_cz\W\XINT_sub_AC_onestep #1%

```

```

996 {%
997   \XINT_cuz
998 }%
999 \def\xint_sub_az\W\XINT_sub_B #1#2#3#4#5#6#7%
1000 {%
1001   \xint_gob_til_W
1002   #4\xint_sub_ez
1003   \W\XINT_sub_Eenter #1{#3}#4#5#6#7%
1004 }%

  le premier nombre continue, le résultat sera < 0.

1005 \def\XINT_sub_Eenter #1#2%
1006 {%
1007   \expandafter
1008   \XINT_sub_E\expandafter1\expandafter{\expandafter}%
1009   \romannumeral0%
1010   \XINT_rord_main {}#2%
1011   \xint_relax
1012   \xint_bye\xint_bye\xint_bye\xint_bye
1013   \xint_bye\xint_bye\xint_bye\xint_bye
1014   \xint_relax
1015   \W\X\Y\Z #1%
1016 }%
1017 \def\XINT_sub_E #1#2#3#4#5#6%
1018 {%
1019   \xint_gob_til_W #3\xint_sub_F\W
1020   \XINT_sub_Eonestep #1{#6#5#4#3}{#2}%
1021 }%
1022 \def\XINT_sub_Eonestep #1#2%
1023 {%
1024   \expandafter\XINT_sub_backtoE\the\numexpr 109999-#2+#1.%
1025 }%
1026 \def\XINT_sub_backtoE #1#2#3.#4%
1027 {%
1028   \XINT_sub_E #2{#3#4}%
1029 }%
1030 \def\xint_sub_F\W\XINT_sub_Eonestep #1#2#3#4%
1031 {%
1032   \xint_UDonezerofork
1033   #4#1\dummy {\XINT_sub_Fdec 0}% soustraire 1. Et faire signe -
1034   #1#4\dummy {\XINT_sub_Finc 1}% additionner 1. Et faire signe -
1035   10\dummy \XINT_sub_DD % terminer. Mais avec signe -
1036   \krof
1037   {#3}%
1038 }%
1039 \def\XINT_sub_DD {\expandafter\xint_minus_andstop\romannumeral0\XINT_sub_D }%
1040 \def\XINT_sub_Fdec #1#2#3#4#5#6%
1041 {%
1042   \xint_gob_til_W #3\xint_sub_Fdec_finish\W

```

34 Package *xint* implementation

```

1043 \XINT_sub_Fdec_onestep #1{#6#5#4#3}{#2}%
1044 }%
1045 \def\XINT_sub_Fdec_onestep #1#2%
1046 {%
1047 \expandafter\XINT_sub_backtoFdec\the\numexpr 11#2+#1-\xint_c_i.%
1048 }%
1049 \def\XINT_sub_backtoFdec #1#2#3.#4%
1050 {%
1051 \XINT_sub_Fdec #2{#3#4}%
1052 }%
1053 \def\xint_sub_Fdec_finish\W\XINT_sub_Fdec_onestep #1#2%
1054 {%
1055 \expandafter\xint_minus_andstop\romannumeral0\XINT_cuz
1056 }%
1057 \def\XINT_sub_Finc #1#2#3#4#5#6%
1058 {%
1059 \xint_gob_til_W #3\xint_sub_Finc_finish\W
1060 \XINT_sub_Finc_onestep #1{#6#5#4#3}{#2}%
1061 }%
1062 \def\XINT_sub_Finc_onestep #1#2%
1063 {%
1064 \expandafter\XINT_sub_backtoFinc\the\numexpr 10#2+#1.%
1065 }%
1066 \def\XINT_sub_backtoFinc #1#2#3.#4%
1067 {%
1068 \XINT_sub_Finc #2{#3#4}%
1069 }%
1070 \def\xint_sub_Finc_finish\W\XINT_sub_Finc_onestep #1#2#3%
1071 {%
1072 \xint_UDzerofork
1073 #1\dummy {\expandafter\xint_minus_andstop\xint_cleanupzeros_nospace}%
1074 0\dummy { -1}%
1075 \krof
1076 #3%
1077 }%
1078 \def\xint_sub_ez\W\XINT_sub_Eenter #1%
1079 {%
1080 \xint_UDzerofork
1081 #1\dummy \XINT_sub_K % il y a une retenue
1082 0\dummy \XINT_sub_L % pas de retenue
1083 \krof
1084 }%
1085 \def\XINT_sub_L #1\W\X\Y\Z {\XINT_cuz_loop #1\W\W\W\W\W\W\W\Z }%
1086 \def\XINT_sub_K #1%
1087 {%
1088 \expandafter
1089 \XINT_sub_KK\expandafter1\expandafter{\expandafter}%
1090 \romannumeral0%
1091 \XINT_rord_main {}#1%

```

```

1092      \xint_relax
1093      \xint_bye\xint_bye\xint_bye\xint_bye
1094      \xint_bye\xint_bye\xint_bye\xint_bye
1095      \xint_relax
1096 }%
1097 \def\xINT_sub_KK #1#2#3#4#5#6%
1098 {%
1099     \xint_gob_til_W #3\xint_sub_KK_finish\W
1100     \XINT_sub_KK_onestep #1{#6#5#4#3}{#2}%
1101 }%
1102 \def\xINT_sub_KK_onestep #1#2%
1103 {%
1104     \expandafter\xINT_sub_backtoKK\the\numexpr 109999-#2+#1.%
1105 }%
1106 \def\xINT_sub_backtoKK #1#2#3.#4%
1107 {%
1108     \XINT_sub_KK #2{#3#4}%
1109 }%
1110 \def\xint_sub_KK_finish\W\xINT_sub_KK_onestep #1#2#3%
1111 {%
1112     \expandafter\xint_minus_andstop
1113     \romannumeral0\xINT_cuz_loop #3\W\W\W\W\W\W\W\W\Z
1114 }%

```

34.27 \xintCmp

Release 1.09a has \xintnum inserted into \xintCmp. Unnecessary \xintiCmp suppressed in 1.09f.

```

1115 \def\xintCmp {\romannumeral0\xintcmp }%
1116 \def\xintcmp #1%
1117 {%
1118     \expandafter\xint_cmp\expandafter{\romannumeral0\xintnum{#1}}%
1119 }%
1120 \def\xint_cmp #1#2%
1121 {%
1122     \expandafter\xINT_cmp_fork \romannumeral0\xintnum{#2}\Z #1\Z
1123 }%
1124 \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*

1125 \def\xINT_cmp_fork #1#2\Z #3#4\Z
1126 {%
1127     \xint_UDsignsfork
1128     #1#3\dummy \XINT_cmp_minusminus
1129     #1-\dummy \XINT_cmp_minusplus

```


34 Package *xint* implementation

```

1130      #3-\dummy \XINT_cmp_plusminus
1131      --\dummy {\xint_UDzerosfork
1132      #1#3\dummy \XINT_cmp_zerozero
1133      #10\dummy \XINT_cmp_zeroplus
1134      #30\dummy \XINT_cmp_pluszero
1135      00\dummy \XINT_cmp_plusplus
1136      \krof }%
1137      \krof
1138      {#2}{#4}#1#3%
1139 }%
1140 \def\XINT_cmp_minusplus #1#2#3#4{ 1}%
1141 \def\XINT_cmp_plusminus #1#2#3#4{ -1}%
1142 \def\XINT_cmp_zerozero #1#2#3#4{ 0}%
1143 \def\XINT_cmp_zeroplus #1#2#3#4{ 1}%
1144 \def\XINT_cmp_pluszero #1#2#3#4{ -1}%
1145 \def\XINT_cmp_plusplus #1#2#3#4%
1146 {%
1147      \XINT_cmp_pre {#4#2}{#3#1}%
1148 }%
1149 \def\XINT_cmp_minusminus #1#2#3#4%
1150 {%
1151      \XINT_cmp_pre {#1}{#2}%
1152 }%
1153 \def\XINT_cmp_pre #1%
1154 {%
1155      \expandafter\XINT_cmp_pre_b\expandafter
1156      {\romannumeral0\XINT_RQ {}}#1\R\R\R\R\R\R\R\R\Z }%
1157 }%
1158 \def\XINT_cmp_pre_b #1#2%
1159 {%
1160      \expandafter\XINT_cmp_A
1161      \expandafter1\expandafter{\expandafter}%
1162      \romannumeral0\XINT_RQ {}}#2\R\R\R\R\R\R\R\R\Z
1163      \W\X\Y\Z #1\W\X\Y\Z
1164 }%

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

1165 \def\XINT_cmp_A #1#2#3\W\X\Y\Z #4#5#6#7%
1166 {%
1167      \xint_gob_til_W #4\xint_cmp_az\W
1168      \XINT_cmp_B #1{#4#5#6#7}{#2}#3\W\X\Y\Z
1169 }%
1170 \def\XINT_cmp_B #1#2#3#4#5#6#7%
1171 {%

```

34 Package *xint* implementation

```

1172 \xint_gob_til_W#4\xint_cmp_bz\W
1173 \XINT_cmp_onestep #1#2{#7#6#5#4}{#3}%
1174 }%
1175 \def\XINT_cmp_onestep #1#2#3#4#5#6%
1176 {%
1177 \expandafter\XINT_cmp_backtoA\the\numexpr 11#5#4#3#2-#6+#1-\xint_c_i.%
1178 }%
1179 \def\XINT_cmp_backtoA #1#2#3.#4%
1180 {%
1181 \XINT_cmp_A #2{#3#4}%
1182 }%
1183 \def\xint_cmp_bz\W\XINT_cmp_onestep #1\Z { 1}%
1184 \def\xint_cmp_az\W\XINT_cmp_B #1#2#3#4#5#6#7%
1185 {%
1186 \xint_gob_til_W #4\xint_cmp_ez\W
1187 \XINT_cmp_Eenter #1{#3}#4#5#6#7%
1188 }%
1189 \def\XINT_cmp_Eenter #1\Z { -1}%
1190 \def\xint_cmp_ez\W\XINT_cmp_Eenter #1%
1191 {%
1192 \xint_UDzerofork
1193 #1\dummy \XINT_cmp_K % il y a une retenue
1194 0\dummy \XINT_cmp_L % pas de retenue
1195 \krof
1196 }%
1197 \def\XINT_cmp_K #1\Z { -1}%
1198 \def\XINT_cmp_L #1{\XINT_OneIfPositive_main #1}%
1199 \def\XINT_OneIfPositive #1%
1200 {%
1201 \XINT_OneIfPositive_main #1\W\X\Y\Z%
1202 }%
1203 \def\XINT_OneIfPositive_main #1#2#3#4%
1204 {%
1205 \xint_gob_til_Z #4\xint_OneIfPositive_terminated\Z
1206 \XINT_OneIfPositive_onestep #1#2#3#4%
1207 }%
1208 \def\xint_OneIfPositive_terminated\Z\XINT_OneIfPositive_onestep\W\X\Y\Z { 0}%
1209 \def\XINT_OneIfPositive_onestep #1#2#3#4%
1210 {%
1211 \expandafter\XINT_OneIfPositive_check\the\numexpr #1#2#3#4\relax
1212 }%
1213 \def\XINT_OneIfPositive_check #1%
1214 {%
1215 \xint_gob_til_zero #1\xint_OneIfPositive_backtomain 0%
1216 \XINT_OneIfPositive_finish #1%
1217 }%
1218 \def\XINT_OneIfPositive_finish #1\W\X\Y\Z{ 1}%
1219 \def\xint_OneIfPositive_backtomain 0\XINT_OneIfPositive_finish 0%
1220 {\XINT_OneIfPositive_main }%

```

34.28 \xintEq, \xintGt, \xintLt

1.09a.

```

1221 \def\xintEq {\romannumeral0\xinteq }%
1222 \def\xinteq #1#2{\xintifeq{#1}{#2}{1}{0}}%
1223 \def\xintGt {\romannumeral0\xintgt }%
1224 \def\xintgt #1#2{\xintifgt{#1}{#2}{1}{0}}%
1225 \def\xintLt {\romannumeral0\xintlt }%
1226 \def\xintlt #1#2{\xintiflt{#1}{#2}{1}{0}}%

```

34.29 \xintIsZero, \xintIsNotZero

1.09a.

```

1227 \def\xintIsZero {\romannumeral0\xintiszero }%
1228 \def\xintiszero #1{\xintifsgn {#1}{0}{1}{0}}%
1229 \def\xintIsNotZero {\romannumeral0\xintisnotzero }%
1230 \def\xintisnotzero #1{\xintifsgn {#1}{1}{0}{1}}%

```

34.30 \xintIsTrue, \xintNot

1.09c

```

1231 \let\xintIsTrue\xintIsNotZero
1232 \let\xintNot\xintIsZero

```

34.31 \xintIsTrue:csv

1.09c. For use by \xinttheboolexpr.

```

1233 \def\xintIsTrue:csv #1{\expandafter\XINT_istrue:_a\romannumeral-‘0#1,,^}%
1234 \def\XINT_istrue:_a {\XINT_istrue:_b {}}%
1235 \def\XINT_istrue:_b #1#2,%
1236         {\expandafter\XINT_istrue:_c\romannumeral-‘0#2,{#1}}%
1237 \def\XINT_istrue:_c #1{\if #1,\expandafter\XINT_istrue:_f
1238         \else\expandafter\XINT_istrue:_d\fi #1}%
1239 \def\XINT_istrue:_d #1,%
1240         {\expandafter\XINT_istrue:_e\romannumeral0\xintisnotzero {#1},}%
1241 \def\XINT_istrue:_e #1,#2{\XINT_istrue:_b {#2,#1}}%
1242 \def\XINT_istrue:_f ,#1#2^{\xint_gobble_i #1}%

```

34.32 \xintAND, \xintOR, \xintXOR

1.09a.

```

1243 \def\xintAND {\romannumeral0\xintand }%
1244 \def\xintand #1#2{\xintifzero {#1}{0}{\xintifzero {#2}{0}{1}}}%

```

```

1245 \def\xintOR {\romannumeral0\xintor }%
1246 \def\xintor #1#2{\xintifzero {#1}{\xintifzero {#2}{0}{1}}{1}}%
1247 \def\xintXOR {\romannumeral0\xintxor }%
1248 \def\xintxor #1#2{\ifcase \numexpr\xintIsZero{#1}+\xintIsZero{#2}\relax
1249         \xint_afterfi{ 0}%
1250         \or\xint_afterfi{ 1}%
1251         \else\xint_afterfi { 0}%
1252         \fi }%

```

34.33 \xintANDof

New with 1.09a. \xintANDof works with an empty list.

```

1253 \def\xintANDof {\romannumeral0\xintandof }%
1254 \def\xintandof #1{\expandafter\XINT_andof_a\romannumeral-‘0#1\relax }%
1255 \def\XINT_andof_a #1{\expandafter\XINT_andof_b\romannumeral-‘0#1\Z }%
1256 \def\XINT_andof_b #1%
1257     {\xint_gob_til_relax #1\XINT_andof_e\relax\XINT_andof_c #1}%
1258 \def\XINT_andof_c #1\Z
1259     {\xintifZero{#1}{\XINT_andof_no}{\XINT_andof_a}}%
1260 \def\XINT_andof_no #1\relax { 0}%
1261 \def\XINT_andof_e #1\Z { 1}%

```

34.34 \xintANDof:csv

1.09a. For use by \xintexpr.

```

1262 \def\xintANDof:csv #1{\expandafter\XINT_andof:_a\romannumeral-‘0#1,,^}%
1263 \def\XINT_andof:_a {\expandafter\XINT_andof:_b\romannumeral-‘0}%
1264 \def\XINT_andof:_b #1{\if #1,\expandafter\XINT_andof:_e
1265     \else\expandafter\XINT_andof:_c\fi #1}%
1266 \def\XINT_andof:_c #1,{\xintifZero{#1}{\XINT_andof:_no}{\XINT_andof:_a}}%
1267 \def\XINT_andof:_no #1^{0}%
1268 \def\XINT_andof:_e #1^{1}%

```

34.35 \xintORof

New with 1.09a. Works also with an empty list.

```

1269 \def\xintORof {\romannumeral0\xintorof }%
1270 \def\xintorof #1{\expandafter\XINT_orof_a\romannumeral-‘0#1\relax }%
1271 \def\XINT_orof_a #1{\expandafter\XINT_orof_b\romannumeral-‘0#1\Z }%
1272 \def\XINT_orof_b #1%
1273     {\xint_gob_til_relax #1\XINT_orof_e\relax\XINT_orof_c #1}%
1274 \def\XINT_orof_c #1\Z
1275     {\xintifZero{#1}{\XINT_orof_a}{\XINT_orof_yes}}%
1276 \def\XINT_orof_yes #1\relax { 1}%
1277 \def\XINT_orof_e #1\Z { 0}%

```

34.36 \xintORof:csv

1.09a. For use by \xintexpr.

```

1278 \def\xintORof:csv #1{\expandafter\xint_orof:_a\romannumeral-'0#1,,^}%
1279 \def\xint_orof:_a {\expandafter\xint_orof:_b\romannumeral-'0}%
1280 \def\xint_orof:_b #1{\if #1,\expandafter\xint_orof:_e
1281         \else\expandafter\xint_orof:_c\fi #1}%
1282 \def\xint_orof:_c #1,{\xintifZero{#1}{\XINT_orof:_a}{\XINT_orof:_yes}}%
1283 \def\xint_orof:_yes #1^{1}%
1284 \def\xint_orof:_e #1^{0}%

```

34.37 \xintXORof

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

```

1285 \def\xintXORof      {\romannumeral0\xintxorof }%
1286 \def\xintxorof      #1{\expandafter\xint_xorof_a\expandafter
1287         0\romannumeral-'0#1\relax }%
1288 \def\xint_xorof_a #1#2{\expandafter\xint_xorof_b\romannumeral-'0#2\Z #1}%
1289 \def\xint_xorof_b #1%
1290         {\xint_gob_til_relax #1\xint_xorof_e\relax\xint_xorof_c #1}%
1291 \def\xint_xorof_c #1\Z #2%
1292         {\xintifZero {#1}{\XINT_xorof_a #2}{\ifcase #2
1293         \xint_afterfi{\XINT_xorof_a 1}%
1294         \else
1295         \xint_afterfi{\XINT_xorof_a 0}%
1296         \fi }%
1297         }%
1298 \def\xint_xorof_e #1\Z #2{ #2}%

```

34.38 \xintXORof:csv

1.09a. For use by \xintexpr.

```

1299 \def\xintXORof:csv #1{\expandafter\xint_xorof:_a\expandafter
1300         0\romannumeral-'0#1,,^}%
1301 \def\xint_xorof:_a #1#2,{\expandafter\xint_xorof:_b\romannumeral-'0#2,#1}%
1302 \def\xint_xorof:_b #1{\if #1,\expandafter\xint_xorof:_e
1303         \else\expandafter\xint_xorof:_c\fi #1}%
1304 \def\xint_xorof:_c #1,#2%
1305         {\xintifZero {#1}{\XINT_xorof:_a #2}{\ifcase #2
1306         \xint_afterfi{\XINT_xorof:_a 1}%
1307         \else
1308         \xint_afterfi{\XINT_xorof:_a 0}%
1309         \fi }%
1310         }%
1311 \def\xint_xorof:_e ,#1#2^{#1}% allows empty list

```

34.39 \xintGeq

Release 1.09a has \xintnum added into \xintGeq. Unused and useless \xintiGeq removed in 1.09e. PLUS GRAND OU ÉGAL attention compare les ****valeurs absolues****

```

1312 \def\xintGeq {\romannumeral0\xintgeq }%
1313 \def\xintgeq #1%
1314 {%
1315   \expandafter\xint_geq\expandafter {\romannumeral0\xintnum{#1}}%
1316 }%
1317 \def\xint_geq #1#2%
1318 {%
1319   \expandafter\XINT_geq_fork \romannumeral0\xintnum{#2}\Z #1\Z
1320 }%
1321 \def\XINT_Geq #1#2{\romannumeral0\XINT_geq_fork #2\Z #1\Z }%

PLUS GRAND OU ÉGAL ATTENTION, TESTE les VALEURS ABSOLUES

1322 \def\XINT_geq_fork #1#2\Z #3#4\Z
1323 {%
1324   \xint_UDzerofork
1325   #1\dummy \XINT_geq_secondiszero % |#1#2|=0
1326   #3\dummy \XINT_geq_firstiszero % |#1#2|>0
1327   0\dummy {\xint_UDsignsfork
1328             #1#3\dummy \XINT_geq_minusminus
1329             #1-\dummy \XINT_geq_minusplus
1330             #3-\dummy \XINT_geq_plusminus
1331             --\dummy \XINT_geq_plusplus
1332             \krof }%
1333   \krof
1334   {#2}{#4}#1#3%
1335 }%
1336 \def\XINT_geq_secondiszero #1#2#3#4{ 1}%
1337 \def\XINT_geq_firstiszero #1#2#3#4{ 0}%
1338 \def\XINT_geq_plusplus #1#2#3#4{\XINT_geq_pre {#4#2}{#3#1}}%
1339 \def\XINT_geq_minusminus #1#2#3#4{\XINT_geq_pre {#2}{#1}}%
1340 \def\XINT_geq_minusplus #1#2#3#4{\XINT_geq_pre {#4#2}{#1}}%
1341 \def\XINT_geq_plusminus #1#2#3#4{\XINT_geq_pre {#2}{#3#1}}%
1342 \def\XINT_geq_pre #1%
1343 {%
1344   \expandafter\XINT_geq_pre_b\expandafter
1345   {\romannumeral0\XINT_RQ {}}#1\R\R\R\R\R\R\R\R\Z }%
1346 }%
1347 \def\XINT_geq_pre_b #1#2%
1348 {%
1349   \expandafter\XINT_geq_A
1350   \expandafter1\expandafter{\expandafter}%
1351   \romannumeral0\XINT_RQ {}}#2\R\R\R\R\R\R\R\R\Z
1352   \W\X\Y\Z #1 \W\X\Y\Z
1353 }%

```

```

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

1354 \def\XINT_geq_A #1#2#3\W\X\Y\Z #4#5#6#7%
1355 {%
1356   \xint_gob_til_W #4\xint_geq_az\W
1357   \XINT_geq_B #1{#4#5#6#7}{#2}#3\W\X\Y\Z
1358 }%
1359 \def\XINT_geq_B #1#2#3#4#5#6#7%
1360 {%
1361   \xint_gob_til_W #4\xint_geq_bz\W
1362   \XINT_geq_onestep #1#2{#7#6#5#4}{#3}%
1363 }%
1364 \def\XINT_geq_onestep #1#2#3#4#5#6%
1365 {%
1366   \expandafter\XINT_geq_backtoA\the\numexpr 11#5#4#3#2-#6+#1-\xint_c_i.%
1367 }%
1368 \def\XINT_geq_backtoA #1#2#3.#4%
1369 {%
1370   \XINT_geq_A #2{#3#4}%
1371 }%
1372 \def\xint_geq_bz\W\XINT_geq_onestep #1\W\X\Y\Z { 1}%
1373 \def\xint_geq_az\W\XINT_geq_B #1#2#3#4#5#6#7%
1374 {%
1375   \xint_gob_til_W #4\xint_geq_ez\W
1376   \XINT_geq_Enter #1%
1377 }%
1378 \def\XINT_geq_Enter #1\W\X\Y\Z { 0}%
1379 \def\xint_geq_ez\W\XINT_geq_Enter #1%
1380 {%
1381   \xint_UDzerofork
1382   #1\dummy { 0}           %   il y a une retenue
1383   0\dummy { 1}           %   pas de retenue
1384   \krof
1385 }%

```

34.40 \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`.

```

1386 \def\xintiMax {\romannumeral0\xintimax }%

```

```

1387 \def\xintimax #1%
1388 {%
1389   \expandafter\xint_max\expandafter {\romannumeral0\xintnum{#1}}%
1390 }%
1391 \let\xintMax\xintiMax \let\xintmax\xintimax
1392 \def\xint_max #1#2%
1393 {%
1394   \expandafter\XINT_max_pre\expandafter {\romannumeral0\xintnum{#2}}{#1}%
1395 }%
1396 \def\XINT_max_pre #1#2{\XINT_max_fork #1\Z #2\Z {#2}{#1}}%
1397 \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*

1398 \def\XINT_max_fork #1#2\Z #3#4\Z
1399 {%
1400   \xint_UDsignsfork
1401     #1#3\dummy \XINT_max_minusminus % A < 0, B < 0
1402     #1-\dummy \XINT_max_minusplus % B < 0, A >= 0
1403     #3-\dummy \XINT_max_plusminus % A < 0, B >= 0
1404     --\dummy {\xint_UDzerosfork
1405       #1#3\dummy \XINT_max_zerozero % A = B = 0
1406       #10\dummy \XINT_max_zeroplus % B = 0, A > 0
1407       #30\dummy \XINT_max_pluszero % A = 0, B > 0
1408       00\dummy \XINT_max_plusplus % A, B > 0
1409     \krof }%
1410   \krof
1411   {#2}{#4}#1#3%
1412 }%

A = #4#2, B = #3#1

1413 \def\XINT_max_zerozero #1#2#3#4{\xint_firstoftwo_andstop }%
1414 \def\XINT_max_zeroplus #1#2#3#4{\xint_firstoftwo_andstop }%
1415 \def\XINT_max_pluszero #1#2#3#4{\xint_secondoftwo_andstop }%
1416 \def\XINT_max_minusplus #1#2#3#4{\xint_firstoftwo_andstop }%
1417 \def\XINT_max_plusminus #1#2#3#4{\xint_secondoftwo_andstop }%
1418 \def\XINT_max_plusplus #1#2#3#4%
1419 {%
1420   \ifodd\XINT_Geq {#4#2}{#3#1}
1421     \expandafter\xint_firstoftwo_andstop
1422   \else
1423     \expandafter\xint_secondoftwo_andstop
1424   \fi
1425 }%

#3=-, #4=-, #1 = |B| = -B, #2 = |A| = -A

1426 \def\XINT_max_minusminus #1#2#3#4%
1427 {%

```



```

1428 \ifodd\XINT_Geq {#1}{#2}
1429 \expandafter\xint_firstoftwo_andstop
1430 \else
1431 \expandafter\xint_secondoftwo_andstop
1432 \fi
1433 }%

```

34.41 \xintMaxof

New with 1.09a.

```

1434 \def\xintiMaxof {\romannumeral0\xintimaxof }%
1435 \def\xintimaxof #1{\expandafter\XINT_imaxof_a\romannumeral-‘0#1\relax }%
1436 \def\XINT_imaxof_a #1{\expandafter\XINT_imaxof_b\romannumeral0\xintnum{#1}\Z }%
1437 \def\XINT_imaxof_b #1\Z #2%
1438 {\expandafter\XINT_imaxof_c\romannumeral-‘0#2\Z {#1}\Z}%
1439 \def\XINT_imaxof_c #1%
1440 {\xint_gob_til_relax #1\XINT_imaxof_e\relax\XINT_imaxof_d #1}%
1441 \def\XINT_imaxof_d #1\Z
1442 {\expandafter\XINT_imaxof_b\romannumeral0\xintimax {#1}}%
1443 \def\XINT_imaxof_e #1\Z #2\Z { #2}%
1444 \let\xintMaxof\xintiMaxof \let\xintmaxof\xintimaxof

```

34.42 \xintMin

\xintnum added New with 1.09a.

```

1445 \def\xintiMin {\romannumeral0\xintimin }%
1446 \def\xintimin #1%
1447 {%
1448 \expandafter\xint_min\expandafter {\romannumeral0\xintnum{#1}}%
1449 }%
1450 \let\xintMin\xintiMin \let\xintmin\xintimin
1451 \def\xint_min #1#2%
1452 {%
1453 \expandafter\XINT_min_pre\expandafter {\romannumeral0\xintnum{#2}}{#1}%
1454 }%
1455 \def\XINT_min_pre #1#2{\XINT_min_fork #1\Z #2\Z {#2}{#1}}%
1456 \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*

1457 \def\XINT_min_fork #1#2\Z #3#4\Z
1458 {%
1459 \xint_UDsignsfork
1460 #1#3\dummy \XINT_min_minusminus % A < 0, B < 0
1461 #1-\dummy \XINT_min_minusplus % B < 0, A >= 0
1462 #3-\dummy \XINT_min_plusminus % A < 0, B >= 0
1463 --\dummy {\xint_UDzerosfork

```

34 Package *xint* implementation

```

1464          #1#3\dummy \XINT_min_zerozero % A = B = 0
1465          #10\dummy \XINT_min_zeroplus % B = 0, A > 0
1466          #30\dummy \XINT_min_pluszero % A = 0, B > 0
1467          00\dummy \XINT_min_plusplus % A, B > 0
1468          \krof }%
1469      \krof
1470      {#2}{#4}#1#3%
1471 }%

A = #4#2, B = #3#1

1472 \def\XINT_min_zerozero #1#2#3#4{\xint_firstoftwo_andstop }%
1473 \def\XINT_min_zeroplus #1#2#3#4{\xint_secondoftwo_andstop }%
1474 \def\XINT_min_pluszero #1#2#3#4{\xint_firstoftwo_andstop }%
1475 \def\XINT_min_minusplus #1#2#3#4{\xint_secondoftwo_andstop }%
1476 \def\XINT_min_plusminus #1#2#3#4{\xint_firstoftwo_andstop }%
1477 \def\XINT_min_plusplus #1#2#3#4%
1478 {%
1479     \ifodd\XINT_Geq {#4#2}{#3#1}
1480     \expandafter\xint_secondoftwo_andstop
1481     \else
1482     \expandafter\xint_firstoftwo_andstop
1483     \fi
1484 }%

#3=-, #4=-, #1 = |B| = -B, #2 = |A| = -A

1485 \def\XINT_min_minusminus #1#2#3#4%
1486 {%
1487     \ifodd\XINT_Geq {#1}{#2}
1488     \expandafter\xint_secondoftwo_andstop
1489     \else
1490     \expandafter\xint_firstoftwo_andstop
1491     \fi
1492 }%

```

34.43 \xintMinof

1.09a

```

1493 \def\xintiMinof {\romannumeral0\xintiminof }%
1494 \def\xintiminof #1{\expandafter\XINT_iminof_a\romannumeral-‘0#1\relax }%
1495 \def\XINT_iminof_a #1{\expandafter\XINT_iminof_b\romannumeral0\xintnum{#1}\Z }%
1496 \def\XINT_iminof_b #1\Z #2%
1497     {\expandafter\XINT_iminof_c\romannumeral-‘0#2\Z {#1}\Z}%
1498 \def\XINT_iminof_c #1%
1499     {\xint_gob_til_relax #1\XINT_iminof_e\relax\XINT_iminof_d #1}%
1500 \def\XINT_iminof_d #1\Z
1501     {\expandafter\XINT_iminof_b\romannumeral0\xintimin {#1}}%
1502 \def\XINT_iminof_e #1\Z #2\Z { #2}%
1503 \let\xintMinof\xintiMinof \let\xintminof\xintiminof

```

34.44 \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 the \xintnum overhead. 1.09h renames \xintiSum to \xintiiSum to correctly reflect this.

```

1504 \def\xintiiSum {\romannumeral0\xintiisum }%
1505 \def\xintiisum #1{\xintiisumexpr #1\relax }%
1506 \def\xintiiSumExpr {\romannumeral0\xintiisumexpr }%
1507 \def\xintiisumexpr {\expandafter\XINT_sumexpr\romannumeral-‘0}%
1508 \let\xintSum\xintiiSum \let\xintsum\xintiisum
1509 \let\xintSumExpr\xintiiSumExpr \let\xintsumexpr\xintiisumexpr
1510 \def\XINT_sumexpr {\XINT_sum_loop {0000}{0000}}%
1511 \def\XINT_sum_loop #1#2#3%
1512 {%
1513   \expandafter\XINT_sum_checksign\romannumeral-‘0#3\Z {#1}{#2}%
1514 }%
1515 \def\XINT_sum_checksign #1%
1516 {%
1517   \xint_gob_til_relax #1\XINT_sum_finished\relax
1518   \xint_gob_til_zero #1\XINT_sum_skipzeroinput0%
1519   \xint_UDsignfork
1520   #1\dummy \XINT_sum_N
1521   -\dummy {\XINT_sum_P #1}%
1522   \krof
1523 }%
1524 \def\XINT_sum_finished #1\Z #2#3%
1525 {%
1526   \XINT_sub_A 1{ }#3\W\X\Y\Z #2\W\X\Y\Z
1527 }%
1528 \def\XINT_sum_skipzeroinput #1\krof #2\Z {\XINT_sum_loop }%
1529 \def\XINT_sum_P #1\Z #2%
1530 {%
1531   \expandafter\XINT_sum_loop\expandafter
1532   {\romannumeral0\expandafter
1533   \XINT_addr_A\expandafter0\expandafter{\expandafter}%
1534   \romannumeral0\XINT_RQ { }#1\R\R\R\R\R\R\R\Z
1535   \W\X\Y\Z #2\W\X\Y\Z }%
1536 }%
1537 \def\XINT_sum_N #1\Z #2#3%
1538 {%
1539   \expandafter\XINT_sum_NN\expandafter
1540   {\romannumeral0\expandafter
1541   \XINT_addr_A\expandafter0\expandafter{\expandafter}%

```

```

1542 \romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z
1543 \W\X\Y\Z #3\W\X\Y\Z {}#2}%
1544 }%
1545 \def\XINT_sum_NN #1#2{\XINT_sum_loop {}#2}{}#1}%

```

34.45 \xintMul

1.09a adds \xintnum

```

1546 \def\xintiMul {\romannumeral0\xintiimul }%
1547 \def\xintiimul #1%
1548 {%
1549 \expandafter\xint_iimul\expandafter {\romannumeral-‘0#1}%
1550 }%
1551 \def\xint_iimul #1#2%
1552 {%
1553 \expandafter\XINT_mul_fork \romannumeral-‘0#2\Z #1\Z
1554 }%
1555 \def\xintiMul {\romannumeral0\xintimul }%
1556 \def\xintimul #1%
1557 {%
1558 \expandafter\xint_mul\expandafter {\romannumeral0\xintnum{#1}}%
1559 }%
1560 \def\xint_mul #1#2%
1561 {%
1562 \expandafter\XINT_mul_fork \romannumeral0\xintnum{#2}\Z #1\Z
1563 }%
1564 \let\xintMul\xintiMul \let\xintmul\xintimul
1565 \def\XINT_Mul #1#2{\romannumeral0\XINT_mul_fork #2\Z #1\Z }%

```

MULTIPLICATION

Ici #1#2 = 2e input et #3#4 = 1er input

Release 1.03 adds some overhead to first compute and compare the lengths of the two inputs. The algorithm is asymmetrical and whether the first input is the longest or the shortest sometimes has a strong impact. 50 digits times 1000 digits used to be 5 times faster than 1000 digits times 50 digits. With the new code, the user input order does not matter as it is decided by the routine what is best. This is important for the extension to fractions, as there is no way then to generally control or guess the most frequent sizes of the inputs besides actually computing their lengths.

```

1566 \def\XINT_mul_fork #1#2\Z #3#4\Z
1567 {%
1568 \xint_UDzerofork
1569 #1\dummy \XINT_mul_zero
1570 #3\dummy \XINT_mul_zero
1571 0\dummy
1572 {\xint_UDsignsfork
1573 #1#3\dummy \XINT_mul_minusminus % #1 = #3 = -

```

34 Package *xint* implementation

```

1574      #1-\dummy {\XINT_mul_minusplus #3}%           % #1 = -
1575      #3-\dummy {\XINT_mul_plusminus #1}%           % #3 = -
1576      --\dummy {\XINT_mul_plusplus #1#3}%
1577      \krof }%
1578      \krof
1579      {#2}{#4}%
1580 }%
1581 \def\XINT_mul_zero #1#2{ 0}%
1582 \def\XINT_mul_minusminus #1#2%
1583 {%
1584      \expandafter\XINT_mul_choice_a
1585      \expandafter{\romannumeral0\xintlength {#2}}%
1586      {\romannumeral0\xintlength {#1}}{#1}{#2}%
1587 }%
1588 \def\XINT_mul_minusplus #1#2#3%
1589 {%
1590      \expandafter\xint_minus_andstop\romannumeral0\expandafter
1591      \XINT_mul_choice_a
1592      \expandafter{\romannumeral0\xintlength {#1#3}}%
1593      {\romannumeral0\xintlength {#2}}{#2}{#1#3}%
1594 }%
1595 \def\XINT_mul_plusminus #1#2#3%
1596 {%
1597      \expandafter\xint_minus_andstop\romannumeral0\expandafter
1598      \XINT_mul_choice_a
1599      \expandafter{\romannumeral0\xintlength {#3}}%
1600      {\romannumeral0\xintlength {#1#2}}{#1#2}{#3}%
1601 }%
1602 \def\XINT_mul_plusplus #1#2#3#4%
1603 {%
1604      \expandafter\XINT_mul_choice_a
1605      \expandafter{\romannumeral0\xintlength {#2#4}}%
1606      {\romannumeral0\xintlength {#1#3}}{#1#3}{#2#4}%
1607 }%
1608 \def\XINT_mul_choice_a #1#2%
1609 {%
1610      \expandafter\XINT_mul_choice_b\expandafter{#2}{#1}%
1611 }%
1612 \def\XINT_mul_choice_b #1#2%
1613 {%
1614      \ifnum #1<\xint_c_v
1615          \expandafter\XINT_mul_choice_littlebyfirst
1616      \else
1617          \ifnum #2<\xint_c_v
1618              \expandafter\expandafter\expandafter\XINT_mul_choice_littlebysecond
1619          \else
1620              \expandafter\expandafter\expandafter\XINT_mul_choice_compare
1621          \fi
1622      \fi

```

```

1623   {#1}{#2}%
1624 }%
1625 \def\XINT_mul_choice_littlebyfirst #1#2#3#4%
1626 {%
1627   \expandafter\XINT_mul_M
1628   \expandafter{\the\numexpr #3\expandafter}%
1629   \romannumeral0\XINT_RQ {}#4\R\R\R\R\R\R\R\R\Z \Z\Z\Z\Z
1630 }%
1631 \def\XINT_mul_choice_littlebysecond #1#2#3#4%
1632 {%
1633   \expandafter\XINT_mul_M
1634   \expandafter{\the\numexpr #4\expandafter}%
1635   \romannumeral0\XINT_RQ {}#3\R\R\R\R\R\R\R\R\Z \Z\Z\Z\Z
1636 }%
1637 \def\XINT_mul_choice_compare #1#2%
1638 {%
1639   \ifnum #1>#2
1640     \expandafter \XINT_mul_choice_i
1641   \else
1642     \expandafter \XINT_mul_choice_ii
1643   \fi
1644   {#1}{#2}%
1645 }%
1646 \def\XINT_mul_choice_i #1#2%
1647 {%
1648   \ifnum #1<\numexpr\ifcase \numexpr (#2-\xint_c_iii)/\xint_c_iv\relax
1649     \or 330\or 168\or 109\or 80\or 66\or 52\else 0\fi\relax
1650     \expandafter\XINT_mul_choice_same
1651   \else
1652     \expandafter\XINT_mul_choice_permute
1653   \fi
1654 }%
1655 \def\XINT_mul_choice_ii #1#2%
1656 {%
1657   \ifnum #2<\numexpr\ifcase \numexpr (#1-\xint_c_iii)/\xint_c_iv\relax
1658     \or 330\or 168\or 109\or 80\or 66\or 52\else 0\fi\relax
1659     \expandafter\XINT_mul_choice_permute
1660   \else
1661     \expandafter\XINT_mul_choice_same
1662   \fi
1663 }%
1664 \def\XINT_mul_choice_same #1#2%
1665 {%
1666   \expandafter\XINT_mul_enter
1667   \romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z
1668   \Z\Z\Z\Z #2\W\W\W\W
1669 }%
1670 \def\XINT_mul_choice_permute #1#2%
1671 {%

```

```

1672 \expandafter\XINT_mul_enter
1673 \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
1674 \Z\Z\Z\Z #1\W\W\W\W
1675 }%

```

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 $4n$.

```

1676 \def\XINT_mul_Ar #1#2#3#4#5#6%
1677 {%
1678 \xint_gob_til_Z #6\xint_mul_br\Z\XINT_mul_Br #1{#6#5#4#3}{#2}%
1679 }%
1680 \def\xint_mul_br\Z\XINT_mul_Br #1#2%
1681 {%
1682 \XINT_addr_AC_checkcarry #1%
1683 }%
1684 \def\XINT_mul_Br #1#2#3#4\W\X\Y\Z #5#6#7#8%
1685 {%
1686 \expandafter\XINT_mul_ABEAr
1687 \the\numexpr #1+10#2+#8#7#6#5.{#3}#4\W\X\Y\Z
1688 }%
1689 \def\XINT_mul_ABEAr #1#2#3#4#5#6.#7%
1690 {%
1691 \XINT_mul_Ar #2{#7#6#5#4#3}%
1692 }%

```

<< 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.

```

1693 \def\XINT_mul_Mr #1%
1694 {%
1695 \expandafter\XINT_mul_Mr_checkifzeroorone\expandafter{\the\numexpr #1}%
1696 }%
1697 \def\XINT_mul_Mr_checkifzeroorone #1%
1698 {%
1699 \ifcase #1
1700 \expandafter\XINT_mul_Mr_zero
1701 \or
1702 \expandafter\XINT_mul_Mr_one
1703 \else
1704 \expandafter\XINT_mul_Nr
1705 \fi
1706 {0000}{#1}%
1707 }%
1708 \def\XINT_mul_Mr_zero #1\Z\Z\Z\Z { 0000}%
1709 \def\XINT_mul_Mr_one #1#2#3#4\Z\Z\Z\Z { #4}%

```

34 Package *xint* implementation

```

1710 \def\XINT_mul_Nr #1#2#3#4#5#6#7%
1711 {%
1712   \xint_gob_til_Z #4\xint_mul_pr\Z\XINT_mul_Pr {#1}{#3}{#7#6#5#4}{#2}{#3}%
1713 }%
1714 \def\XINT_mul_Pr #1#2#3%
1715 {%
1716   \expandafter\XINT_mul_Lr\the\numexpr \xint_c_x^viii+#1+#2*#3\relax
1717 }%
1718 \def\XINT_mul_Lr 1#1#2#3#4#5#6#7#8#9%
1719 {%
1720   \XINT_mul_Nr {#1#2#3#4}{#9#8#7#6#5}%
1721 }%
1722 \def\xint_mul_pr\Z\XINT_mul_Pr #1#2#3#4#5%
1723 {%
1724   \xint_gob_til_zeros_iv #1\XINT_mul_Mr_end_nocarry 0000%
1725   \XINT_mul_Mr_end_carry #1{#4}%
1726 }%
1727 \def\XINT_mul_Mr_end_nocarry 0000\XINT_mul_Mr_end_carry 0000#1{ #1}%
1728 \def\XINT_mul_Mr_end_carry #1#2#3#4#5{ #5#4#3#2#1}%

  << Petite >> multiplication. renvoie le résultat *à l'endroit*, avec *nettoyage
  des leading zéros*.
  \romannumeral0\XINT_mul_M {<n>}<N>\Z\Z\Z
  Fait la multiplication de <N> par <n>, qui est < 10000. <N> est présenté *à
  l'envers*, sur *4n*.

1729 \def\XINT_mul_M #1%
1730 {%
1731   \expandafter\XINT_mul_M_checkifzeroorone\expandafter{\the\numexpr #1}%
1732 }%
1733 \def\XINT_mul_M_checkifzeroorone #1%
1734 {%
1735   \ifcase #1
1736     \expandafter\XINT_mul_M_zero
1737   \or
1738     \expandafter\XINT_mul_M_one
1739   \else
1740     \expandafter\XINT_mul_N
1741   \fi
1742   {0000}{#1}%
1743 }%
1744 \def\XINT_mul_M_zero #1\Z\Z\Z\Z { 0}%
1745 \def\XINT_mul_M_one #1#2#3#4\Z\Z\Z\Z
1746 {%
1747   \expandafter\xint_cleanupzeros_andstop\romannumeral0\xintreverseorder{#4}%
1748 }%
1749 \def\XINT_mul_N #1#2#3#4#5#6#7%
1750 {%
1751   \xint_gob_til_Z #4\xint_mul_p\Z\XINT_mul_P {#1}{#3}{#7#6#5#4}{#2}{#3}%
1752 }%

```



```

1753 \def\XINT_mul_P #1#2#3%
1754 {%
1755   \expandafter\XINT_mul_L\the\numexpr \xint_c_x^viii+#1+#2*#3\relax
1756 }%
1757 \def\XINT_mul_L #1#2#3#4#5#6#7#8#9%
1758 {%
1759   \XINT_mul_N {#1#2#3#4}{#5#6#7#8#9}%
1760 }%
1761 \def\xint_mul_p\Z\XINT_mul_P #1#2#3#4#5%
1762 {%
1763   \XINT_mul_M_end #1#4%
1764 }%
1765 \def\XINT_mul_M_end #1#2#3#4#5#6#7#8%
1766 {%
1767   \expandafter\space\the\numexpr #1#2#3#4#5#6#7#8\relax
1768 }%

```

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 la 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.

```

1769 \def\XINT_mul_enter #1\Z\Z\Z\Z #2#3#4#5%
1770 {%
1771   \xint_gob_til_W #5\XINT_mul_exit_a\W
1772   \XINT_mul_start {#2#3#4#5}#1\Z\Z\Z\Z
1773 }%
1774 \def\XINT_mul_exit_a\W\XINT_mul_start #1%
1775 {%
1776   \XINT_mul_exit_b #1%
1777 }%
1778 \def\XINT_mul_exit_b #1#2#3#4%
1779 {%
1780   \xint_gob_til_W
1781   #2\XINT_mul_exit_ci
1782   #3\XINT_mul_exit_cii
1783   \W\XINT_mul_exit_ciii #1#2#3#4%

```

```

1784 }%
1785 \def\XINT_mul_exit_ciii #1\W #2\Z\Z\Z\Z \W\W\W
1786 {%
1787   \XINT_mul_M {#1}#2\Z\Z\Z\Z
1788 }%
1789 \def\XINT_mul_exit_cii\W\XINT_mul_exit_ciii #1\W\W #2\Z\Z\Z\Z \W\W
1790 {%
1791   \XINT_mul_M {#1}#2\Z\Z\Z\Z
1792 }%
1793 \def\XINT_mul_exit_ci\W\XINT_mul_exit_cii
1794   \W\XINT_mul_exit_ciii #1\W\W\W #2\Z\Z\Z\Z \W
1795 {%
1796   \XINT_mul_M {#1}#2\Z\Z\Z\Z
1797 }%
1798 \def\XINT_mul_start #1#2\Z\Z\Z\Z
1799 {%
1800   \expandafter\XINT_mul_main\expandafter
1801   {\romannumeral0\XINT_mul_Mr {#1}#2\Z\Z\Z\Z}#2\Z\Z\Z\Z
1802 }%
1803 \def\XINT_mul_main #1#2\Z\Z\Z\Z #3#4#5#6%
1804 {%
1805   \xint_gob_til_W #6\XINT_mul_finish_a\W
1806   \XINT_mul_compute {#3#4#5#6}{#1}#2\Z\Z\Z\Z
1807 }%
1808 \def\XINT_mul_compute #1#2#3\Z\Z\Z\Z
1809 {%
1810   \expandafter\XINT_mul_main\expandafter
1811   {\romannumeral0\expandafter
1812   \XINT_mul_Ar\expandafter0\expandafter{\expandafter}%
1813   \romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z
1814   \W\X\Y\Z 0000#2\W\X\Y\Z }#3\Z\Z\Z\Z
1815 }%

```

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.

```

1816 \def\XINT_mul_finish_a\W\XINT_mul_compute #1%
1817 {%
1818   \XINT_mul_finish_b #1%
1819 }%
1820 \def\XINT_mul_finish_b #1#2#3#4%
1821 {%
1822   \xint_gob_til_W
1823   #1\XINT_mul_finish_c
1824   #2\XINT_mul_finish_ci
1825   #3\XINT_mul_finish_cii
1826   \W\XINT_mul_finish_ciii #1#2#3#4%
1827 }%

```

```

1828 \def\XINT_mul_finish_ciii #1\W #2#3\Z\Z\Z\Z \W\W\W
1829 {%
1830   \expandafter\XINT_addm_A\expandafter0\expandafter{\expandafter}%
1831   \romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z \W\X\Y\Z 000#2\W\X\Y\Z
1832 }%
1833 \def\XINT_mul_finish_cii
1834   \W\XINT_mul_finish_ciii #1\W\W #2#3\Z\Z\Z\Z \W\W
1835 {%
1836   \expandafter\XINT_addm_A\expandafter0\expandafter{\expandafter}%
1837   \romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z \W\X\Y\Z 00#2\W\X\Y\Z
1838 }%
1839 \def\XINT_mul_finish_ci #1\XINT_mul_finish_ciii #2\W\W\W #3#4\Z\Z\Z\Z \W
1840 {%
1841   \expandafter\XINT_addm_A\expandafter0\expandafter{\expandafter}%
1842   \romannumeral0\XINT_mul_Mr {#2}#4\Z\Z\Z\Z \W\X\Y\Z 0#3\W\X\Y\Z
1843 }%
1844 \def\XINT_mul_finish_c #1\XINT_mul_finish_ciii \W\W\W\W #2#3\Z\Z\Z\Z
1845 {%
1846   \expandafter\xint_cleanupzeros_andstop\romannumeral0\xintreverseorder{#2}%
1847 }%

```

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.

```

1848 \def\XINT_mulr_enter #1\Z\Z\Z\Z #2#3#4#5%
1849 {%
1850   \xint_gob_til_W #5\XINT_mulr_exit_a\W
1851   \XINT_mulr_start {#2#3#4#5}#1\Z\Z\Z\Z
1852 }%
1853 \def\XINT_mulr_exit_a\W\XINT_mulr_start #1%
1854 {%
1855   \XINT_mulr_exit_b #1%
1856 }%
1857 \def\XINT_mulr_exit_b #1#2#3#4%
1858 {%
1859   \xint_gob_til_W
1860   #2\XINT_mulr_exit_ci
1861   #3\XINT_mulr_exit_cii
1862   \W\XINT_mulr_exit_ciii #1#2#3#4%
1863 }%
1864 \def\XINT_mulr_exit_ciii #1\W #2\Z\Z\Z\Z \W\W\W\W
1865 {%
1866   \XINT_mul_Mr {#1}#2\Z\Z\Z\Z
1867 }%

```

34 Package *xint* implementation

```

1868 \def\XINT_mulr_exit_cii\W\XINT_mulr_exit_ciii #1\W\W #2\Z\Z\Z\Z \W\W
1869 {%
1870     \XINT_mul_Mr {#1}#2\Z\Z\Z\Z
1871 }%
1872 \def\XINT_mulr_exit_ci\W\XINT_mulr_exit_cii
1873     \W\XINT_mulr_exit_ciii #1\W\W\W #2\Z\Z\Z\Z \W
1874 {%
1875     \XINT_mul_Mr {#1}#2\Z\Z\Z\Z
1876 }%
1877 \def\XINT_mulr_start #1#2\Z\Z\Z\Z
1878 {%
1879     \expandafter\XINT_mulr_main\expandafter
1880     {\romannumeral0\XINT_mul_Mr {#1}#2\Z\Z\Z\Z}#2\Z\Z\Z\Z
1881 }%
1882 \def\XINT_mulr_main #1#2\Z\Z\Z\Z #3#4#5#6%
1883 {%
1884     \xint_gob_til_W #6\XINT_mulr_finish_a\W
1885     \XINT_mulr_compute {#3#4#5#6}{#1}#2\Z\Z\Z\Z
1886 }%
1887 \def\XINT_mulr_compute #1#2#3\Z\Z\Z\Z
1888 {%
1889     \expandafter\XINT_mulr_main\expandafter
1890     {\romannumeral0\expandafter
1891     \XINT_mul_Ar\expandafter0\expandafter{\expandafter}%
1892     \romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z
1893     \W\X\Y\Z 0000#2\W\X\Y\Z }#3\Z\Z\Z\Z
1894 }%
1895 \def\XINT_mulr_finish_a\W\XINT_mulr_compute #1%
1896 {%
1897     \XINT_mulr_finish_b #1%
1898 }%
1899 \def\XINT_mulr_finish_b #1#2#3#4%
1900 {%
1901     \xint_gob_til_W
1902     #1\XINT_mulr_finish_c
1903     #2\XINT_mulr_finish_ci
1904     #3\XINT_mulr_finish_cii
1905     \W\XINT_mulr_finish_ciii #1#2#3#4%
1906 }%
1907 \def\XINT_mulr_finish_ciii #1\W #2#3\Z\Z\Z\Z \W\W\W
1908 {%
1909     \expandafter\XINT_addp_A\expandafter0\expandafter{\expandafter}%
1910     \romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z \W\X\Y\Z 000#2\W\X\Y\Z
1911 }%
1912 \def\XINT_mulr_finish_cii
1913     \W\XINT_mulr_finish_ciii #1\W\W #2#3\Z\Z\Z\Z \W\W
1914 {%
1915     \expandafter\XINT_addp_A\expandafter0\expandafter{\expandafter}%
1916     \romannumeral0\XINT_mul_Mr {#1}#3\Z\Z\Z\Z \W\X\Y\Z 00#2\W\X\Y\Z

```

```

1917 }%
1918 \def\XINT_mulr_finish_ci #1\XINT_mulr_finish_ciii #2\W\W\W #3#4\Z\Z\Z\Z \W
1919 {%
1920   \expandafter\XINT_addp_A\expandafter0\expandafter{\expandafter}%
1921   \romannumeral0\XINT_mul_Mr {#2}#4\Z\Z\Z\Z \W\X\Y\Z 0#3\W\X\Y\Z
1922 }%
1923 \def\XINT_mulr_finish_c #1\XINT_mulr_finish_ciii \W\W\W\W #2#3\Z\Z\Z\Z { #2}%

```

34.46 \xintSqr

```

1924 \def\xintiisqr {\romannumeral0\xintiisqr }%
1925 \def\xintiisqr #1%
1926 {%
1927   \expandafter\XINT_sqr\expandafter {\romannumeral0\xintiisqr{#1}}%
1928 }%
1929 \def\xintiSqr {\romannumeral0\xintisqr }%
1930 \def\xintisqr #1%
1931 {%
1932   \expandafter\XINT_sqr\expandafter {\romannumeral0\xintiisqr{#1}}%
1933 }%
1934 \let\xintSqr\xintiSqr \let\xintsqr\xintisqr
1935 \def\XINT_sqr #1%
1936 {%
1937   \expandafter\XINT_mul_enter
1938   \romannumeral0%
1939   \XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z
1940   \Z\Z\Z\Z #1\W\W\W\W
1941 }%

```

34.47 \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.

1.09a does NOT add the `\xintnum` overhead. 1.09h renames `\xintiPrd` to `\xintiiPrd` to correctly reflect this.

```

1942 \def\xintiiPrd {\romannumeral0\xintiiprd }%
1943 \def\xintiiprd #1{\xintiiprdexpr #1\relax }%
1944 \let\xintPrd\xintiiPrd
1945 \let\xintprd\xintiiprd
1946 \def\xintiiPrdExpr {\romannumeral0\xintiiprdexpr }%
1947 \def\xintiiprdexpr {\expandafter\XINT_prdexpr\romannumeral-‘0}%
1948 \let\xintPrdExpr\xintiiPrdExpr
1949 \let\xintprdexpr\xintiiprdexpr
1950 \def\XINT_prdexpr {\XINT_prod_loop_a 1\Z }%
1951 \def\XINT_prod_loop_a #1\Z #2%
1952 {%
1953   \expandafter\XINT_prod_loop_b \romannumeral-‘0#2\Z #1\Z \Z
1954 }%
1955 \def\XINT_prod_loop_b #1%
1956 {%
1957   \xint_gob_til_relax #1\XINT_prod_finished\relax
1958   \XINT_prod_loop_c #1%
1959 }%
1960 \def\XINT_prod_loop_c
1961 {%
1962   \expandafter\XINT_prod_loop_a\romannumeral0\XINT_mul_fork
1963 }%
1964 \def\XINT_prod_finished #1\Z #2\Z \Z { #2}%

```

34.48 \xintFac

Modified with 1.02 and again in 1.03 for greater efficiency. I am tempted, here and elsewhere, to use `\ifcase\XINT_Geq {#1}{1000000000}` rather than `\ifnum\xintLength {#1}>9` but for the time being I leave things as they stand. With release 1.05, rather than using `\xintLength` I opt finally for direct use of `\numexpr` (which will throw a suitable number too big message), and to raise the `\xintError:FactorialOfTooBigNumber` for argument larger than 1000000 (rather than 1000000000). With 1.09a, `\xintFac` uses `\xintnum`.

```

1965 \def\xintiFac {\romannumeral0\xintifac }%
1966 \def\xintifac #1%
1967 {%
1968   \expandafter\XINT_fac_fork\expandafter{\the\numexpr #1}%
1969 }%
1970 \let\xintFac\xintiFac \let\xintfac\xintifac
1971 \def\XINT_fac_fork #1%
1972 {%
1973   \ifcase\XINT_Sgn {#1}
1974     \xint_afterfi{\expandafter\space\expandafter 1\xint_gobble_i }%
1975   \or
1976     \expandafter\XINT_fac_checklength
1977   \else
1978     \xint_afterfi{\expandafter\xintError:FactorialOfNegativeNumber
1979       \expandafter\space\expandafter 1\xint_gobble_i }%

```

```

1980 \fi
1981 {#1}%
1982 }%
1983 \def\XINT_fac_checklength #1%
1984 {%
1985 \ifnum #1>999999
1986 \xint_afterfi{\expandafter\xintError:FactorialOfTooBigNumber
1987 \expandafter\space\expandafter 1\xint_gobble_i }%
1988 \else
1989 \xint_afterfi{\ifnum #1>9999
1990 \expandafter\XINT_fac_big_loop
1991 \else
1992 \expandafter\XINT_fac_loop
1993 \fi }%
1994 \fi
1995 {#1}%
1996 }%
1997 \def\XINT_fac_big_loop #1{\XINT_fac_big_loop_main {10000}{#1}{}}%
1998 \def\XINT_fac_big_loop_main #1#2#3%
1999 {%
2000 \ifnum #1<#2
2001 \expandafter
2002 \XINT_fac_big_loop_main
2003 \expandafter
2004 {\the\numexpr #1+1\expandafter }%
2005 \else
2006 \expandafter\XINT_fac_big_docomputation
2007 \fi
2008 {#2}{#3{#1}}%
2009 }%
2010 \def\XINT_fac_big_docomputation #1#2%
2011 {%
2012 \expandafter \XINT_fac_bigcompute_loop \expandafter
2013 {\romannumeral0\XINT_fac_loop {9999}}#2\relax
2014 }%
2015 \def\XINT_fac_bigcompute_loop #1#2%
2016 {%
2017 \xint_gob_til_relax #2\XINT_fac_bigcompute_end\relax
2018 \expandafter\XINT_fac_bigcompute_loop\expandafter
2019 {\expandafter\XINT_mul_enter
2020 \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
2021 \Z\Z\Z\Z #1\W\W\W\W }%
2022 }%
2023 \def\XINT_fac_bigcompute_end #1#2#3#4#5%
2024 {%
2025 \XINT_fac_bigcompute_end_ #5%
2026 }%
2027 \def\XINT_fac_bigcompute_end_ #1\R #2\Z \W\X\Y\Z #3\W\X\Y\Z { #3}%
2028 \def\XINT_fac_loop #1{\XINT_fac_loop_main 1{1000}{#1}}%

```

```

2029 \def\XINT_fac_loop_main #1#2#3%
2030 {%
2031   \ifnum #3>#1
2032   \else
2033     \expandafter\XINT_fac_loop_exit
2034   \fi
2035   \expandafter\XINT_fac_loop_main\expandafter
2036   {\the\numexpr #1+1\expandafter }\expandafter
2037   {\romannumeral0\XINT_mul_Mr {#1}#2\Z\Z\Z }%
2038   {#3}%
2039 }%
2040 \def\XINT_fac_loop_exit #1#2#3#4#5#6#7%
2041 {%
2042   \XINT_fac_loop_exit_ #6%
2043 }%
2044 \def\XINT_fac_loop_exit_ #1#2#3%
2045 {%
2046   \XINT_mul_M
2047 }%

```

34.49 \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. However this added some overhead to some inner macros of the \xintPow routine of xintfrac.sty... we did the similar things correctly for \xintiadd etc, but not here, so 1.09f has now the necessary \xintiipow.

```

2048 \def\xintiipow {\romannumeral0\xintiipow }%
2049 \def\xintiipow #1%
2050 {%
2051   \expandafter\xint_pow\romannumeral-‘0#1\Z%
2052 }%
2053 \def\xintiPow {\romannumeral0\xintipow }%
2054 \def\xintipow #1%
2055 {%
2056   \expandafter\xint_pow\romannumeral0\xintnum{#1}\Z%
2057 }%
2058 \let\xintPow\xintiPow \let\xintpow\xintipow
2059 \def\xint_pow #1#2\Z
2060 {%
2061   \xint_UDsignfork
2062   #1\dummy \XINT_pow_Aneg
2063   -\dummy \XINT_pow_Anonneg
2064   \krof
2065   #1{#2}%

```


34 Package *xint* implementation

```

2066 }%
2067 \def\XINT_pow_Aneg #1#2#3%
2068 {%
2069   \expandafter\XINT_pow_Aneg_\expandafter{\the\numexpr #3}{#2}%
2070 }%
2071 \def\XINT_pow_Aneg_ #1%
2072 {%
2073   \ifodd #1
2074     \expandafter\XINT_pow_Aneg_Bodd
2075   \fi
2076   \XINT_pow_Anonneg_ {#1}%
2077 }%
2078 \def\XINT_pow_Aneg_Bodd #1%
2079 {%
2080   \expandafter\XINT_opp\romannumeral0\XINT_pow_Anonneg_
2081 }%

  B = #3, faire le xpxp. Modified with 1.06: use of \numexpr.

2082 \def\XINT_pow_Anonneg #1#2#3%
2083 {%
2084   \expandafter\XINT_pow_Anonneg_\expandafter {\the\numexpr #3}{#1#2}%
2085 }%

  #1 = B, #2 = |A|

2086 \def\XINT_pow_Anonneg_ #1#2%
2087 {%
2088   \ifcase\XINT_Cmp {#2}{1}
2089     \expandafter\XINT_pow_AisOne
2090   \or
2091     \expandafter\XINT_pow_AatleastTwo
2092   \else
2093     \expandafter\XINT_pow_AisZero
2094   \fi
2095   {#1}{#2}%
2096 }%
2097 \def\XINT_pow_AisOne #1#2{ 1}%

  #1 = B

2098 \def\XINT_pow_AisZero #1#2%
2099 {%
2100   \ifcase\XINT_Sgn {#1}
2101     \xint_afterfi { 1}%
2102   \or
2103     \xint_afterfi { 0}%
2104   \else
2105     \xint_afterfi {\xintError:DivisionByZero\space 0}%
2106   \fi

```

```

2107}%
2108\def\XINT_pow_AatleastTwo #1%
2109{%
2110  \ifcase\XINT_Sgn {#1}
2111    \expandafter\XINT_pow_BisZero
2112  \or
2113    \expandafter\XINT_pow_checkBsize
2114  \else
2115    \expandafter\XINT_pow_BisNegative
2116  \fi
2117  {#1}%
2118}%
2119\def\XINT_pow_BisNegative #1#2{\xintError:FractionRoundedToZero\space 0}%
2120\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.

2121\def\XINT_pow_checkBsize #1#2%
2122{%
2123  \ifnum #1>999999999
2124    \expandafter\XINT_pow_BtooBig
2125  \else
2126    \expandafter\XINT_pow_loop
2127  \fi
2128  {#1}{#2}\XINT_pow_posprod
2129  \xint_relax
2130  \xint_bye\xint_bye\xint_bye\xint_bye
2131  \xint_bye\xint_bye\xint_bye\xint_bye
2132  \xint_relax
2133}%
2134\def\XINT_pow_BtooBig #1\xint_relax #2\xint_relax
2135  {\xintError:ExponentTooBig\space 0}%
2136\def\XINT_pow_loop #1#2%
2137{%
2138  \ifnum #1 = 1
2139    \expandafter\XINT_pow_loop_end
2140  \else
2141    \xint_afterfi{\expandafter\XINT_pow_loop_a
2142      \expandafter{\the\numexpr 2*(#1/2)-#1\expandafter }% b mod 2
2143      \expandafter{\the\numexpr #1-#1/2\expandafter }% [b/2]
2144      \expandafter{\romannumeral0\xintiisqr{#2}}}%
2145    \fi
2146    {{#2}}}%
2147}%
2148\def\XINT_pow_loop_end {\romannumeral0\XINT_rord_main {} \relax}%
2149\def\XINT_pow_loop_a #1%
2150{%
2151  \ifnum #1 = 1

```

```

2152      \expandafter\XINT_pow_loop
2153    \else
2154      \expandafter\XINT_pow_loop_throwaway
2155    \fi
2156 }%
2157 \def\XINT_pow_loop_throwaway #1#2#3%
2158 {%
2159   \XINT_pow_loop {#1}{#2}%
2160 }%

```

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.

```

2161 \def\XINT_pow_posprod #1%
2162 {%
2163   \XINT_pow_pprod_checkifempty #1\Z
2164 }%
2165 \def\XINT_pow_pprod_checkifempty #1%
2166 {%
2167   \xint_gob_til_relax #1\XINT_pow_pprod_emptyproduct\relax
2168   \XINT_pow_pprod_RQfirst #1%
2169 }%
2170 \def\XINT_pow_pprod_emptyproduct #1\Z { 1}%
2171 \def\XINT_pow_pprod_RQfirst #1\Z
2172 {%
2173   \expandafter\XINT_pow_pprod_getnext\expandafter
2174   {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z}%
2175 }%
2176 \def\XINT_pow_pprod_getnext #1#2%
2177 {%
2178   \XINT_pow_pprod_checkiffinished #2\Z {#1}%
2179 }%
2180 \def\XINT_pow_pprod_checkiffinished #1%
2181 {%
2182   \xint_gob_til_relax #1\XINT_pow_pprod_end\relax
2183   \XINT_pow_pprod_compute #1%
2184 }%
2185 \def\XINT_pow_pprod_compute #1\Z #2%
2186 {%
2187   \expandafter\XINT_pow_pprod_getnext\expandafter
2188   {\romannumeral0\XINT_mulr_enter #2\Z\Z\Z\Z #1\W\W\W }%
2189 }%
2190 \def\XINT_pow_pprod_end\relax\XINT_pow_pprod_compute #1\Z #2%
2191 {%
2192   \expandafter\xint_cleanupzeros_andstop
2193   \romannumeral0\xintreverseorder {#2}%

```

2194 }%

34.50 `\xintDivision`, `\xintQuo`, `\xintRem`

1.09a inserts the use of `\xintnum`. However this was also used in internal macros in places it should not for reasons of efficiency, so in 1.09f I reinstall the private versions with less overhead. Besides, there was some duplicated code in `xintfrac.sty` which is removed.

```

2195 \def\xintiQuo {\romannumeral0\xintiQuo }%
2196 \def\xintiRem {\romannumeral0\xintiRem }%
2197 \def\xintiQuo {\expandafter\xint_firstoftwo_andstop
2198             \romannumeral0\xintiDivision }%
2199 \def\xintiRem {\expandafter\xint_secondoftwo_andstop
2200             \romannumeral0\xintiDivision }%
2201 \def\xintQuo {\romannumeral0\xintQuo }%
2202 \def\xintRem {\romannumeral0\xintRem }%
2203 \def\xintQuo {\expandafter\xint_firstoftwo_andstop
2204             \romannumeral0\xintDivision }%
2205 \def\xintRem {\expandafter\xint_secondoftwo_andstop
2206             \romannumeral0\xintDivision }%

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

2207 \def\xintiDivision #1%
2208 {%
2209     \expandafter\xint_iidivision\expandafter {\romannumeral-‘0#1}%
2210 }%
2211 \def\xint_iidivision #1#2%
2212 {%
2213     \expandafter\XINT_div_fork \romannumeral-‘0#2\Z #1\Z
2214 }%
2215 \def\xintDivision {\romannumeral0\xintDivision }%
2216 \def\xintDivision #1%
2217 {%
2218     \expandafter\xint_division\expandafter {\romannumeral0\xintnum{#1}}%
2219 }%
2220 \def\xint_division #1#2%
2221 {%
2222     \expandafter\XINT_div_fork \romannumeral0\xintnum{#2}\Z #1\Z
2223 }%
2224 \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

2225 \def\XINT_div_fork #1#2\Z #3#4\Z
2226 {%
2227     \xint_UDzerofork

```

```

2228      #1\dummy \XINT_div_BisZero
2229      #3\dummy \XINT_div_AisZero
2230      0\dummy
2231      {\xint_UDsignfork
2232          #1\dummy \XINT_div_BisNegative % B < 0
2233          #3\dummy \XINT_div_AisNegative % A < 0, B > 0
2234          -\dummy \XINT_div_plusplus % B > 0, A > 0
2235      \krof }%
2236      \krof
2237      {#2}{#4}#1#3% #1#2=B, #3#4=A
2238 }%
2239 \def\XINT_div_BisZero #1#2#3#4{\xintError:DivisionByZero\space {0}{0}}%
2240 \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.

```

2241 \def\XINT_div_plusplus #1#2#3#4%
2242 {%
2243     \XINT_div_prepare {#3#1}{#4#2}%
2244 }%

```

$B = \#3\#1 < 0$, A non nul positif ou négatif

```

2245 \def\XINT_div_BisNegative #1#2#3#4%
2246 {%
2247     \expandafter\XINT_div_BisNegative_post
2248     \romannumeral0\XINT_div_fork #1\Z #4#2\Z
2249 }%
2250 \def\XINT_div_BisNegative_post #1%
2251 {%
2252     \expandafter\space\expandafter {\romannumeral0\XINT_opp #1}%
2253 }%

```

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

```

2254 \def\XINT_div_AisNegative #1#2#3#4%
2255 {%
2256     \expandafter\XINT_div_AisNegative_post
2257     \romannumeral0\XINT_div_prepare {#3#1}{#2}{#3#1}%
2258 }%
2259 \def\XINT_div_AisNegative_post #1#2%
2260 {%
2261     \ifcase\XINT_Sgn {#2}
2262         \expandafter \XINT_div_AisNegative_zerorem

```

```

2263 \or
2264 \expandafter \XINT_div_AisNegative_posrem
2265 \fi
2266 {#1}{#2}%
2267 }%

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

2268 \def\XINT_div_AisNegative_zerorem #1#2#3%
2269 {%
2270 \expandafter\space\expandafter {\romannumeral0\XINT_opp #1}{0}%
2271 }%

#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  $a = qb + r$ ,  $0 \leq r < |b|$  est valable

2272 \def\XINT_div_AisNegative_posrem #1%
2273 {%
2274 \expandafter \XINT_div_AisNegative_posrem_b \expandafter
2275 {\romannumeral0\xintiioopp{\xintInc {#1}}}%
2276 }%
2277 \def\XINT_div_AisNegative_posrem_b #1#2#3%
2278 {%
2279 \expandafter \xint_exchangetwo_keepbraces_andstop \expandafter
2280 {\romannumeral0\XINT_sub {#3}{#2}}{#1}%
2281 }%

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

2282 \def\XINT_div_prepare #1%
2283 {%
2284 \expandafter \XINT_div_prepareB_aa \expandafter
2285 {\romannumeral0\xintlenght {#1}}{#1}% B  $> 0$  ici
2286 }%
2287 \def\XINT_div_prepareB_aa #1%
2288 {%
2289 \ifnum #1=1
2290 \expandafter\XINT_div_prepareB_ab
2291 \else
2292 \expandafter\XINT_div_prepareB_a
2293 \fi
2294 {#1}%
2295 }%
2296 \def\XINT_div_prepareB_ab #1#2%
2297 {%
2298 \ifnum #2=1
2299 \expandafter\XINT_div_prepareB_BisOne

```

34 Package *xint* implementation

```

2300 \else
2301 \expandafter\XINT_div_prepareB_e
2302 \fi {000}{3}{4}{#2}%
2303 }%
2304 \def\XINT_div_prepareB_BisOne #1#2#3#4#5{ {#5}{0}}%
2305 \def\XINT_div_prepareB_a #1%
2306 {%
2307 \expandafter\XINT_div_prepareB_c\expandafter
2308 {\the\numexpr \xint_c_iv*((#1+\xint_c_i)/\xint_c_iv)}{#1}%
2309 }%

#1 = K

2310 \def\XINT_div_prepareB_c #1#2%
2311 {%
2312 \ifcase \numexpr #1-#2\relax
2313 \expandafter\XINT_div_prepareB_d
2314 \or
2315 \expandafter\XINT_div_prepareB_di
2316 \or
2317 \expandafter\XINT_div_prepareB_dii
2318 \or
2319 \expandafter\XINT_div_prepareB_diii
2320 \fi {#1}%
2321 }%
2322 \def\XINT_div_prepareB_d {\XINT_div_prepareB_e {}{0}}%
2323 \def\XINT_div_prepareB_di {\XINT_div_prepareB_e {0}{1}}%
2324 \def\XINT_div_prepareB_dii {\XINT_div_prepareB_e {00}{2}}%
2325 \def\XINT_div_prepareB_diii {\XINT_div_prepareB_e {000}{3}}%

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

2326 \def\XINT_div_prepareB_e #1#2#3#4%
2327 {%
2328 \XINT_div_prepareB_f #4#1\Z {#3}{#2}{#1}%
2329 }%

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.

2330 \def\XINT_div_prepareB_f #1#2#3#4#5\Z
2331 {%
2332 \expandafter \XINT_div_prepareB_g \expandafter
2333 {\romannumeral0\xintreverseorder {#1#2#3#4#5}}{#1#2#3#4}%
2334 }%

#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

```

34 Package *xint* implementation

```

2335 \def\XINT_div_prepareB_g #1#2#3#4#5#6%
2336 {%
2337   \XINT_div_prepareA_a {#6#5}{#2}{#3}{#1}{#4}%
2338 }%

A, x, K, B, c,

2339 \def\XINT_div_prepareA_a #1%
2340 {%
2341   \expandafter \XINT_div_prepareA_b \expandafter
2342     {\romannumeral0\xintlength {#1}}{#1}% A >0 ici
2343 }%

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

2344 \def\XINT_div_prepareA_b #1%
2345 {%
2346   \expandafter\XINT_div_prepareA_c\expandafter{\the\numexpr 4*((#1+1)/4)}{#1}%
2347 }%

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

2348 \def\XINT_div_prepareA_c #1#2%
2349 {%
2350   \ifcase \numexpr #1-#2\relax
2351     \expandafter\XINT_div_prepareA_d
2352   \or
2353     \expandafter\XINT_div_prepareA_di
2354   \or
2355     \expandafter\XINT_div_prepareA_dii
2356   \or
2357     \expandafter\XINT_div_prepareA_diii
2358   \fi {#1}%
2359 }%
2360 \def\XINT_div_prepareA_d {\XINT_div_prepareA_e {}}%
2361 \def\XINT_div_prepareA_di {\XINT_div_prepareA_e {0}}%
2362 \def\XINT_div_prepareA_dii {\XINT_div_prepareA_e {00}}%
2363 \def\XINT_div_prepareA_diii {\XINT_div_prepareA_e {000}}%

#1#3 = A préparé, #2 = longueur de ce A préparé,

2364 \def\XINT_div_prepareA_e #1#2#3%
2365 {%
2366   \XINT_div_startswitch {#1#3}{#2}%
2367 }%

A, L, x, K, B, c

2368 \def\XINT_div_startswitch #1#2#3#4%
2369 {%
2370   \ifnum #2 > #4

```



```

2371 \expandafter\XINT_div_body_a
2372 \else
2373 \ifnum #2 = #4
2374 \expandafter\expandafter\expandafter\XINT_div_final_a
2375 \else
2376 \expandafter\expandafter\expandafter\XINT_div_finished_a
2377 \fi\fi {#1}{#4}{#3}{0000}{#2}%
2378 }%

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

2379 \def\XINT_div_finished_a #1#2#3%
2380 {%
2381 \expandafter\XINT_div_finished_b\expandafter {\romannumeral0\XINT_cuz {#1}}%
2382 }%

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

2383 \def\XINT_div_finished_b #1#2#3#4#5%
2384 {%
2385 \ifcase \XINT_Sgn {#1}
2386 \xint_afterfi {\XINT_div_finished_c {0}}%
2387 \or
2388 \xint_afterfi {\expandafter\XINT_div_finished_c\expandafter
2389 {\romannumeral0\XINT_dsh_checksngx #5\Z {#1}}%
2390 }%
2391 \fi
2392 {#2}%
2393 }%
2394 \def\XINT_div_finished_c #1#2%
2395 {%
2396 \expandafter\space\expandafter {\romannumeral0\XINT_rev_andcuz {#2}}{#1}%
2397 }%

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

2398 \def\XINT_div_final_a #1%
2399 {%
2400 \XINT_div_final_b #1\Z
2401 }%
2402 \def\XINT_div_final_b #1#2#3#4#5\Z
2403 {%
2404 \xint_gob_til_zeros_iv #1#2#3#4\xint_div_final_c0000%
2405 \XINT_div_final_c {#1#2#3#4}{#1#2#3#4#5}%
2406 }%
2407 \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).

```

```

2408 \def\XINT_div_final_c #1#2#3#4%
2409 {%
2410   \expandafter \XINT_div_final_da \expandafter
2411   {\the\numexpr #1-(#1/#4)*#4\expandafter }\expandafter
2412   {\the\numexpr #1/#4\expandafter }\expandafter
2413   {\romannumeral0\xint_cleanupzeros_andstop #2}%
2414 }%

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

2415 \def\XINT_div_final_da #1%
2416 {%
2417   \ifnum #1>\xint_c_ix
2418     \expandafter\XINT_div_final_dP
2419   \else
2420     \xint_afterfi
2421     {\ifnum #1<\xint_c_
2422       \expandafter\XINT_div_final_dN
2423       \else
2424       \expandafter\XINT_div_final_db
2425       \fi }%
2426   \fi
2427 }%
2428 \def\XINT_div_final_dN #1%
2429 {%
2430   \expandafter\XINT_div_final_dP\the\numexpr #1-\xint_c_i\relax
2431 }%
2432 \def\XINT_div_final_dP #1#2#3#4#5% q,A,Q,L,B (puis c)
2433 {%
2434   \expandafter \XINT_div_final_f \expandafter
2435   {\romannumeral0\xintiisub {#2}%
2436    {\romannumeral0\XINT_mul_M {#1}#5\Z\Z\Z\Z }}%
2437   {\romannumeral0\XINT_add_A 0{#1#2#3#4#5\W\X\Y\Z #3\W\X\Y\Z }}%
2438 }%
2439 \def\XINT_div_final_db #1#2#3#4#5% q,A,Q,L,B (puis c)
2440 {%
2441   \expandafter\XINT_div_final_dc\expandafter
2442   {\romannumeral0\xintiisub {#2}%
2443    {\romannumeral0\XINT_mul_M {#1}#5\Z\Z\Z\Z }}%
2444   {#1}{#2}{#3}{#4}{#5}%
2445 }%
2446 \def\XINT_div_final_dc #1#2%
2447 {%
2448   \ifnum\XINT_Sgn{#1}<\xint_c_
2449     \xint_afterfi
2450     {\expandafter\XINT_div_final_dP\the\numexpr #2-\xint_c_i\relax}%
2451   \else \xint_afterfi {\XINT_div_final_e {#1}#2}%
2452   \fi
2453 }%
2454 \def\XINT_div_final_e #1#2#3#4#5#6% A final, q, trash, Q, L, B

```

34 Package *xint* implementation

```

2455 {%
2456   \XINT_div_final_f {#1}%
2457   {\romannumeral0\XINT_add_A 0}{#2000\W\X\Y\Z #4\W\X\Y\Z }%
2458 }%
2459 \def\XINT_div_final_f #1#2#3% R,Q \'a d\'evelopper,c
2460 {%
2461   \ifcase \XINT_Sgn {#1}
2462     \xint_afterfi {\XINT_div_final_end {0}}%
2463   \or
2464     \xint_afterfi {\expandafter\XINT_div_final_end\expandafter
2465                   {\romannumeral0\XINT_dsh_checksngx #3\Z {#1}}%
2466                   }%
2467   \fi
2468   {#2}%
2469 }%
2470 \def\XINT_div_final_end #1#2%
2471 {%
2472   \expandafter\space\expandafter {#2}{#1}%
2473 }%

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

2474 \def\XINT_div_body_a #1%
2475 {%
2476   \XINT_div_body_b #1\Z {#1}%
2477 }%
2478 \def\XINT_div_body_b #1#2#3#4#5#6#7#8#9\Z
2479 {%
2480   \XINT_div_body_c {#1#2#3#4#5#6#7#8}%
2481 }%

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

2482 \def\XINT_div_body_c #1#2#3%
2483 {%
2484   \XINT_div_body_d {#3}{#2\Z {#1}{#3}%
2485 }%
2486 \def\XINT_div_body_d #1#2#3#4#5#6%
2487 {%
2488   \ifnum #1 >\xint_c_
2489     \expandafter\XINT_div_body_d
2490     \expandafter{\the\numexpr #1-\xint_c_iv\expandafter }%
2491   \else
2492     \expandafter\XINT_div_body_e
2493   \fi
2494   {#6#5#4#3#2}%
2495 }%
2496 \def\XINT_div_body_e #1#2\Z #3%
2497 {%

```

34 Package *xint* implementation

```

2498 \XINT_div_body_f {#3}{#1}{#2}%
2499 }%

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

2500 \def\XINT_div_body_f #1#2#3#4#5#6#7#8%
2501 {%
2502 \expandafter\XINT_div_body_gg
2503 \the\numexpr (#1+(#5+\xint_c_i)/\xint_c_ii)/(#5+\xint_c_i)+99999\relax
2504 {#8}{#2}{#8}{#4}{#5}{#3}{#6}{#7}{#8}%
2505 }%

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

2506 \def\XINT_div_body_gg #1#2#3#4#5#6%
2507 {%
2508 \xint_UDzerofork
2509 #2\dummy \XINT_div_body_gk
2510 0\dummy {\XINT_div_body_ggk #2}%
2511 \krof
2512 {#3#4#5#6}%
2513 }%
2514 \def\XINT_div_body_gk #1#2#3%
2515 {%
2516 \expandafter\XINT_div_body_h
2517 \romannumeral0\XINT_div_sub_xpxp
2518 {\romannumeral0\XINT_mul_Mr {#1}#2\Z\Z\Z\Z }{#3}\Z {#1}%
2519 }%
2520 \def\XINT_div_body_ggk #1#2#3%
2521 {%
2522 \expandafter \XINT_div_body_gggk \expandafter
2523 {\romannumeral0\XINT_mul_Mr {#1}0000#3\Z\Z\Z\Z }%
2524 {\romannumeral0\XINT_mul_Mr {#2}#3\Z\Z\Z\Z }%
2525 {#1#2}%
2526 }%
2527 \def\XINT_div_body_gggk #1#2#3#4%
2528 {%
2529 \expandafter\XINT_div_body_h
2530 \romannumeral0\XINT_div_sub_xpxp
2531 {\romannumeral0\expandafter\XINT_mul_Ar
2532 \expandafter0\expandafter{\expandafter}#2\W\X\Y\Z #1\W\X\Y\Z }%
2533 {#4}\Z {#3}%
2534 }%

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

2535 \def\XINT_div_body_h #1#2#3#4#5#6#7#8#9\Z
2536 {%
2537 \ifnum #1#2#3#4>\xint_c_
2538 \xint_afterfi{\XINT_div_body_i {#1#2#3#4#5#6#7#8}}%

```

```

2539 \else
2540 \expandafter\XINT_div_body_k
2541 \fi
2542 {#1#2#3#4#5#6#7#8#9}%
2543 }%
2544 \def\XINT_div_body_k #1#2#3%
2545 {%
2546 \XINT_div_body_l {#1}{#2}%
2547 }%

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

2548 \def\XINT_div_body_i #1#2#3#4#5#6%
2549 {%
2550 \expandafter\XINT_div_body_j
2551 \expandafter{\the\numexpr (#1+(#6+1)/2)/(#6+1)-1}%
2552 {#2}{#3}{#4}{#5}{#6}%
2553 }%
2554 \def\XINT_div_body_j #1#2#3#4%
2555 {%
2556 \expandafter \XINT_div_body_l \expandafter
2557 {\romannumeral0\XINT_div_sub_xpxp
2558 {\romannumeral0\XINT_mul_Mr {#1}#4\Z\Z\Z\Z }{\xintReverseOrder{#2}}}%
2559 {#3+#1}%
2560 }%

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

2561 \def\XINT_div_body_l #1#2#3#4#5#6#7%
2562 {%
2563 \expandafter\XINT_div_body_m
2564 \the\numexpr \xint_c_x^viii+#2\relax {#6}{#3}{#7}{#1#5}{#4}%
2565 }%

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

2566 \def\XINT_div_body_m 1#1#2#3#4#5#6#7#8%
2567 {%
2568 \ifnum #1#2#3#4>\xint_c_
2569 \xint_afterfi {\XINT_div_body_n {#8#7#6#5#4#3#2#1}}%
2570 \else
2571 \xint_afterfi {\XINT_div_body_n {#8#7#6#5}}%
2572 \fi
2573 }%

q renversé, Q, K, L, A', x, B, c

2574 \def\XINT_div_body_n #1#2%
2575 {%
2576 \expandafter\XINT_div_body_o\expandafter
2577 {\romannumeral0\XINT_addr_A 0{ }#1\W\X\Y\Z #2\W\X\Y\Z }%
2578 }%

```

```

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

2579 \def\XINT_div_body_o #1#2#3#4%
2580 {%
2581     \XINT_div_body_p {#3}{#2}{}#4\Z {#1}%
2582 }%

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

2583 \def\XINT_div_body_p #1#2#3#4#5#6#7%
2584 {%
2585     \ifnum #1 > #2
2586         \xint_afterfi
2587         {\ifnum #4#5#6#7 > \xint_c_
2588             \expandafter\XINT_div_body_q
2589             \else
2590             \expandafter\XINT_div_body_repeatp
2591             \fi }%
2592     \else
2593         \expandafter\XINT_div_gotofinal_a
2594     \fi
2595     {#1}{#2}{#3}#4#5#6#7%
2596 }%

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

2597 \def\XINT_div_body_repeatp #1#2#3#4#5#6#7%
2598 {%
2599     \expandafter\XINT_div_body_p\expandafter{\the\numexpr #1-4}{#2}{0000#3}%
2600 }%

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

2601 \def\XINT_div_body_q #1#2#3#4\Z #5#6%
2602 {%
2603     \XINT_div_body_b #4\Z {#4}{#2}{#6}{#3#5}{#1}%
2604 }%

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

2605 \def\XINT_div_gotofinal_a #1#2#3#4\Z %
2606 {%
2607     \XINT_div_gotofinal_b #3\Z {#4}{#1}%
2608 }%
2609 \def\XINT_div_gotofinal_b 0000#1\Z #2#3#4#5%

```

```

2610 {%
2611   \XINT_div_final_a {#2}{#3}{#5}{#1#4}{#3}%
2612 }%

```

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*.

```

2613 \def\XINT_div_sub_xpxp #1%
2614 {%
2615   \expandafter \XINT_div_sub_xpxp_a \expandafter{#1}%
2616 }%
2617 \def\XINT_div_sub_xpxp_a #1#2%
2618 {%
2619   \expandafter\expandafter\expandafter\XINT_div_sub_xpxp_b
2620   #2\W\X\Y\Z #1\W\X\Y\Z
2621 }%
2622 \def\XINT_div_sub_xpxp_b
2623 {%
2624   \XINT_div_sub_A 1}%
2625 }%
2626 \def\XINT_div_sub_A #1#2#3#4#5#6%
2627 {%
2628   \xint_gob_til_W #3\xint_div_sub_az\W
2629   \XINT_div_sub_B #1{#3#4#5#6}{#2}%
2630 }%
2631 \def\XINT_div_sub_B #1#2#3#4\W\X\Y\Z #5#6#7#8%
2632 {%
2633   \xint_gob_til_W #5\xint_div_sub_bz\W
2634   \XINT_div_sub_onestep #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
2635 }%
2636 \def\XINT_div_sub_onestep #1#2#3#4#5#6%
2637 {%
2638   \expandafter\XINT_div_sub_backtoA
2639   \the\numexpr 11#5#4#3#2-#6+#1-\xint_c_i.%
2640 }%
2641 \def\XINT_div_sub_backtoA #1#2#3.#4%
2642 {%
2643   \XINT_div_sub_A #2{#3#4}%
2644 }%
2645 \def\xint_div_sub_bz\W\XINT_div_sub_onestep #1#2#3#4#5#6#7%
2646 {%
2647   \xint_UDzerofork
2648   #1\dummy \XINT_div_sub_C %
2649   0\dummy \XINT_div_sub_D % pas de retenue
2650   \krof
2651   {#7}#2#3#4#5%

```

```

2652 }%
2653 \def\XINT_div_sub_D #1#2\W\X\Y\Z
2654 {%
2655     \expandafter\space
2656     \romannumeral0%
2657     \XINT_rord_main {}#2%
2658     \xint_relax
2659     \xint_bye\xint_bye\xint_bye\xint_bye
2660     \xint_bye\xint_bye\xint_bye\xint_bye
2661     \xint_relax
2662     #1%
2663 }%
2664 \def\XINT_div_sub_C #1#2#3#4#5%
2665 {%
2666     \xint_gob_til_W #2\xint_div_sub_cz\W
2667     \XINT_div_sub_AC_onestep {#5#4#3#2}{#1}%
2668 }%
2669 \def\XINT_div_sub_AC_onestep #1%
2670 {%
2671     \expandafter\XINT_div_sub_backtoC\the\numexpr 11#1-\xint_c_i.%
2672 }%
2673 \def\XINT_div_sub_backtoC #1#2#3.#4%
2674 {%
2675     \XINT_div_sub_AC_checkcarry #2{#3#4}% la retenue va \^etre examin\'ee
2676 }%
2677 \def\XINT_div_sub_AC_checkcarry #1%
2678 {%
2679     \xint_gob_til_one #1\xint_div_sub_AC_nocarry 1\XINT_div_sub_C
2680 }%
2681 \def\xint_div_sub_AC_nocarry 1\XINT_div_sub_C #1#2\W\X\Y\Z
2682 {%
2683     \expandafter\space
2684     \romannumeral0%
2685     \XINT_rord_main {}#2%
2686     \xint_relax
2687     \xint_bye\xint_bye\xint_bye\xint_bye
2688     \xint_bye\xint_bye\xint_bye\xint_bye
2689     \xint_relax
2690     #1%
2691 }%
2692 \def\xint_div_sub_cz\W\XINT_div_sub_AC_onestep #1#2{ #2}%
2693 \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.

34.51 \xintFDg

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

```

2694 \def\xintiifDg {\romannumeral0\xintiifdg }%
2695 \def\xintiifdg #1%
2696 {%
2697   \expandafter\XINT_fdg \romannumeral-‘0#1\W\Z
2698 }%
2699 \def\xintFDg {\romannumeral0\xintfdg }%
2700 \def\xintfdg #1%
2701 {%
2702   \expandafter\XINT_fdg \romannumeral0\xintnum{#1}\W\Z
2703 }%
2704 \def\XINT_FDg #1{\romannumeral0\XINT_fdg #1\W\Z }%
2705 \def\XINT_fdg #1#2#3\Z
2706 {%
2707   \xint_UDzerominusfork
2708   #1-\dummy { 0}% zero
2709   0#1\dummy { #2}% negative
2710   0-\dummy { #1}% positive
2711   \krof
2712 }%

```

34.52 \xintLDg

LAST DIGIT. Simplified in 1.05. And prepared for extension by xintfrac to parse through \xintNum. Release 1.09a adds the \xintnum already here, and this propagates to \xintOdd, etc... 1.09e The \xintiLDg is for defining \xintiiOdd which is used once (currently) elsewhere .

```

2713 \def\xintiLDg {\romannumeral0\xintiildg }%
2714 \def\xintiildg #1%
2715 {%
2716   \expandafter\XINT_ldg\expandafter {\romannumeral-‘0#1}%
2717 }%
2718 \def\xintLDg {\romannumeral0\xintldg }%
2719 \def\xintldg #1%
2720 {%
2721   \expandafter\XINT_ldg\expandafter {\romannumeral0\xintnum{#1}}%
2722 }%
2723 \def\XINT_LDg #1{\romannumeral0\XINT_ldg {#1}}%
2724 \def\XINT_ldg #1%
2725 {%
2726   \expandafter\XINT_ldg_\romannumeral0\xintreverseorder {#1}\Z
2727 }%
2728 \def\XINT_ldg_ #1#2\Z{ #1}%

```

34.53 \xintMON, \xintMMON

MINUS ONE TO THE POWER N and $(-1)^{N-1}$

```

2729 \def\xintiimon {\romannumeral0\xintiimon }%
2730 \def\xintiimon #1%
2731 {%
2732   \ifodd\xintiiLDg {#1}
2733     \xint_afterfi{ -1}%
2734   \else
2735     \xint_afterfi{ 1}%
2736   \fi
2737 }%
2738 \def\xintiiMMON {\romannumeral0\xintiiimon }%
2739 \def\xintiiimon #1%
2740 {%
2741   \ifodd\xintiiLDg {#1}
2742     \xint_afterfi{ 1}%
2743   \else
2744     \xint_afterfi{ -1}%
2745   \fi
2746 }%
2747 \def\xintMON {\romannumeral0\xintmon }%
2748 \def\xintmon #1%
2749 {%
2750   \ifodd\xintLDg {#1}
2751     \xint_afterfi{ -1}%
2752   \else
2753     \xint_afterfi{ 1}%
2754   \fi
2755 }%
2756 \def\xintMMON {\romannumeral0\xintmmon }%
2757 \def\xintmmon #1%
2758 {%
2759   \ifodd\xintLDg {#1}
2760     \xint_afterfi{ 1}%
2761   \else
2762     \xint_afterfi{ -1}%
2763   \fi
2764 }%

```

34.54 \xintOdd

1.05 has \xintiOdd, whereas \xintOdd parses through \xintNum. Inadvertently, 1.09a redefined \xintiLDg so \xintiOdd also parsed through \xintNum. Anyway, having a \xintOdd and a \xintiOdd was silly. Removed in 1.09f

```

2765 \def\xintiOdd {\romannumeral0\xintiiodd }%
2766 \def\xintiiodd #1%

```

```

2767 {%
2768   \ifodd\xintiLDg{#1}
2769     \xint_afterfi{ 1}%
2770   \else
2771     \xint_afterfi{ 0}%
2772   \fi
2773 }%
2774 \def\xintOdd {\romannumeral0\xintodd }%
2775 \def\xintodd #1%
2776 {%
2777   \ifodd\xintLDg{#1}
2778     \xint_afterfi{ 1}%
2779   \else
2780     \xint_afterfi{ 0}%
2781   \fi
2782 }%

```

34.55 \xintDSL

DECIMAL SHIFT LEFT (=MULTIPLICATION PAR 10)

```

2783 \def\xintDSL {\romannumeral0\xintdsl }%
2784 \def\xintdsl #1%
2785 {%
2786   \expandafter\XINT_dsl \romannumeral-'0#1\Z
2787 }%
2788 \def\XINT_DSL #1{\romannumeral0\XINT_dsl #1\Z }%
2789 \def\XINT_dsl #1%
2790 {%
2791   \xint_gob_til_zero #1\xint_dsl_zero 0\XINT_dsl_ #1%
2792 }%
2793 \def\xint_dsl_zero 0\XINT_dsl_ 0#1\Z { 0}%
2794 \def\XINT_dsl_ #1\Z { #10}%

```

34.56 \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

```

2795 \def\xintDSR {\romannumeral0\xintdsr }%
2796 \def\xintdsr #1%
2797 {%
2798   \expandafter\XINT_dsr_a\expandafter {\romannumeral-'0#1}\W\Z
2799 }%
2800 \def\XINT_DSR #1{\romannumeral0\XINT_dsr_a {#1}\W\Z }%
2801 \def\XINT_dsr_a
2802 {%

```

```

2803 \expandafter\XINT_dsr_b\romannumeral0\xintreverseorder
2804 }%
2805 \def\XINT_dsr_b #1#2#3\Z
2806 {%
2807 \xint_gob_til_W #2\xint_dsr_onedigit\W
2808 \xint_gob_til_minus #2\xint_dsr_onedigit-%
2809 \expandafter\XINT_dsr_remove
2810 \romannumeral0\xintreverseorder {#2#3}%
2811 }%
2812 \def\xint_dsr_onedigit #1\xintreverseorder #2{ 0}%
2813 \def\XINT_dsr_remove #1\W { }%

```

34.57 \xintDSH, \xintDSHr

DECIMAL SHIFTS \xintDSH {x}{A}

si $x \leq 0$, fait $A \rightarrow A \cdot 10^{|x|}$. v1.03 corrige l'oversight pour $A=0$ si $x > 0$, et $A \geq 0$, fait $A \rightarrow \text{quo}(A, 10^x)$

si $x > 0$, et $A < 0$, fait $A \rightarrow -\text{quo}(-A, 10^x)$
 (donc pour $x > 0$ c'est comme DSR itéré x fois)
 \xintDSHr donne le 'reste' (si $x \leq 0$ donne zéro).

Release 1.06 now feeds x to a \numexpr first. I will revise the legacy code on another occasion.

```

2814 \def\xintDSHr {\romannumeral0\xintdshr }%
2815 \def\xintdshr #1%
2816 {%
2817 \expandafter\XINT_dshr_checkxpositive \the\numexpr #1\relax\Z
2818 }%
2819 \def\XINT_dshr_checkxpositive #1%
2820 {%
2821 \xint_UDzerominusfork
2822 0#1\dummy \XINT_dshr_xzeroorneg
2823 #1-\dummy \XINT_dshr_xzeroorneg
2824 0-\dummy \XINT_dshr_xpositive
2825 \krof #1%
2826 }%
2827 \def\XINT_dshr_xzeroorneg #1\Z #2{ 0}%
2828 \def\XINT_dshr_xpositive #1\Z
2829 {%
2830 \expandafter\xint_secondoftwo_andstop\romannumeral0\xintdsx {#1}%
2831 }%
2832 \def\xintDSH {\romannumeral0\xintdsh }%
2833 \def\xintdsh #1#2%
2834 {%
2835 \expandafter\xint_dsh\expandafter {\romannumeral-'0#2}{#1}%
2836 }%
2837 \def\xint_dsh #1#2%
2838 {%
2839 \expandafter\XINT_dsh_checksignx \the\numexpr #2\relax\Z {#1}%

```

```

2840 }%
2841 \def\XINT_dsh_checksignx #1%
2842 {%
2843   \xint_UDzerominusfork
2844   #1-\dummy \XINT_dsh_xiszero
2845   0#1\dummy \XINT_dsx_xisNeg_checkA % on passe direct dans DSx
2846   0-\dummy {\XINT_dsh_xisPos #1}%
2847   \krof
2848 }%
2849 \def\XINT_dsh_xiszero #1\Z #2{ #2}%
2850 \def\XINT_dsh_xisPos #1\Z #2%
2851 {%
2852   \expandafter\xint_firstoftwo_andstop
2853   \romannumeral0\XINT_dsx_checksighA #2\Z {#1}% via DSx
2854 }%

```

34.58 \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 $A \rightarrow A \cdot 10^{|x|}$
si $x \geq 0$, et $A \geq 0$, fait $A \rightarrow \{ \text{quo}(A, 10^x) \} \{ \text{rem}(A, 10^x) \}$
si $x \geq 0$, et $A < 0$, d'abord on calcule $\{ \text{quo}(-A, 10^x) \} \{ \text{rem}(-A, 10^x) \}$
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; $40000 = 8 \times 5000$ 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!

```

2855 \def\xintDSx {\romannumeral0\xintdsx }%
2856 \def\xintdsx #1#2%
2857 {%
2858   \expandafter\xint_dsx\expandafter {\romannumeral-'0#2}{#1}%
2859 }%
2860 \def\xint_dsx #1#2%
2861 {%

```

```

2862 \expandafter\XINT_dsx_checksignx \the\numexpr #2\relax\Z {#1}%
2863 }%
2864 \def\XINT_DSx #1#2{\romannumeral0\XINT_dsx_checksignx #1\Z {#2}}%
2865 \def\XINT_dsx #1#2{\XINT_dsx_checksignx #1\Z {#2}}%
2866 \def\XINT_dsx_checksignx #1%
2867 {%
2868 \xint_UDzerominusfork
2869 #1-\dummy \XINT_dsx_xisZero
2870 0#1\dummy \XINT_dsx_xisNeg_checkA
2871 0-\dummy {\XINT_dsx_xisPos #1}%
2872 \krof
2873 }%
2874 \def\XINT_dsx_xisZero #1\Z #2{ {#2}{0}}% attention comme x > 0
2875 \def\XINT_dsx_xisNeg_checkA #1\Z #2%
2876 {%
2877 \XINT_dsx_xisNeg_checkA_ #2\Z {#1}%
2878 }%
2879 \def\XINT_dsx_xisNeg_checkA_ #1#2\Z #3%
2880 {%
2881 \xint_gob_til_zero #1\XINT_dsx_xisNeg_Azero 0%
2882 \XINT_dsx_xisNeg_checkx {#3}{#3}{ }\Z {#1#2}%
2883 }%
2884 \def\XINT_dsx_xisNeg_Azero #1\Z #2{ 0}%
2885 \def\XINT_dsx_xisNeg_checkx #1%
2886 {%
2887 \ifnum #1>999999999
2888 \xint_afterfi
2889 {\xintError:TooBigDecimalShift
2890 \expandafter\space\expandafter 0\xint_gobble_iv }%
2891 \else
2892 \expandafter \XINT_dsx_zeroloop
2893 \fi
2894 }%
2895 \def\XINT_dsx_zeroloop #1#2%
2896 {%
2897 \ifnum #1<9 \XINT_dsx_exita\fi
2898 \expandafter\XINT_dsx_zeroloop\expandafter
2899 {\the\numexpr #1-8}{#2000000000}%
2900 }%
2901 \def\XINT_dsx_exita\fi\expandafter\XINT_dsx_zeroloop
2902 {%
2903 \fi\expandafter\XINT_dsx_exitb
2904 }%
2905 \def\XINT_dsx_exitb #1#2%
2906 {%
2907 \expandafter\expandafter\expandafter
2908 \XINT_dsx_addzeros\csname xint_gobble_\romannumeral -#1\endcsname #2%
2909 }%
2910 \def\XINT_dsx_addzeros #1\Z #2{ #2#1}%

```

34 Package *xint* implementation

```

2911 \def\XINT_dsx_xisPos #1\Z #2%
2912 {%
2913   \XINT_dsx_checksiga #2\Z {#1}%
2914 }%
2915 \def\XINT_dsx_checksiga #1%
2916 {%
2917   \xint_UDzerominusfork
2918   #1-\dummy \XINT_dsx_AisZero
2919   0#1\dummy \XINT_dsx_AisNeg
2920   0-\dummy {\XINT_dsx_AisPos #1}%
2921   \krof
2922 }%
2923 \def\XINT_dsx_AisZero #1\Z #2{ {0}{0}}%
2924 \def\XINT_dsx_AisNeg #1\Z #2%
2925 {%
2926   \expandafter\XINT_dsx_AisNeg_dosplit_andcheckfirst
2927   \romannumeral0\XINT_split_checksizex {#2}{#1}%
2928 }%
2929 \def\XINT_dsx_AisNeg_dosplit_andcheckfirst #1%
2930 {%
2931   \XINT_dsx_AisNeg_checkiffirstempty #1\Z
2932 }%
2933 \def\XINT_dsx_AisNeg_checkiffirstempty #1%
2934 {%
2935   \xint_gob_til_Z #1\XINT_dsx_AisNeg_finish_zero\Z
2936   \XINT_dsx_AisNeg_finish_notzero #1%
2937 }%
2938 \def\XINT_dsx_AisNeg_finish_zero\Z
2939   \XINT_dsx_AisNeg_finish_notzero\Z #1%
2940 {%
2941   \expandafter\XINT_dsx_end
2942   \expandafter {\romannumeral0\XINT_num {-#1}}{0}%
2943 }%
2944 \def\XINT_dsx_AisNeg_finish_notzero #1\Z #2%
2945 {%
2946   \expandafter\XINT_dsx_end
2947   \expandafter {\romannumeral0\XINT_num {#2}}{-#1}%
2948 }%
2949 \def\XINT_dsx_AisPos #1\Z #2%
2950 {%
2951   \expandafter\XINT_dsx_AisPos_finish
2952   \romannumeral0\XINT_split_checksizex {#2}{#1}%
2953 }%
2954 \def\XINT_dsx_AisPos_finish #1#2%
2955 {%
2956   \expandafter\XINT_dsx_end
2957   \expandafter {\romannumeral0\XINT_num {#2}}%
2958   {\romannumeral0\XINT_num {#1}}%
2959 }%

```

```

2960 \def\XINT_dsx_end #1#2%
2961 {%
2962   \expandafter\space\expandafter{#2}{#1}%
2963 }%

```

34.59 \xintDecSplit, \xintDecSplitL, \xintDecSplitR

DECIMAL SPLIT

The macro `\xintDecSplit {x}{A}` first replaces `A` with `|A|` (*) This macro cuts the number into two pieces `L` and `R`. The concatenation `LR` always reproduces `|A|`, and `R` may be empty or have leading zeros. The position of the cut is specified by the first argument `x`. If `x` is zero or positive the cut location is `x` slots to the left of the right end of the number. If `x` becomes equal to or larger than the length of the number then `L` becomes empty. If `x` is negative the location of the cut is `|x|` slots to the right of the left end of the number.

(*) warning: this may change in a future version. Only the behavior for `A` non-negative is guaranteed to remain the same.

v1.05a: `\XINT_split_checksizex` does not compute the length anymore, rather the error will be from a `\numexpr`; but the limit of 999999999 does not make much sense.

v1.06: Improvements in `\XINT_split_fromleft_loop`, `\XINT_split_fromright_loop` and related macros. More readable coding, speed gains. Also, I now feed immediately a `\numexpr` with `x`. Some simplifications should probably be made to the code, which is kept as is for the time being.

1.09e pays attention to the use of `xinttiabs` which acquired in 1.09a the `xintnum` overhead. So `xinttiabs` rather without that overhead.

```

2964 \def\xintDecSplitL {\romannumeral0\xintdecsplitl }%
2965 \def\xintDecSplitR {\romannumeral0\xintdecsplitr }%
2966 \def\xintdecsplitl
2967 {%
2968   \expandafter\xint_firstoftwo_andstop
2969   \romannumeral0\xintdecsplit
2970 }%
2971 \def\xintdecsplitr
2972 {%
2973   \expandafter\xint_secondoftwo_andstop
2974   \romannumeral0\xintdecsplit
2975 }%
2976 \def\xintDecSplit {\romannumeral0\xintdecsplit }%
2977 \def\xintdecsplit #1#2%
2978 {%
2979   \expandafter \xint_split \expandafter
2980   {\romannumeral0\xinttiabs {#2}}{#1}%  fait expansion de A
2981 }%
2982 \def\xint_split #1#2%
2983 {%
2984   \expandafter\XINT_split_checksizex\expandafter{\the\numexpr #2}{#1}%
2985 }%
2986 \def\XINT_split_checksizex #1% 999999999 is anyhow very big, could be reduced

```



```

2987 {%
2988   \ifnum\numexpr\XINT_Abs{#1}>999999999
2989     \xint_afterfi {\xintError:TooBigDecimalSplit\XINT_split_bigx }%
2990   \else
2991     \expandafter\XINT_split_xfork
2992   \fi
2993   #1\Z
2994 }%
2995 \def\XINT_split_bigx #1\Z #2%
2996 {%
2997   \ifcase\XINT_Sgn {#1}
2998   \or \xint_afterfi { {}{#2}}% positive big x
2999   \else
3000     \xint_afterfi { {}{#2}}% negative big x
3001   \fi
3002 }%
3003 \def\XINT_split_xfork #1%
3004 {%
3005   \xint_UDzerominusfork
3006   #1-\dummy \XINT_split_zerosplit
3007   0#1\dummy \XINT_split_fromleft
3008   0-\dummy {\XINT_split_fromright #1}%
3009   \krof
3010 }%
3011 \def\XINT_split_zerosplit #1\Z #2{ {}{#2}}%
3012 \def\XINT_split_fromleft #1\Z #2%
3013 {%
3014   \XINT_split_fromleft_loop {#1}{}#2\W\W\W\W\W\W\W\W\Z
3015 }%
3016 \def\XINT_split_fromleft_loop #1%
3017 {%
3018   \ifnum #1<8 \XINT_split_fromleft_exita\fi
3019   \expandafter\XINT_split_fromleft_loop_perhaps\expandafter
3020   {\the\numexpr #1-8\expandafter}\XINT_split_fromleft_eight
3021 }%
3022 \def\XINT_split_fromleft_eight #1#2#3#4#5#6#7#8#9{#9{#1#2#3#4#5#6#7#8#9}}%
3023 \def\XINT_split_fromleft_loop_perhaps #1#2%
3024 {%
3025   \xint_gob_til_W #2\XINT_split_fromleft_toofar\W
3026   \XINT_split_fromleft_loop {#1}%
3027 }%
3028 \def\XINT_split_fromleft_toofar\W\XINT_split_fromleft_loop #1#2#3\Z
3029 {%
3030   \XINT_split_fromleft_toofar_b #2\Z
3031 }%
3032 \def\XINT_split_fromleft_toofar_b #1\W #2\Z { {}{#1}}%
3033 \def\XINT_split_fromleft_exita\fi
3034   \expandafter\XINT_split_fromleft_loop_perhaps\expandafter #1#2%
3035   {\fi \XINT_split_fromleft_exitb #1}%

```

```

3036 \def\XINT_split_fromleft_exitb\the\numexpr #1-8\expandafter
3037 {%
3038   \csname XINT_split_fromleft_endsplit_\romannumeral #1\endcsname
3039 }%
3040 \def\XINT_split_fromleft_endsplit_ #1#2\W #3\Z { {#1}{#2}}%
3041 \def\XINT_split_fromleft_endsplit_i #1#2%
3042   {\XINT_split_fromleft_checkiftoofar #2{#1#2}}%
3043 \def\XINT_split_fromleft_endsplit_ii #1#2#3%
3044   {\XINT_split_fromleft_checkiftoofar #3{#1#2#3}}%
3045 \def\XINT_split_fromleft_endsplit_iii #1#2#3#4%
3046   {\XINT_split_fromleft_checkiftoofar #4{#1#2#3#4}}%
3047 \def\XINT_split_fromleft_endsplit_iv #1#2#3#4#5%
3048   {\XINT_split_fromleft_checkiftoofar #5{#1#2#3#4#5}}%
3049 \def\XINT_split_fromleft_endsplit_v #1#2#3#4#5#6%
3050   {\XINT_split_fromleft_checkiftoofar #6{#1#2#3#4#5#6}}%
3051 \def\XINT_split_fromleft_endsplit_vi #1#2#3#4#5#6#7%
3052   {\XINT_split_fromleft_checkiftoofar #7{#1#2#3#4#5#6#7}}%
3053 \def\XINT_split_fromleft_endsplit_vii #1#2#3#4#5#6#7#8%
3054   {\XINT_split_fromleft_checkiftoofar #8{#1#2#3#4#5#6#7#8}}%
3055 \def\XINT_split_fromleft_checkiftoofar #1#2#3\W #4\Z
3056 {%
3057   \xint_gob_til_W #1\XINT_split_fromleft_wenttoofar\W
3058   \space {#2}{#3}%
3059 }%
3060 \def\XINT_split_fromleft_wenttoofar\W\space #1%
3061 {%
3062   \XINT_split_fromleft_wenttoofar_b #1\Z
3063 }%
3064 \def\XINT_split_fromleft_wenttoofar_b #1\W #2\Z { {#1}}%
3065 \def\XINT_split_fromright #1\Z #2%
3066 {%
3067   \expandafter \XINT_split_fromright_a \expandafter
3068   {\romannumeral0\xintreverseorder {#2}}{#1}{#2}%
3069 }%
3070 \def\XINT_split_fromright_a #1#2%
3071 {%
3072   \XINT_split_fromright_loop {#2}{#1\W\W\W\W\W\W\W\W\Z
3073 }%
3074 \def\XINT_split_fromright_loop #1%
3075 {%
3076   \ifnum #1<8 \XINT_split_fromright_exita\fi
3077   \expandafter\XINT_split_fromright_loop_perhaps\expandafter
3078   {\the\numexpr #1-8\expandafter }\XINT_split_fromright_eight
3079 }%
3080 \def\XINT_split_fromright_eight #1#2#3#4#5#6#7#8#9{#9{#9#8#7#6#5#4#3#2#1}}%
3081 \def\XINT_split_fromright_loop_perhaps #1#2%
3082 {%
3083   \xint_gob_til_W #2\XINT_split_fromright_toofar\W
3084   \XINT_split_fromright_loop {#1}%

```

```

3085 }%
3086 \def\XINT_split_fromright_toofar\W\XINT_split_fromright_loop #1#2#3\Z { {}}%
3087 \def\XINT_split_fromright_exita\fi
3088     \expandafter\XINT_split_fromright_loop_perhaps\expandafter #1#2%
3089     {\fi \XINT_split_fromright_exitb #1}%
3090 \def\XINT_split_fromright_exitb\the\numexpr #1-8\expandafter
3091 {%
3092     \csname XINT_split_fromright_endsplit_\romannumeral #1\endcsname
3093 }%
3094 \def\XINT_split_fromright_endsplit_ #1#2\W #3\Z #4%
3095 {%
3096     \expandafter\space\expandafter {\romannumeral0\xintreverseorder {#2}}{#1}%
3097 }%
3098 \def\XINT_split_fromright_endsplit_i #1#2%
3099     {\XINT_split_fromright_checkiftoofar #2{#2#1}}%
3100 \def\XINT_split_fromright_endsplit_ii #1#2#3%
3101     {\XINT_split_fromright_checkiftoofar #3{#3#2#1}}%
3102 \def\XINT_split_fromright_endsplit_iii #1#2#3#4%
3103     {\XINT_split_fromright_checkiftoofar #4{#4#3#2#1}}%
3104 \def\XINT_split_fromright_endsplit_iv #1#2#3#4#5%
3105     {\XINT_split_fromright_checkiftoofar #5{#5#4#3#2#1}}%
3106 \def\XINT_split_fromright_endsplit_v #1#2#3#4#5#6%
3107     {\XINT_split_fromright_checkiftoofar #6{#6#5#4#3#2#1}}%
3108 \def\XINT_split_fromright_endsplit_vi #1#2#3#4#5#6#7%
3109     {\XINT_split_fromright_checkiftoofar #7{#7#6#5#4#3#2#1}}%
3110 \def\XINT_split_fromright_endsplit_vii #1#2#3#4#5#6#7#8%
3111     {\XINT_split_fromright_checkiftoofar #8{#8#7#6#5#4#3#2#1}}%
3112 \def\XINT_split_fromright_checkiftoofar #1%
3113 {%
3114     \xint_gob_til_W #1\XINT_split_fromright_wenttoofar\W
3115     \XINT_split_fromright_endsplit_
3116 }%
3117 \def\XINT_split_fromright_wenttoofar\W\XINT_split_fromright_endsplit_ #1\Z #2%
3118     { {}{#2}}%

```

34.60 \xintDouble

v1.08

```

3119 \def\xintDouble {\romannumeral0\xintdouble }%
3120 \def\xintdouble #1%
3121 {%
3122     \expandafter\XINT_dbl\romannumeral-'0#1%
3123     \R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W
3124 }%
3125 \def\XINT_dbl #1%
3126 {%
3127     \xint_UDzerominusfork
3128     #1-\dummy \XINT_dbl_zero

```

```

3129      0#1\dummy \XINT_dbl_neg
3130      0-\dummy {\XINT_dbl_pos #1}%
3131      \krof
3132 }%
3133 \def\XINT_dbl_zero #1\Z \W\W\W\W\W\W\W { 0}%
3134 \def\XINT_dbl_neg
3135      {\expandafter\xint_minus_andstop\romannumeral0\XINT_dbl_pos }%
3136 \def\XINT_dbl_pos
3137 {%
3138      \expandafter\XINT_dbl_a \expandafter{\expandafter}\expandafter 0%
3139      \romannumeral0\XINT_SQ {}%
3140 }%
3141 \def\XINT_dbl_a #1#2#3#4#5#6#7#8#9%
3142 {%
3143      \xint_gob_til_W #9\XINT_dbl_end_a\W
3144      \expandafter\XINT_dbl_b
3145      \the\numexpr \xint_c_x^viii+#2+\xint_c_ii*#9#8#7#6#5#4#3\relax {#1}%
3146 }%
3147 \def\XINT_dbl_b 1#1#2#3#4#5#6#7#8#9%
3148 {%
3149      \XINT_dbl_a {#2#3#4#5#6#7#8#9}{#1}%
3150 }%
3151 \def\XINT_dbl_end_a #1+#2+#3\relax #4%
3152 {%
3153      \expandafter\XINT_dbl_end_b #2#4%
3154 }%
3155 \def\XINT_dbl_end_b #1#2#3#4#5#6#7#8%
3156 {%
3157      \expandafter\space\the\numexpr #1#2#3#4#5#6#7#8\relax
3158 }%

```

34.61 \xintHalf

v1.08

```

3159 \def\xintHalf {\romannumeral0\xinthalft }%
3160 \def\xinthalft #1%
3161 {%
3162      \expandafter\XINT_half\romannumeral-‘0#1%
3163      \R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W
3164 }%
3165 \def\XINT_half #1%
3166 {%
3167      \xint_UDzerominusfork
3168      #1-\dummy \XINT_half_zero
3169      0#1\dummy \XINT_half_neg
3170      0-\dummy {\XINT_half_pos #1}%
3171      \krof
3172 }%

```

```

3173 \def\XINT_half_zero #1\Z \W\W\W\W\W\W { 0}%
3174 \def\XINT_half_neg {\expandafter\XINT_opp\romannumeral0\XINT_half_pos }%
3175 \def\XINT_half_pos {\expandafter\XINT_half_a\romannumeral0\XINT_SQ {}}%
3176 \def\XINT_half_a #1#2#3#4#5#6#7#8%
3177 {%
3178   \xint_gob_til_W #8\XINT_half_dont\W
3179   \expandafter\XINT_half_b
3180   \the\numexpr \xint_c_x^viii+\xint_c_v*#7#6#5#4#3#2#1\relax #8%
3181 }%
3182 \def\XINT_half_dont\W\expandafter\XINT_half_b
3183   \the\numexpr \xint_c_x^viii+\xint_c_v*#1#2#3#4#5#6#7\relax \W\W\W\W\W\W
3184 {%
3185   \expandafter\space
3186   \the\numexpr (#1#2#3#4#5#6#7+\xint_c_i)/\xint_c_ii-\xint_c_i \relax
3187 }%
3188 \def\XINT_half_b 1#1#2#3#4#5#6#7#8%
3189 {%
3190   \XINT_half_c {#2#3#4#5#6#7}{#1}%
3191 }%
3192 \def\XINT_half_c #1#2#3#4#5#6#7#8#9%
3193 {%
3194   \xint_gob_til_W #3\XINT_half_end_a #2\W
3195   \expandafter\XINT_half_d
3196   \the\numexpr \xint_c_x^viii+\xint_c_v*#9#8#7#6#5#4#3+#2\relax {#1}%
3197 }%
3198 \def\XINT_half_d 1#1#2#3#4#5#6#7#8#9%
3199 {%
3200   \XINT_half_c {#2#3#4#5#6#7#8#9}{#1}%
3201 }%
3202 \def\XINT_half_end_a #1\W #2\relax #3%
3203 {%
3204   \xint_gob_til_zero #1\XINT_half_end_b 0\space #1#3%
3205 }%
3206 \def\XINT_half_end_b 0\space 0#1#2#3#4#5#6#7%
3207 {%
3208   \expandafter\space\the\numexpr #1#2#3#4#5#6#7\relax
3209 }%

```

34.62 \xintDec

v1.08

```

3210 \def\xintDec {\romannumeral0\xintdec }%
3211 \def\xintdec #1%
3212 {%
3213   \expandafter\XINT_dec\romannumeral-'0#1%
3214   \R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W\W
3215 }%
3216 \def\XINT_dec #1%

```

```

3217 {%
3218   \xint_UDzerominusfork
3219   #1-\dummy \XINT_dec_zero
3220   0#1\dummy \XINT_dec_neg
3221   0-\dummy {\XINT_dec_pos #1}%
3222   \krof
3223 }%
3224 \def\XINT_dec_zero #1\W\W\W\W\W\W\W { -1}%
3225 \def\XINT_dec_neg
3226   {\expandafter\xint_minus_andstop\romannumeral0\XINT_inc_pos }%
3227 \def\XINT_dec_pos
3228 {%
3229   \expandafter\XINT_dec_a \expandafter{\expandafter}%
3230   \romannumeral0\XINT_0Q {}%
3231 }%
3232 \def\XINT_dec_a #1#2#3#4#5#6#7#8#9%
3233 {%
3234   \expandafter\XINT_dec_b
3235   \the\numexpr 11#9#8#7#6#5#4#3#2-\xint_c_i\relax {#1}%
3236 }%
3237 \def\XINT_dec_b 1#1%
3238 {%
3239   \xint_gob_til_one #1\XINT_dec_A 1\XINT_dec_c
3240 }%
3241 \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}}%
3242 \def\XINT_dec_A 1\XINT_dec_c #1#2#3#4#5#6#7#8#9%
3243   {\XINT_dec_B {#1#2#3#4#5#6#7#8#9}}%
3244 \def\XINT_dec_B #1#2\W\W\W\W\W\W\W\W
3245 {%
3246   \expandafter\XINT_dec_cleanup
3247   \romannumeral0\XINT_rord_main {}%#2%
3248   \xint_relax
3249   \xint_bye\xint_bye\xint_bye\xint_bye
3250   \xint_bye\xint_bye\xint_bye\xint_bye
3251   \xint_relax
3252   #1%
3253 }%
3254 \def\XINT_dec_cleanup #1#2#3#4#5#6#7#8%
3255 {\expandafter\space\the\numexpr #1#2#3#4#5#6#7#8\relax }%

```

34.63 \xintInc

v1.08

```

3256 \def\xintInc {\romannumeral0\xintinc }%
3257 \def\xintinc #1%
3258 {%
3259   \expandafter\XINT_inc\romannumeral-‘0#1%
3260   \R\R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W\W

```

```

3261 }%
3262 \def\XINT_inc #1%
3263 {%
3264   \xint_UDzerominusfork
3265   #1-\dummy \XINT_inc_zero
3266   0#1\dummy \XINT_inc_neg
3267   0-\dummy {\XINT_inc_pos #1}%
3268   \krof
3269 }%
3270 \def\XINT_inc_zero #1\W\W\W\W\W\W\W\W { 1}%
3271 \def\XINT_inc_neg {\expandafter\XINT_opp\romannumeral0\XINT_dec_pos }%
3272 \def\XINT_inc_pos
3273 {%
3274   \expandafter\XINT_inc_a \expandafter{\expandafter}%
3275   \romannumeral0\XINT_OQ }%
3276 }%
3277 \def\XINT_inc_a #1#2#3#4#5#6#7#8#9%
3278 {%
3279   \xint_gob_til_W #9\XINT_inc_end\W
3280   \expandafter\XINT_inc_b
3281   \the\numexpr 10#9#8#7#6#5#4#3#2+\xint_c_i\relax {#1}%
3282 }%
3283 \def\XINT_inc_b 1#1%
3284 {%
3285   \xint_gob_til_zero #1\XINT_inc_A 0\XINT_inc_c
3286 }%
3287 \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}}%
3288 \def\XINT_inc_A 0\XINT_inc_c #1#2#3#4#5#6#7#8#9%
3289   {\XINT_dec_B {#1#2#3#4#5#6#7#8#9}}%
3290 \def\XINT_inc_end\W #1\relax #2{ 1#2}%

```

34.64 \xintiSqrt, \xintiSquareRoot

v1.08. 1.09a uses \xintnum. Very embarrassing to discover at the time of 1.09e that \xintiSqrt {0} was buggy!

Some overhead was added inadvertently in 1.09a to inner routines when \xintiquo and \xintidivision were promoted to use \xintnum. Reverted in 1.09f.

```

3291 \def\XINT_dsx_addzerosnofuss #1{\XINT_dsx_zeroloop {#1}}{Z }%
3292 \def\xintiSqrt {\romannumeral0\xintisqrt }%
3293 \def\xintisqrt
3294   {\expandafter\XINT_sqrt_post\romannumeral0\xintisquareroot }%
3295 \def\XINT_sqrt_post #1#2{\XINT_dec_pos #1\R\R\R\R\R\R\R\R\Z
3296   \W\W\W\W\W\W\W\W }%
3297 \def\xintiSquareRoot {\romannumeral0\xintisquareroot }%
3298 \def\xintisquareroot #1%
3299   {\expandafter\XINT_sqrt_checkin\romannumeral0\xintnum{#1}\Z}%
3300 \def\XINT_sqrt_checkin #1%
3301 {%

```

```

3302 \xint_UDzerominusfork
3303 #1-\dummy \XINT_sqrt_iszero
3304 0#1\dummy \XINT_sqrt_isneg
3305 0-\dummy {\XINT_sqrt #1}%
3306 \krof
3307 }%
3308 \def\XINT_sqrt_iszero #1\Z { 1.}% 1.09e was wrong from inception in 1.08 :-(
3309 \def\XINT_sqrt_isneg #1\Z {\xintError:RootOfNegative\space 1.}%
3310 \def\XINT_sqrt #1\Z
3311 {%
3312 \expandafter\XINT_sqrt_start\expandafter
3313 {\romannumeral0\xintlength {#1}}{#1}%
3314 }%
3315 \def\XINT_sqrt_start #1%
3316 {%
3317 \ifnum #1<\xint_c_x
3318 \expandafter\XINT_sqrt_small_a
3319 \else
3320 \expandafter\XINT_sqrt_big_a
3321 \fi
3322 {#1}%
3323 }%
3324 \def\XINT_sqrt_small_a #1{\XINT_sqrt_a {#1}\XINT_sqrt_small_d }%
3325 \def\XINT_sqrt_big_a #1{\XINT_sqrt_a {#1}\XINT_sqrt_big_d }%
3326 \def\XINT_sqrt_a #1%
3327 {%
3328 \ifodd #1
3329 \expandafter\XINT_sqrt_bB
3330 \else
3331 \expandafter\XINT_sqrt_bA
3332 \fi
3333 {#1}%
3334 }%
3335 \def\XINT_sqrt_bA #1#2#3%
3336 {%
3337 \XINT_sqrt_bA_b #3\Z #2{#1}{#3}%
3338 }%
3339 \def\XINT_sqrt_bA_b #1#2#3\Z
3340 {%
3341 \XINT_sqrt_c {#1#2}%
3342 }%
3343 \def\XINT_sqrt_bB #1#2#3%
3344 {%
3345 \XINT_sqrt_bB_b #3\Z #2{#1}{#3}%
3346 }%
3347 \def\XINT_sqrt_bB_b #1#2\Z
3348 {%
3349 \XINT_sqrt_c #1%
3350 }%

```



```

3351 \def\XINT_sqrt_c #1#2%
3352 {%
3353   \expandafter #2%
3354   \ifcase #1
3355     \or 2\or 2\or 2\or 3\or 3\or 3\or 3\or 3\or 3\or %3+5
3356     4\or 4\or 4\or 4\or 4\or 4\or 4\or %7
3357     5\or 5\or 5\or 5\or 5\or 5\or 5\or 5\or 5\or %9
3358     6\or 6\or 6\or 6\or 6\or 6\or 6\or 6\or 6\or 6\or %11
3359     7\or 7\or 7\or 7\or 7\or 7\or 7\or 7\or 7\or 7\or 7\or %13
3360     8\or 8\or 8\or 8\or 8\or 8\or 8\or
3361     8\or 8\or 8\or 8\or 8\or 8\or 8\or 8\or %15
3362     9\or 9\or 9\or 9\or 9\or 9\or 9\or 9\or
3363     9\or 9\or 9\or 9\or 9\or 9\or 9\or 9\or 9\or %17
3364     10\or 10\or 10\or 10\or 10\or 10\or 10\or 10\or 10\or
3365     10\or 10\or 10\or 10\or 10\or 10\or 10\or 10\or 10\or 10\or\fi %19
3366}%
3367 \def\XINT_sqrt_small_d #1\or #2\fi #3%
3368 {%
3369   \fi
3370   \expandafter\XINT_sqrt_small_de
3371   \ifcase \numexpr #3/\xint_c_ii-\xint_c_i\relax
3372     {}%
3373   \or
3374     0%
3375   \or
3376     {00}%
3377   \or
3378     {000}%
3379   \or
3380     {0000}%
3381   \or
3382   \fi {#1}%
3383}%
3384 \def\XINT_sqrt_small_de #1\or #2\fi #3%
3385 {%
3386   \fi\XINT_sqrt_small_e {#3#1}%
3387}%
3388 \def\XINT_sqrt_small_e #1#2%
3389 {%
3390   \expandafter\XINT_sqrt_small_f\expandafter {\the\numexpr #1*#1-#2}{#1}%
3391}%
3392 \def\XINT_sqrt_small_f #1#2%
3393 {%
3394   \expandafter\XINT_sqrt_small_g\expandafter
3395   {\the\numexpr ((#1+#2)/(\xint_c_ii*#2))-\xint_c_i}{#1}{#2}%
3396}%
3397 \def\XINT_sqrt_small_g #1%
3398 {%
3399   \ifnum #1>\xint_c_

```

```

3400     \expandafter\XINT_sqrt_small_h
3401     \else
3402     \expandafter\XINT_sqrt_small_end
3403     \fi
3404     {#1}%
3405 }%
3406 \def\XINT_sqrt_small_h #1#2#3%
3407 {%
3408     \expandafter\XINT_sqrt_small_f\expandafter
3409     {\the\numexpr #2-\xint_c_ii*#1*#3+#1*#1\expandafter}\expandafter
3410     {\the\numexpr #3-#1}%
3411 }%
3412 \def\XINT_sqrt_small_end #1#2#3{ {#3}{#2}}%
3413 \def\XINT_sqrt_big_d #1\or #2\fi #3%
3414 {%
3415     \fi
3416     \ifodd #3
3417     \xint_afterfi{\expandafter\XINT_sqrt_big_eB}%
3418     \else
3419     \xint_afterfi{\expandafter\XINT_sqrt_big_eA}%
3420     \fi
3421     \expandafter{\the\numexpr #3/\xint_c_ii }{#1}%
3422 }%
3423 \def\XINT_sqrt_big_eA #1#2#3%
3424 {%
3425     \XINT_sqrt_big_eA_a #3\Z {#2}{#1}{#3}%
3426 }%
3427 \def\XINT_sqrt_big_eA_a #1#2#3#4#5#6#7#8#9\Z
3428 {%
3429     \XINT_sqrt_big_eA_b {#1#2#3#4#5#6#7#8}%
3430 }%
3431 \def\XINT_sqrt_big_eA_b #1#2%
3432 {%
3433     \expandafter\XINT_sqrt_big_f
3434     \romannumeral0\XINT_sqrt_small_e {#2000}{#1}{#1}%
3435 }%
3436 \def\XINT_sqrt_big_eB #1#2#3%
3437 {%
3438     \XINT_sqrt_big_eB_a #3\Z {#2}{#1}{#3}%
3439 }%
3440 \def\XINT_sqrt_big_eB_a #1#2#3#4#5#6#7#8#9%
3441 {%
3442     \XINT_sqrt_big_eB_b {#1#2#3#4#5#6#7#8#9}%
3443 }%
3444 \def\XINT_sqrt_big_eB_b #1#2\Z #3%
3445 {%
3446     \expandafter\XINT_sqrt_big_f
3447     \romannumeral0\XINT_sqrt_small_e {#30000}{#1}{#1}%
3448 }%

```

```

3449 \def\XINT_sqrt_big_f #1#2#3#4%
3450 {%
3451   \expandafter\XINT_sqrt_big_f_a\expandafter
3452   {\the\numexpr #2+#3\expandafter}\expandafter
3453   {\romannumeral0\XINT_dsx_addzerosnofuss
3454     {\numexpr #4-\xint_c_iv\relax}{#1}}{#4}%
3455 }%
3456 \def\XINT_sqrt_big_f_a #1#2#3#4%
3457 {%
3458   \expandafter\XINT_sqrt_big_g\expandafter
3459   {\romannumeral0\xintiisub
3460     {\XINT_dsx_addzerosnofuss
3461       {\numexpr \xint_c_ii*#3-\xint_c_viii\relax}{#1}}{#4}}%
3462   {#2}{#3}%
3463 }%
3464 \def\XINT_sqrt_big_g #1#2%
3465 {%
3466   \expandafter\XINT_sqrt_big_j
3467   \romannumeral0\xintiidivision{#1}
3468   {\romannumeral0\XINT_dbl_pos #2\R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W }{#2}%
3469 }%
3470 \def\XINT_sqrt_big_j #1%
3471 {%
3472   \ifcase\XINT_Sgn {#1}
3473     \expandafter \XINT_sqrt_big_end
3474   \or \expandafter \XINT_sqrt_big_k
3475   \fi {#1}%
3476 }%
3477 \def\XINT_sqrt_big_k #1#2#3%
3478 {%
3479   \expandafter\XINT_sqrt_big_l\expandafter
3480   {\romannumeral0\xintiisub {#3}{#1}}%
3481   {\romannumeral0\xintiiadd {#2}{\xintiiSqr {#1}}}%
3482 }%
3483 \def\XINT_sqrt_big_l #1#2%
3484 {%
3485   \expandafter\XINT_sqrt_big_g\expandafter
3486   {#2}{#1}%
3487 }%
3488 \def\XINT_sqrt_big_end #1#2#3#4{ {#3}{#2}}%
3489 \let\XINT_tmpa\relax \let\XINT_tmpb\relax \let\XINT_tmpc\relax
3490 \XINT_restorecatcodes_endinput%

```

35 Package *xintbinhex* implementation

The commenting is currently (2013/11/28) very sparse.

Contents

.1	Catcodes, ε -TeX and reload detection ..	242	.7	\xintHexToDec	249
.2	Confirmation of xint loading	243	.8	\xintBinToDec	251
.3	Catcodes	243	.9	\xintBinToHex	253
.4	Package identification	244	.10	\xintHexToBin	254
.5	Constants, etc... ..	244	.11	\xintCHexToBin	255
.6	\xintDecToHex, \xintDecToBin	246			

35.1 Catcodes, ε -TeX and reload detection

The code for reload detection is copied from HEIKO OBERDIEK's packages, and adapted here to check for previous loading of the master **xint** package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2  \catcode13=5      % ^^M
3  \endlinechar=13 %
4  \catcode123=1     % {
5  \catcode125=2     % }
6  \catcode64=11     % @
7  \catcode35=6      % #
8  \catcode44=12     % ,
9  \catcode45=12     % -
10 \catcode46=12     % .
11 \catcode58=12     % :
12 \def\space { }%
13 \let\z\endgroup
14 \expandafter\let\expandafter\x\csname ver@xintbinhex.sty\endcsname
15 \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
16 \expandafter
17   \ifx\csname PackageInfo\endcsname\relax
18     \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
19   \else
20     \def\y#1#2{\PackageInfo{#1}{#2}}%
21   \fi
22 \expandafter
23 \ifx\csname numexpr\endcsname\relax
24   \y{xintbinhex}{numexpr not available, aborting input}%
25   \aftergroup\endinput
26 \else
27   \ifx\x\relax % plain-TeX, first loading of xintbinhex.sty
28     \ifx\w\relax % but xint.sty not yet loaded.
29       \y{xintbinhex}{Package xint is required}%
30       \y{xintbinhex}{Will try \string\input\space xint.sty}%
31       \def\z{\endgroup\input xint.sty\relax}%
32     \fi
33   \else

```

```

34 \def\empty {}%
35 \ifx\x\empty % LaTeX, first loading,
36 % variable is initialized, but \ProvidesPackage not yet seen
37 \ifx\w\relax % xint.sty not yet loaded.
38 \y{xintbinhex}{Package xint is required}%
39 \y{xintbinhex}{Will try \string\RequirePackage{xint}}%
40 \def\z{\endgroup\RequirePackage{xint}}%
41 \fi
42 \else
43 \y{xintbinhex}{I was already loaded, aborting input}%
44 \aftergroup\endinput
45 \fi
46 \fi
47 \fi
48 \z%
```

35.2 Confirmation of *xint* loading

```

49 \begingroup\catcode61\catcode48\catcode32=10\relax%
50 \catcode13=5 % ^^M
51 \endlinechar=13 %
52 \catcode123=1 % {
53 \catcode125=2 % }
54 \catcode64=11 % @
55 \catcode35=6 % #
56 \catcode44=12 % ,
57 \catcode45=12 % -
58 \catcode46=12 % .
59 \catcode58=12 % :
60 \ifdefined\PackageInfo
61 \def\y#1#2{\PackageInfo{#1}{#2}}%
62 \else
63 \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
64 \fi
65 \def\empty {}%
66 \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
67 \ifx\w\relax % Plain TeX, user gave a file name at the prompt
68 \y{xintbinhex}{Loading of package xint failed, aborting input}%
69 \aftergroup\endinput
70 \fi
71 \ifx\w\empty % LaTeX, user gave a file name at the prompt
72 \y{xintbinhex}{Loading of package xint failed, aborting input}%
73 \aftergroup\endinput
74 \fi
75 \endgroup%
```

35.3 Catcodes

```

76 \XINTsetupcatcodes%
```

35.4 Package identification

```

77 \XINT_providespackage
78 \ProvidesPackage{xintbinhex}%
79 [2013/11/28 v1.09h Expandable binary and hexadecimal conversions (jfb)]%

```

35.5 Constants, etc...

v1.08

```

80 \chardef\xint_c_xvi          16
81 \chardef\xint_c_ii^v        32
82 \chardef\xint_c_ii^vi       64
83 \chardef\xint_c_ii^vii     128
84 \mathchardef\xint_c_ii^viii 256
85 \mathchardef\xint_c_ii^xii 4096
86 \newcount\xint_c_ii^xv \xint_c_ii^xv 32768
87 \newcount\xint_c_ii^xvi \xint_c_ii^xvi 65536
88 \newcount\xint_c_x^v \xint_c_x^v 100000
89 \newcount\xint_c_x^ix \xint_c_x^ix 1000000000
90 \def\XINT_tmpa #1{%
91   \expandafter\edef\csname XINT_sdt#1\endcsname
92   {\ifcase #1 0\or 1\or 2\or 3\or 4\or 5\or 6\or 7\or
93     8\or 9\or A\or B\or C\or D\or E\or F\fi}}%
94 \xintApplyInline\XINT_tmpa
95   {{0}{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}{12}{13}{14}{15}}%
96 \def\XINT_tmpa #1{%
97   \expandafter\edef\csname XINT_sdtb_#1\endcsname
98   {\ifcase #1
99     0000\or 0001\or 0010\or 0011\or 0100\or 0101\or 0110\or 0111\or
100    1000\or 1001\or 1010\or 1011\or 1100\or 1101\or 1110\or 1111\fi}}%
101 \xintApplyInline\XINT_tmpa
102   {{0}{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}{12}{13}{14}{15}}%
103 \let\XINT_tmpa\relax
104 \expandafter\def\csname XINT_sbtd_0000\endcsname {0}%
105 \expandafter\def\csname XINT_sbtd_0001\endcsname {1}%
106 \expandafter\def\csname XINT_sbtd_0010\endcsname {2}%
107 \expandafter\def\csname XINT_sbtd_0011\endcsname {3}%
108 \expandafter\def\csname XINT_sbtd_0100\endcsname {4}%
109 \expandafter\def\csname XINT_sbtd_0101\endcsname {5}%
110 \expandafter\def\csname XINT_sbtd_0110\endcsname {6}%
111 \expandafter\def\csname XINT_sbtd_0111\endcsname {7}%
112 \expandafter\def\csname XINT_sbtd_1000\endcsname {8}%
113 \expandafter\def\csname XINT_sbtd_1001\endcsname {9}%
114 \expandafter\def\csname XINT_sbtd_1010\endcsname {10}%
115 \expandafter\def\csname XINT_sbtd_1011\endcsname {11}%
116 \expandafter\def\csname XINT_sbtd_1100\endcsname {12}%
117 \expandafter\def\csname XINT_sbtd_1101\endcsname {13}%
118 \expandafter\def\csname XINT_sbtd_1110\endcsname {14}%
119 \expandafter\def\csname XINT_sbtd_1111\endcsname {15}%

```

```

120 \expandafter\let\csname XINT_sbth_0000\expandafter\endcsname
121       \csname XINT_sbtd_0000\endcsname
122 \expandafter\let\csname XINT_sbth_0001\expandafter\endcsname
123       \csname XINT_sbtd_0001\endcsname
124 \expandafter\let\csname XINT_sbth_0010\expandafter\endcsname
125       \csname XINT_sbtd_0010\endcsname
126 \expandafter\let\csname XINT_sbth_0011\expandafter\endcsname
127       \csname XINT_sbtd_0011\endcsname
128 \expandafter\let\csname XINT_sbth_0100\expandafter\endcsname
129       \csname XINT_sbtd_0100\endcsname
130 \expandafter\let\csname XINT_sbth_0101\expandafter\endcsname
131       \csname XINT_sbtd_0101\endcsname
132 \expandafter\let\csname XINT_sbth_0110\expandafter\endcsname
133       \csname XINT_sbtd_0110\endcsname
134 \expandafter\let\csname XINT_sbth_0111\expandafter\endcsname
135       \csname XINT_sbtd_0111\endcsname
136 \expandafter\let\csname XINT_sbth_1000\expandafter\endcsname
137       \csname XINT_sbtd_1000\endcsname
138 \expandafter\let\csname XINT_sbth_1001\expandafter\endcsname
139       \csname XINT_sbtd_1001\endcsname
140 \expandafter\def\csname XINT_sbth_1010\endcsname {A}%
141 \expandafter\def\csname XINT_sbth_1011\endcsname {B}%
142 \expandafter\def\csname XINT_sbth_1100\endcsname {C}%
143 \expandafter\def\csname XINT_sbth_1101\endcsname {D}%
144 \expandafter\def\csname XINT_sbth_1110\endcsname {E}%
145 \expandafter\def\csname XINT_sbth_1111\endcsname {F}%
146 \expandafter\def\csname XINT_shtb_0\endcsname {0000}%
147 \expandafter\def\csname XINT_shtb_1\endcsname {0001}%
148 \expandafter\def\csname XINT_shtb_2\endcsname {0010}%
149 \expandafter\def\csname XINT_shtb_3\endcsname {0011}%
150 \expandafter\def\csname XINT_shtb_4\endcsname {0100}%
151 \expandafter\def\csname XINT_shtb_5\endcsname {0101}%
152 \expandafter\def\csname XINT_shtb_6\endcsname {0110}%
153 \expandafter\def\csname XINT_shtb_7\endcsname {0111}%
154 \expandafter\def\csname XINT_shtb_8\endcsname {1000}%
155 \expandafter\def\csname XINT_shtb_9\endcsname {1001}%
156 \def\XINT_shtb_A {1010}%
157 \def\XINT_shtb_B {1011}%
158 \def\XINT_shtb_C {1100}%
159 \def\XINT_shtb_D {1101}%
160 \def\XINT_shtb_E {1110}%
161 \def\XINT_shtb_F {1111}%
162 \def\XINT_shtb_G {}%
163 \def\XINT_smallhex #1%
164 {%
165     \expandafter\XINT_smallhex_a\expandafter
166     {\the\numexpr (#1+\xint_c_viii)/\xint_c_xvi-\xint_c_i}{#1}%
167 }%
168 \def\XINT_smallhex_a #1#2%

```

```

169 {%
170   \csname XINT_sdth_#1\expandafter\expandafter\expandafter\endcsname
171   \csname XINT_sdth_\the\numexpr #2-\xint_c_xvi*#1\endcsname
172 }%
173 \def\XINT_smallbin #1%
174 {%
175   \expandafter\XINT_smallbin_a\expandafter
176   {\the\numexpr (#1+\xint_c_viii)/\xint_c_xvi-\xint_c_i}{#1}%
177 }%
178 \def\XINT_smallbin_a #1#2%
179 {%
180   \csname XINT_sdtb_#1\expandafter\expandafter\expandafter\endcsname
181   \csname XINT_sdtb_\the\numexpr #2-\xint_c_xvi*#1\endcsname
182 }%

```

35.6 \xintDecToHex, \xintDecToBin

v1.08

```

183 \def\xintDecToHex {\romannumeral0\xintdectohex }%
184 \def\xintdectohex #1%
185   {\expandafter\XINT_dth_checkin\romannumeral-'0#1\W\W\W\W \T}%
186 \def\XINT_dth_checkin #1%
187 {%
188   \xint_UDsignfork
189   #1\dummy \XINT_dth_N
190   -\dummy {\XINT_dth_P #1}%
191   \krof
192 }%
193 \def\XINT_dth_N {\expandafter\xint_minus_andstop\romannumeral0\XINT_dth_P }%
194 \def\XINT_dth_P {\expandafter\XINT_dth_III\romannumeral-'0\XINT_dtbh_I {0.}}%
195 \def\xintDecToBin {\romannumeral0\xintdectobin }%
196 \def\xintdectobin #1%
197   {\expandafter\XINT_dtb_checkin\romannumeral-'0#1\W\W\W\W \T }%
198 \def\XINT_dtb_checkin #1%
199 {%
200   \xint_UDsignfork
201   #1\dummy \XINT_dtb_N
202   -\dummy {\XINT_dtb_P #1}%
203   \krof
204 }%
205 \def\XINT_dtb_N {\expandafter\xint_minus_andstop\romannumeral0\XINT_dtb_P }%
206 \def\XINT_dtb_P {\expandafter\XINT_dtb_III\romannumeral-'0\XINT_dtbh_I {0.}}%
207 \def\XINT_dtbh_I #1#2#3#4#5%
208 {%
209   \xint_gob_til_W #5\XINT_dtbh_II_a\W\XINT_dtbh_I_a  {{{#2#3#4#5}#1\Z.%
210 }%
211 \def\XINT_dtbh_II_a\W\XINT_dtbh_I_a #1#2{\XINT_dtbh_II_b #2}%
212 \def\XINT_dtbh_II_b #1#2#3#4%

```


35 Package *xintbinhex* implementation

```

213 {%
214   \xint_gob_til_W
215   #1\XINT_dtbh_II_c
216   #2\XINT_dtbh_II_ci
217   #3\XINT_dtbh_II_cii
218   \W\XINT_dtbh_II_ciii #1#2#3#4%
219 }%
220 \def\XINT_dtbh_II_c \W\XINT_dtbh_II_ci
221   \W\XINT_dtbh_II_cii
222   \W\XINT_dtbh_II_ciii \W\W\W\W {{{}%
223 \def\XINT_dtbh_II_ci #1\XINT_dtbh_II_ciii #2\W\W\W
224   {\XINT_dtbh_II_d {{}{#2}{0}}}%
225 \def\XINT_dtbh_II_cii\W\XINT_dtbh_II_ciii #1#2\W\W
226   {\XINT_dtbh_II_d {{}{#1#2}{00}}}%
227 \def\XINT_dtbh_II_ciii #1#2#3\W
228   {\XINT_dtbh_II_d {{}{#1#2#3}{000}}}%
229 \def\XINT_dtbh_I_a #1#2#3.%
230 {%
231   \xint_gob_til_Z #3\XINT_dtbh_I_z\Z
232   \expandafter\XINT_dtbh_I_b\the\numexpr #2+#30000.{#1}%
233 }%
234 \def\XINT_dtbh_I_b #1.%
235 {%
236   \expandafter\XINT_dtbh_I_c\the\numexpr
237   (#1+\xint_c_ii^xv)/\xint_c_ii^xvi-\xint_c_i.#1.%
238 }%
239 \def\XINT_dtbh_I_c #1.#2.%
240 {%
241   \expandafter\XINT_dtbh_I_d\expandafter
242   {\the\numexpr #2-\xint_c_ii^xvi*#1}{#1}%
243 }%
244 \def\XINT_dtbh_I_d #1#2#3{\XINT_dtbh_I_a {#3#1.}{#2}}%
245 \def\XINT_dtbh_I_z\Z\expandafter\XINT_dtbh_I_b\the\numexpr #1+#2.%
246 {%
247   \ifnum #1=\xint_c_ \expandafter\XINT_dtbh_I_end_zb\fi
248   \XINT_dtbh_I_end_za {#1}%
249 }%
250 \def\XINT_dtbh_I_end_za #1#2{\XINT_dtbh_I {#2#1.}}%
251 \def\XINT_dtbh_I_end_zb\XINT_dtbh_I_end_za #1#2{\XINT_dtbh_I {#2}}%
252 \def\XINT_dtbh_II_d #1#2#3#4.%
253 {%
254   \xint_gob_til_Z #4\XINT_dtbh_II_z\Z
255   \expandafter\XINT_dtbh_II_e\the\numexpr #2+#4#3.{#1}{#3}%
256 }%
257 \def\XINT_dtbh_II_e #1.%
258 {%
259   \expandafter\XINT_dtbh_II_f\the\numexpr
260   (#1+\xint_c_ii^xv)/\xint_c_ii^xvi-\xint_c_i.#1.%
261 }%

```

```

262 \def\XINT_dtbh_II_f #1.#2.%
263 {%
264   \expandafter\XINT_dtbh_II_g\expandafter
265   {\the\numexpr #2-\xint_c_ii^xvi*#1}{#1}%
266 }%
267 \def\XINT_dtbh_II_g #1#2#3{\XINT_dtbh_II_d {#3#1.}{#2}}%
268 \def\XINT_dtbh_II_z\Z\expandafter\XINT_dtbh_II_e\the\numexpr #1+#2.%
269 {%
270   \ifnum #1=\xint_c_ \expandafter\XINT_dtbh_II_end_zb\fi
271   \XINT_dtbh_II_end_za {#1}%
272 }%
273 \def\XINT_dtbh_II_end_za #1#2#3{{#2#1.\Z.}%
274 \def\XINT_dtbh_II_end_zb\XINT_dtbh_II_end_za #1#2#3{{#2\Z.}%
275 \def\XINT_dth_III #1#2.%
276 {%
277   \xint_gob_til_Z #2\XINT_dth_end\Z
278   \expandafter\XINT_dth_III\expandafter
279   {\romannumeral-'0\XINT_dth_small #2.#1}%
280 }%
281 \def\XINT_dth_small #1.%
282 {%
283   \expandafter\XINT_smallhex\expandafter
284   {\the\numexpr (#1+\xint_c_ii^vii)/\xint_c_ii^viii-\xint_c_i\expandafter}%
285   \romannumeral-'0\expandafter\XINT_smallhex\expandafter
286   {\the\numexpr
287   #1-((#1+\xint_c_ii^vii)/\xint_c_ii^viii-\xint_c_i)*\xint_c_ii^viii}%
288 }%
289 \def\XINT_dth_end\Z\expandafter\XINT_dth_III\expandafter #1#2\T
290 {%
291   \XINT_dth_end_b #1%
292 }%
293 \def\XINT_dth_end_b #1.{\XINT_dth_end_c }%
294 \def\XINT_dth_end_c #1{\xint_gob_til_zero #1\XINT_dth_end_d 0\space #1}%
295 \def\XINT_dth_end_d 0\space 0#1%
296 {%
297   \xint_gob_til_zero #1\XINT_dth_end_e 0\space #1%
298 }%
299 \def\XINT_dth_end_e 0\space 0#1%
300 {%
301   \xint_gob_til_zero #1\XINT_dth_end_f 0\space #1%
302 }%
303 \def\XINT_dth_end_f 0\space 0{ }%
304 \def\XINT_dtb_III #1#2.%
305 {%
306   \xint_gob_til_Z #2\XINT_dtb_end\Z
307   \expandafter\XINT_dtb_III\expandafter
308   {\romannumeral-'0\XINT_dtb_small #2.#1}%
309 }%
310 \def\XINT_dtb_small #1.%

```

```

311 {%
312   \expandafter\XINT_smallbin\expandafter
313   {\the\numexpr (#1+\xint_c_ii^vii)/\xint_c_ii^viii-\xint_c_i\expandafter}%
314   \romannumeral-'0\expandafter\XINT_smallbin\expandafter
315   {\the\numexpr
316   #1-((#1+\xint_c_ii^vii)/\xint_c_ii^viii-\xint_c_i)*\xint_c_ii^viii}%
317 }%
318 \def\XINT_dtb_end_Z\expandafter\XINT_dtb_III\expandafter #1#2\T
319 {%
320   \XINT_dtb_end_b #1%
321 }%
322 \def\XINT_dtb_end_b #1.{\XINT_dtb_end_c }%
323 \def\XINT_dtb_end_c #1#2#3#4#5#6#7#8%
324 {%
325   \expandafter\XINT_dtb_end_d\the\numexpr #1#2#3#4#5#6#7#8\relax
326 }%
327 \def\XINT_dtb_end_d #1#2#3#4#5#6#7#8#9%
328 {%
329   \expandafter\space\the\numexpr #1#2#3#4#5#6#7#8#9\relax
330 }%

```

35.7 \xintHexToDec

v1.08

```

331 \def\xintHexToDec {\romannumeral0\xinthextodec }%
332 \def\xinthextodec #1%
333   {\expandafter\XINT_htd_checkin\romannumeral-'0#1\W\W\W\W \T }%
334 \def\XINT_htd_checkin #1%
335 {%
336   \xint_UDsignfork
337   #1\dummy \XINT_htd_neg
338   -\dummy {\XINT_htd_I {0000}#1}%
339   \krof
340 }%
341 \def\XINT_htd_neg {\expandafter\xint_minus_andstop
342   \romannumeral0\XINT_htd_I {0000}}%
343 \def\XINT_htd_I #1#2#3#4#5%
344 {%
345   \xint_gob_til_W #5\XINT_htd_II_a\W
346   \XINT_htd_I_a {}{"#2#3#4#5}#1\Z\Z\Z\Z
347 }%
348 \def\XINT_htd_II_a \W\XINT_htd_I_a #1#2{\XINT_htd_II_b #2}%
349 \def\XINT_htd_II_b "#1#2#3#4%
350 {%
351   \xint_gob_til_W
352   #1\XINT_htd_II_c
353   #2\XINT_htd_II_ci
354   #3\XINT_htd_II_cii

```

35 Package *xintbinhex* implementation

```

355      \W\XINT_htd_II_ciii #1#2#3#4%
356 }%
357 \def\XINT_htd_II_c \W\XINT_htd_II_ci
358      \W\XINT_htd_II_cii
359      \W\XINT_htd_II_ciii \W\W\W\W #1\Z\Z\Z\Z\T
360 {%
361      \expandafter\xint_cleanupzeros_andstop
362      \romannumeral0\XINT_rord_main { }#1%
363      \xint_relax
364      \xint_bye\xint_bye\xint_bye\xint_bye
365      \xint_bye\xint_bye\xint_bye\xint_bye
366      \xint_relax
367 }%
368 \def\XINT_htd_II_ci #1\XINT_htd_II_ciii
369      #2\W\W\W {\XINT_htd_II_d { }{"#2}{\xint_c_xvi}}%
370 \def\XINT_htd_II_cii\W\XINT_htd_II_ciii
371      #1#2\W\W {\XINT_htd_II_d { }{"#1#2}{\xint_c_ii^viii}}%
372 \def\XINT_htd_II_ciii #1#2#3\W {\XINT_htd_II_d { }{"#1#2#3}{\xint_c_ii^xii}}%
373 \def\XINT_htd_I_a #1#2#3#4#5#6%
374 {%
375      \xint_gob_til_Z #3\XINT_htd_I_end_a\Z
376      \expandafter\XINT_htd_I_b\the\numexpr
377      #2+\xint_c_ii^xvi*#6#5#4#3+\xint_c_x^ix\relax {#1}%
378 }%
379 \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}}%
380 \def\XINT_htd_I_c #1#2#3{\XINT_htd_I_a {#3#2}{#1}}%
381 \def\XINT_htd_I_end_a\Z\expandafter\XINT_htd_I_b\the\numexpr #1+#2\relax
382 {%
383      \expandafter\XINT_htd_I_end_b\the\numexpr \xint_c_x^v+#1\relax
384 }%
385 \def\XINT_htd_I_end_b 1#1#2#3#4#5%
386 {%
387      \xint_gob_til_zero #1\XINT_htd_I_end_bz0%
388      \XINT_htd_I_end_c #1#2#3#4#5%
389 }%
390 \def\XINT_htd_I_end_c #1#2#3#4#5#6{\XINT_htd_I {#6#5#4#3#2#1000}}%
391 \def\XINT_htd_I_end_bz0\XINT_htd_I_end_c 0#1#2#3#4%
392 {%
393      \xint_gob_til_zeros_iv #1#2#3#4\XINT_htd_I_end_bzz 0000%
394      \XINT_htd_I_end_D {#4#3#2#1}%
395 }%
396 \def\XINT_htd_I_end_D #1#2{\XINT_htd_I {#2#1}}%
397 \def\XINT_htd_I_end_bzz 0000\XINT_htd_I_end_D #1{\XINT_htd_I }%
398 \def\XINT_htd_II_d #1#2#3#4#5#6#7%
399 {%
400      \xint_gob_til_Z #4\XINT_htd_II_end_a\Z
401      \expandafter\XINT_htd_II_e\the\numexpr
402      #2+#3*#7#6#5#4+\xint_c_x^viii\relax {#1}{#3}%
403 }%

```

```

404 \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}}%
405 \def\XINT_htd_II_f #1#2#3{\XINT_htd_II_d {#2#3}{#1}}%
406 \def\XINT_htd_II_end_a\Z\expandafter\XINT_htd_II_e
407   \the\numexpr #1+#2\relax #3#4\T
408 {%
409   \XINT_htd_II_end_b #1#3%
410 }%
411 \def\XINT_htd_II_end_b #1#2#3#4#5#6#7#8%
412 {%
413   \expandafter\space\the\numexpr #1#2#3#4#5#6#7#8\relax
414 }%

```

35.8 \xintBinToDec

v1.08

```

415 \def\xintBinToDec {\romannumeral0\xintbintodec }%
416 \def\xintbintodec #1{\expandafter\XINT_btd_checkin
417   \romannumeral-‘0#1\W\W\W\W\W\W\W\W\T }%
418 \def\XINT_btd_checkin #1%
419 {%
420   \xint_UDsignfork
421   #1\dummy \XINT_btd_neg
422   -\dummy {\XINT_btd_I {0000000}{#1}}%
423   \krof
424 }%
425 \def\XINT_btd_neg {\expandafter\xint_minus_andstop
426   \romannumeral0\XINT_btd_I {0000000}}%
427 \def\XINT_btd_I #1#2#3#4#5#6#7#8#9%
428 {%
429   \xint_gob_til_W #9\XINT_btd_II_a {#2#3#4#5#6#7#8#9}\W
430   \XINT_btd_I_a }{\csname XINT_sbtd_#2#3#4#5\endcsname*\xint_c_xvi+%
431     \csname XINT_sbtd_#6#7#8#9\endcsname}%
432   #1\Z\Z\Z\Z\Z\Z\Z
433 }%
434 \def\XINT_btd_II_a #1\W\XINT_btd_I_a #2#3{\XINT_btd_II_b #1}%
435 \def\XINT_btd_II_b #1#2#3#4#5#6#7#8%
436 {%
437   \xint_gob_til_W
438   #1\XINT_btd_II_c
439   #2\XINT_btd_II_ci
440   #3\XINT_btd_II_cii
441   #4\XINT_btd_II_ciii
442   #5\XINT_btd_II_civ
443   #6\XINT_btd_II_cv
444   #7\XINT_btd_II_cvi
445   \W\XINT_btd_II_cvii #1#2#3#4#5#6#7#8%
446 }%
447 \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

```

35 Package *xintbinhex* implementation

```

448 {%
449   \expandafter\XINT_btd_II_c_end
450   \romannumeral0\XINT_rord_main {}#2%
451   \xint_relax
452   \xint_bye\xint_bye\xint_bye\xint_bye
453   \xint_bye\xint_bye\xint_bye\xint_bye
454   \xint_relax
455 }%
456 \def\XINT_btd_II_c_end #1#2#3#4#5#6%
457 {%
458   \expandafter\space\the\numexpr #1#2#3#4#5#6\relax
459 }%
460 \def\XINT_btd_II_ci #1\XINT_btd_II_cvii #2\W\W\W\W\W\W\W
461   {\XINT_btd_II_d {}{#2}{\xint_c_ii }}%
462 \def\XINT_btd_II_cii #1\XINT_btd_II_cvii #2\W\W\W\W\W\W
463   {\XINT_btd_II_d {}{\csname XINT_sbtd_00#2\endcsname }{\xint_c_iv }}%
464 \def\XINT_btd_II_ciii #1\XINT_btd_II_cvii #2\W\W\W\W\W
465   {\XINT_btd_II_d {}{\csname XINT_sbtd_0#2\endcsname }{\xint_c_viii }}%
466 \def\XINT_btd_II_civ #1\XINT_btd_II_cvii #2\W\W\W\W
467   {\XINT_btd_II_d {}{\csname XINT_sbtd_#2\endcsname }{\xint_c_xvi }}%
468 \def\XINT_btd_II_cv #1\XINT_btd_II_cvii #2#3#4#5#6\W\W\W
469 {%
470   \XINT_btd_II_d {}{\csname XINT_sbtd_#2#3#4#5\endcsname*\xint_c_ii+%
471     #6}{\xint_c_ii^v }}%
472 }%
473 \def\XINT_btd_II_cvi #1\XINT_btd_II_cvii #2#3#4#5#6#7\W\W
474 {%
475   \XINT_btd_II_d {}{\csname XINT_sbtd_#2#3#4#5\endcsname*\xint_c_iv+%
476     \csname XINT_sbtd_00#6#7\endcsname }{\xint_c_ii^vi }}%
477 }%
478 \def\XINT_btd_II_cvii #1#2#3#4#5#6#7\W
479 {%
480   \XINT_btd_II_d {}{\csname XINT_sbtd_#1#2#3#4\endcsname*\xint_c_viii+%
481     \csname XINT_sbtd_0#5#6#7\endcsname }{\xint_c_ii^vii }}%
482 }%
483 \def\XINT_btd_II_d #1#2#3#4#5#6#7#8#9%
484 {%
485   \xint_gob_til_Z #4\XINT_btd_II_end_a\Z
486   \expandafter\XINT_btd_II_e\the\numexpr
487   #2+(\xint_c_x^ix+#3*#9#8#7#6#5#4)\relax {#1}{#3}%
488 }%
489 \def\XINT_btd_II_e #1#2#3#4#5#6#7#8#9{\XINT_btd_II_f {#1#2#3}{#4#5#6#7#8#9}}%
490 \def\XINT_btd_II_f #1#2#3{\XINT_btd_II_d {#2#3}{#1}}%
491 \def\XINT_btd_II_end_a\Z\expandafter\XINT_btd_II_e
492   \the\numexpr #1+(\Z\relax #3#4\T
493 {%
494   \XINT_btd_II_end_b #1#3%
495 }%
496 \def\XINT_btd_II_end_b #1#2#3#4#5#6#7#8#9%

```

35 Package *xintbinhex* implementation

```
497 {%
498   \expandafter\space\the\numexpr #1#2#3#4#5#6#7#8#9\relax
499 }%
500 \def\XINT_btd_I_a #1#2#3#4#5#6#7#8%
501 {%
502   \xint_gob_til_Z #3\XINT_btd_I_end_a\Z
503   \expandafter\XINT_btd_I_b\the\numexpr
504   #2+\xint_c_ii^viii*#8#7#6#5#4#3+\xint_c_x^ix\relax {#1}%
505 }%
506 \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}}%
507 \def\XINT_btd_I_c #1#2#3{\XINT_btd_I_a {#3#2}{#1}}%
508 \def\XINT_btd_I_end_a\Z\expandafter\XINT_btd_I_b
509   \the\numexpr #1+\xint_c_ii^viii #2\relax
510 {%
511   \expandafter\XINT_btd_I_end_b\the\numexpr 1000+#1\relax
512 }%
513 \def\XINT_btd_I_end_b 1#1#2#3%
514 {%
515   \xint_gob_til_zeros_iii #1#2#3\XINT_btd_I_end_bz 000%
516   \XINT_btd_I_end_c #1#2#3%
517 }%
518 \def\XINT_btd_I_end_c #1#2#3#4{\XINT_btd_I {#4#3#2#1000}}%
519 \def\XINT_btd_I_end_bz 000\XINT_btd_I_end_c 000{\XINT_btd_I }%
```

35.9 \xintBinToHex

v1.08

```
520 \def\xintBinToHex {\romannumeral0\xintbinto hex }%
521 \def\xintbinto hex #1%
522 {%
523   \expandafter\XINT_bth_checkin
524   \romannumeral0\expandafter\XINT_num_loop
525   \romannumeral-'0#1\xint_relax\xint_relax
526   \xint_relax\xint_relax
527   \xint_relax\xint_relax\xint_relax\xint_relax\Z
528   \R\R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W\W
529 }%
530 \def\XINT_bth_checkin #1%
531 {%
532   \xint_UDsignfork
533   #1\dummy \XINT_bth_N
534   -\dummy {\XINT_bth_P #1}%
535   \krof
536 }%
537 \def\XINT_bth_N {\expandafter\xint_minus_andstop\romannumeral0\XINT_bth_P }%
538 \def\XINT_bth_P {\expandafter\XINT_bth_I\expandafter{\expandafter}%
539   \romannumeral0\XINT_OQ {}}%
540 \def\XINT_bth_I #1#2#3#4#5#6#7#8#9%
```

```

541 {%
542   \xint_gob_til_W #9\XINT_bth_end_a\W
543   \expandafter\expandafter\expandafter
544   \XINT_bth_I
545   \expandafter\expandafter\expandafter
546   {\csname XINT_sbth_#9#8#7#6\expandafter\expandafter\expandafter\endcsname
547    \csname XINT_sbth_#5#4#3#2\endcsname #1}%
548 }%
549 \def\XINT_bth_end_a\W \expandafter\expandafter\expandafter
550   \XINT_bth_I \expandafter\expandafter\expandafter #1%
551 {%
552   \XINT_bth_end_b #1%
553 }%
554 \def\XINT_bth_end_b #1\endcsname #2\endcsname #3%
555 {%
556   \xint_gob_til_zero #3\XINT_bth_end_z 0\space #3%
557 }%
558 \def\XINT_bth_end_z0\space 0{ }%

```

35.10 \xintHexToBin

v1.08

```

559 \def\xintHexToBin {\romannumeral0\xinthextobin }%
560 \def\xinthextobin #1%
561 {%
562   \expandafter\XINT_htb_checkin\romannumeral-'0#1GGGGGGGG\T
563 }%
564 \def\XINT_htb_checkin #1%
565 {%
566   \xint_UDsignfork
567     #1\dummy \XINT_htb_N
568     -\dummy {\XINT_htb_P #1}%
569   \krof
570 }%
571 \def\XINT_htb_N {\expandafter\xint_minus_andstop\romannumeral0\XINT_htb_P }%
572 \def\XINT_htb_P {\XINT_htb_I_a {}}%
573 \def\XINT_htb_I_a #1#2#3#4#5#6#7#8#9%
574 {%
575   \xint_gob_til_G #9\XINT_htb_II_a G%
576   \expandafter\expandafter\expandafter
577   \XINT_htb_I_b
578   \expandafter\expandafter\expandafter
579   {\csname XINT_shtb_#2\expandafter\expandafter\expandafter\endcsname
580    \csname XINT_shtb_#3\expandafter\expandafter\expandafter\endcsname
581    \csname XINT_shtb_#4\expandafter\expandafter\expandafter\endcsname
582    \csname XINT_shtb_#5\expandafter\expandafter\expandafter\endcsname
583    \csname XINT_shtb_#6\expandafter\expandafter\expandafter\endcsname
584    \csname XINT_shtb_#7\expandafter\expandafter\expandafter\endcsname

```


35 Package *xintbinhex* implementation

```
585 \csname XINT_shtb_#8\expandafter\expandafter\expandafter\endcsname
586 \csname XINT_shtb_#9\endcsname }{#1}%
587 }%
588 \def\XINT_htb_I_b #1#2{\XINT_htb_I_a {#2#1}}%
589 \def\XINT_htb_II_a G\expandafter\expandafter\expandafter\XINT_htb_I_b
590 {%
591 \expandafter\expandafter\expandafter \XINT_htb_II_b
592 }%
593 \def\XINT_htb_II_b #1#2#3\T
594 {%
595 \XINT_num_loop #2#1%
596 \xint_relax\xint_relax\xint_relax\xint_relax
597 \xint_relax\xint_relax\xint_relax\xint_relax\Z
598 }%
```

35.11 \xintCHexToBin

v1.08

```
599 \def\xintCHexToBin {\romannumeral0\xintchextobin }%
600 \def\xintchextobin #1%
601 {%
602 \expandafter\XINT_chtb_checkin\romannumeral-'0#1%
603 \R\R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W\W
604 }%
605 \def\XINT_chtb_checkin #1%
606 {%
607 \xint_UDsignfork
608 #1\dummy \XINT_chtb_N
609 -\dummy {\XINT_chtb_P #1}%
610 \krof
611 }%
612 \def\XINT_chtb_N {\expandafter\xint_minus_andstop\romannumeral0\XINT_chtb_P }%
613 \def\XINT_chtb_P {\expandafter\XINT_chtb_I\expandafter{\expandafter}%
614 \romannumeral0\XINT_0Q {}}%
615 \def\XINT_chtb_I #1#2#3#4#5#6#7#8#9%
616 {%
617 \xint_gob_til_W #9\XINT_chtb_end_a\W
618 \expandafter\expandafter\expandafter
619 \XINT_chtb_I
620 \expandafter\expandafter\expandafter
621 {\csname XINT_shtb_#9\expandafter\expandafter\expandafter\endcsname
622 \csname XINT_shtb_#8\expandafter\expandafter\expandafter\endcsname
623 \csname XINT_shtb_#7\expandafter\expandafter\expandafter\endcsname
624 \csname XINT_shtb_#6\expandafter\expandafter\expandafter\endcsname
625 \csname XINT_shtb_#5\expandafter\expandafter\expandafter\endcsname
626 \csname XINT_shtb_#4\expandafter\expandafter\expandafter\endcsname
627 \csname XINT_shtb_#3\expandafter\expandafter\expandafter\endcsname
628 \csname XINT_shtb_#2\endcsname
```

```

629      #1}%
630 }%
631 \def\XINT_chtb_end_a\W\expandafter\expandafter\expandafter
632   \XINT_chtb_I\expandafter\expandafter\expandafter #1%
633 {%
634   \XINT_chtb_end_b #1%
635   \xint_relax\xint_relax\xint_relax\xint_relax
636   \xint_relax\xint_relax\xint_relax\xint_relax\Z
637 }%
638 \def\XINT_chtb_end_b #1\W#2\W#3\W#4\W#5\W#6\W#7\W#8\W\endcsname
639 {%
640   \XINT_num_loop
641 }%
642 \XINT_restorecatcodes_endinput%

```

36 Package *xintgcd* implementation

The commenting is currently (2013/11/28) very sparse. Release 1.09h has modified a bit the `\xintTypesetEuclideanAlgorithm` and `\xintTypesetBezoutAlgorithm` layout with respect to line indentation in particular. And they use the *xinttools* `\xintloop` rather than the Plain \TeX or \LaTeX 's `\loop`.

Contents

.1	Catcodes, ε - \TeX and reload detection ..	256	.9	<code>\xintLCMof</code>	260
.2	Confirmation of <i>xint</i> loading	257	.10	<code>\xintLCMof:csv</code>	260
.3	Catcodes	258	.11	<code>\xintBezout</code>	261
.4	Package identification	258	.12	<code>\xintEuclideanAlgorithm</code>	265
.5	<code>\xintGCD</code>	258	.13	<code>\xintBezoutAlgorithm</code>	266
.6	<code>\xintGCDoF</code>	259	.14	<code>\xintTypesetEuclideanAlgorithm</code> ...	268
.7	<code>\xintGCDoF:csv</code>	259	.15	<code>\xintTypesetBezoutAlgorithm</code>	269
.8	<code>\xintLCM</code>	260			

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 master *xint* package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2   \catcode13=5      % ^^M
3   \endlinechar=13 %
4   \catcode123=1     % {
5   \catcode125=2     % }
6   \catcode64=11     % @
7   \catcode35=6      % #

```

```

8  \catcode44=12    % ,
9  \catcode45=12    % -
10 \catcode46=12    % .
11 \catcode58=12    % :
12 \def\space { }%
13 \let\z\endgroup
14 \expandafter\let\expandafter\x\csname ver@xintgcd.sty\endcsname
15 \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
16 \expandafter
17   \ifx\csname PackageInfo\endcsname\relax
18     \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
19   \else
20     \def\y#1#2{\PackageInfo{#1}{#2}}%
21   \fi
22 \expandafter
23 \ifx\csname numexpr\endcsname\relax
24   \y{xintgcd}{\numexpr not available, aborting input}%
25   \aftergroup\endinput
26 \else
27   \ifx\x\relax    % plain-TeX, first loading of xintgcd.sty
28     \ifx\w\relax % but xint.sty not yet loaded.
29       \y{xintgcd}{Package xint is required}%
30       \y{xintgcd}{Will try \string\input\space xint.sty}%
31       \def\z{\endgroup\input xint.sty\relax}%
32     \fi
33   \else
34     \def\empty {}%
35     \ifx\x\empty % LaTeX, first loading,
36       % variable is initialized, but \ProvidesPackage not yet seen
37       \ifx\w\relax % xint.sty not yet loaded.
38         \y{xintgcd}{Package xint is required}%
39         \y{xintgcd}{Will try \string\RequirePackage{xint}}%
40         \def\z{\endgroup\RequirePackage{xint}}%
41       \fi
42     \else
43       \y{xintgcd}{I was already loaded, aborting input}%
44       \aftergroup\endinput
45     \fi
46   \fi
47 \fi
48 \z%

```

36.2 Confirmation of *xint* loading

```

49 \begingroup\catcode61\catcode48\catcode32=10\relax%
50 \catcode13=5    % ^^M
51 \endlinechar=13 %
52 \catcode123=1   % {
53 \catcode125=2   % }

```

```

54 \catcode64=11 % @
55 \catcode35=6 % #
56 \catcode44=12 % ,
57 \catcode45=12 % -
58 \catcode46=12 % .
59 \catcode58=12 % :
60 \ifdefined\PackageInfo
61   \def\y#1#2{\PackageInfo{#1}{#2}}%
62   \else
63     \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
64   \fi
65 \def\empty {}%
66 \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
67 \ifx\w\relax % Plain TeX, user gave a file name at the prompt
68   \y{xintgcd}{Loading of package xint failed, aborting input}%
69   \aftergroup\endinput
70 \fi
71 \ifx\w\empty % LaTeX, user gave a file name at the prompt
72   \y{xintgcd}{Loading of package xint failed, aborting input}%
73   \aftergroup\endinput
74 \fi
75 \endgroup%

```

36.3 Catcodes

```
76 \XINTsetupcatcodes%
```

36.4 Package identification

```

77 \XINT_providespackage
78 \ProvidesPackage{xintgcd}%
79 [2013/11/28 v1.09h Euclidean algorithm with xint package (jfb)]%

```

36.5 \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.

```

80 \def\xintGCD {\romannumeral0\xintgcd}%
81 \def\xintgcd #1%
82 {%
83   \expandafter\XINT_gcd\expandafter{\romannumeral0\xintiabs {#1}}%
84 }%
85 \def\XINT_gcd #1#2%
86 {%
87   \expandafter\XINT_gcd_fork\romannumeral0\xintiabs {#2}\Z #1\Z
88 }%

Ici #3#4=A, #1#2=B

89 \def\XINT_gcd_fork #1#2\Z #3#4\Z

```

```

90 {%
91   \xint_UDzerofork
92   #1\dummy \XINT_gcd_BisZero
93   #3\dummy \XINT_gcd_AisZero
94   0\dummy \XINT_gcd_loop
95   \krof
96   {#1#2}{#3#4}%
97 }%
98 \def\xint_gcd_AisZero #1#2{ #1}%
99 \def\xint_gcd_BisZero #1#2{ #2}%
100 \def\xint_gcd_CheckRem #1#2\Z
101 {%
102   \xint_gob_til_zero #1\xint_gcd_end0\XINT_gcd_loop {#1#2}%
103 }%
104 \def\xint_gcd_end0\XINT_gcd_loop #1#2{ #2}%

#1=B, #2=A

105 \def\xint_gcd_loop #1#2%
106 {%
107   \expandafter\expandafter\expandafter
108   \XINT_gcd_CheckRem
109   \expandafter\xint_seconddoftwo
110   \romannumeral0\XINT_div_prepare {#1}{#2}\Z
111   {#1}%
112 }%

```

36.6 \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 ...

```

113 \def\xintGCDof      {\romannumeral0\xintgcdof }%
114 \def\xintgcdof      #1{\expandafter\XINT_gcdof_a\romannumeral-‘0#1\relax }%
115 \def\XINT_gcdof_a #1{\expandafter\XINT_gcdof_b\romannumeral-‘0#1\Z }%
116 \def\XINT_gcdof_b #1\Z #2{\expandafter\XINT_gcdof_c\romannumeral-‘0#2\Z {#1}\Z}%
117 \def\XINT_gcdof_c #1{\xint_gob_til_relax #1\XINT_gcdof_e\relax\XINT_gcdof_d #1}%
118 \def\XINT_gcdof_d #1\Z {\expandafter\XINT_gcdof_b\romannumeral0\xintgcd {#1}}%
119 \def\XINT_gcdof_e #1\Z #2\Z { #2}%

```

36.7 \xintGCDof:csv

1.09a. For use by \xintexpr.

```

120 \def\xintGCDof:csv #1{\expandafter\XINT_gcdof:_b\romannumeral-‘0#1,,}%
121 \def\XINT_gcdof:_b #1,#2,{\expandafter\XINT_gcdof:_c\romannumeral-‘0#2,{#1},}%
122 \def\XINT_gcdof:_c #1{\if #1,\expandafter\XINT_gcdof:_e
123   \else\expandafter\XINT_gcdof:_d\fi #1}%
124 \def\XINT_gcdof:_d #1,{\expandafter\XINT_gcdof:_b\romannumeral0\xintgcd {#1}}%
125 \def\XINT_gcdof:_e ,#1,{#1}%

```

36.8 \xintLCM

New with 1.09a. Inadvertent use of \xintiQuo which was promoted at the same time to add the \xintnum overhead. So with 1.09f \xintiiQuo without the overhead.

```

126 \def\xintLCM {\romannumeral0\xintlcm}%
127 \def\xintlcm #1%
128 {%
129   \expandafter\xINT_lcm\expandafter{\romannumeral0\xintiabs {#1}}%
130 }%
131 \def\xINT_lcm #1#2%
132 {%
133   \expandafter\xINT_lcm_fork\romannumeral0\xintiabs {#2}\Z #1\Z
134 }%
135 \def\xINT_lcm_fork #1#2\Z #3#4\Z
136 {%
137   \xint_UDzerofork
138     #1\dummy \XINT_lcm_BisZero
139     #3\dummy \XINT_lcm_AisZero
140     0\dummy \expandafter
141   \krof
142   \XINT_lcm_notzero\expandafter{\romannumeral0\xINT_gcd_loop {#1#2}{#3#4}}%
143   {#1#2}{#3#4}%
144 }%
145 \def\xINT_lcm_AisZero #1#2#3#4#5{ 0}%
146 \def\xINT_lcm_BisZero #1#2#3#4#5{ 0}%
147 \def\xINT_lcm_notzero #1#2#3{\xintiimul {#2}{\xintiiQuo{#3}{#1}}}%

```

36.9 \xintLCMof

New with 1.09a

```

148 \def\xintLCMof {\romannumeral0\xintlcmof}%
149 \def\xintlcmof #1{\expandafter\xINT_lcmof_a\romannumeral-‘0#1\relax}%
150 \def\xINT_lcmof_a #1{\expandafter\xINT_lcmof_b\romannumeral-‘0#1\Z}%
151 \def\xINT_lcmof_b #1\Z #2{\expandafter\xINT_lcmof_c\romannumeral-‘0#2\Z {#1}\Z}%
152 \def\xINT_lcmof_c #1{\xint_gob_til_relax #1\xINT_lcmof_e\relax\xINT_lcmof_d #1}%
153 \def\xINT_lcmof_d #1\Z {\expandafter\xINT_lcmof_b\romannumeral0\xintlcm {#1}}%
154 \def\xINT_lcmof_e #1\Z #2\Z { #2}%

```

36.10 \xintLCMof:csv

1.09a. For use by \xintexpr.

```

155 \def\xintLCMof:csv #1{\expandafter\xINT_lcmof:_a\romannumeral-‘0#1,,}%
156 \def\xINT_lcmof:_a #1,#2,{\expandafter\xINT_lcmof:_c\romannumeral-‘0#2,{#1},}%
157 \def\xINT_lcmof:_c #1{\if#1,\expandafter\xINT_lcmof:_e
158   \else\expandafter\xINT_lcmof:_d\fi #1}%
159 \def\xINT_lcmof:_d #1,{\expandafter\xINT_lcmof:_a\romannumeral0\xintlcm {#1}}%
160 \def\xINT_lcmof:_e ,#1,{#1}%

```

36.11 \xintBezout

1.09a inserts use of \xintnum

```

161 \def\xintBezout {\romannumeral0\xintbezout }%
162 \def\xintbezout #1%
163 {%
164   \expandafter\xint_bezout\expandafter {\romannumeral0\xintnum{#1}}%
165 }%
166 \def\xint_bezout #1#2%
167 {%
168   \expandafter\XINT_bezout_fork \romannumeral0\xintnum{#2}\Z #1\Z
169 }%

#3#4 = A, #1#2=B

170 \def\XINT_bezout_fork #1#2\Z #3#4\Z
171 {%
172   \xint_UDzerosfork
173   #1#3\dummy \XINT_bezout_botharezero
174   #10\dummy \XINT_bezout_secondiszero
175   #30\dummy \XINT_bezout_firstiszero
176   00\dummy
177   {\xint_UDsignsfork
178     #1#3\dummy \XINT_bezout_minusminus % A < 0, B < 0
179     #1-\dummy \XINT_bezout_minusplus % A > 0, B < 0
180     #3-\dummy \XINT_bezout_plusminus % A < 0, B > 0
181     --\dummy \XINT_bezout_plusplus % A > 0, B > 0
182   \krof }%
183   \krof
184   {#2}{#4}#1#3{#3#4}{#1#2}% #1#2=B, #3#4=A
185 }%
186 \def\XINT_bezout_botharezero #1#2#3#4#5#6%
187 {%
188   \xintError:NoBezoutForZeros
189   \space {0}{0}{0}{0}{0}%
190 }%

attention première entrée doit être ici (-1)^n donc 1
#4#2 = 0 = A, B = #3#1

191 \def\XINT_bezout_firstiszero #1#2#3#4#5#6%
192 {%
193   \xint_UDsignsfork
194   #3\dummy { {0}{#3#1}{0}{1}{#1}}%
195   -\dummy { {0}{#3#1}{0}{-1}{#1}}%
196   \krof
197 }%

#4#2 = A, B = #3#1 = 0

```

```

198 \def\XINT_bezout_secondiszero #1#2#3#4#5#6%
199 {%
200   \xint_UDsignfork
201     #4\dummy{ {#4#2}{0}{-1}{0}{#2}}%
202     -\dummy{ {#4#2}{0}{1}{0}{#2}}%
203   \krof
204 }%

#4#2= A < 0, #3#1 = B < 0

205 \def\XINT_bezout_minusminus #1#2#3#4%
206 {%
207   \expandafter\XINT_bezout_mm_post
208   \romannumeral0\XINT_bezout_loop_a 1{#1}{#2}1001%
209 }%
210 \def\XINT_bezout_mm_post #1#2%
211 {%
212   \expandafter\XINT_bezout_mm_postb\expandafter
213   {\romannumeral0\xintiiopp{#2}}{\romannumeral0\xintiiopp{#1}}%
214 }%
215 \def\XINT_bezout_mm_postb #1#2%
216 {%
217   \expandafter\XINT_bezout_mm_postc\expandafter {#2}{#1}%
218 }%
219 \def\XINT_bezout_mm_postc #1#2#3#4#5%
220 {%
221   \space {#4}{#5}{#1}{#2}{#3}%
222 }%

minusplus #4#2= A > 0, B < 0

223 \def\XINT_bezout_minusplus #1#2#3#4%
224 {%
225   \expandafter\XINT_bezout_mp_post
226   \romannumeral0\XINT_bezout_loop_a 1{#1}{#4#2}1001%
227 }%
228 \def\XINT_bezout_mp_post #1#2%
229 {%
230   \expandafter\XINT_bezout_mp_postb\expandafter
231   {\romannumeral0\xintiiopp {#2}}{#1}%
232 }%
233 \def\XINT_bezout_mp_postb #1#2#3#4#5%
234 {%
235   \space {#4}{#5}{#2}{#1}{#3}%
236 }%

plusminus A < 0, B > 0

237 \def\XINT_bezout_plusminus #1#2#3#4%
238 {%

```



```

239 \expandafter\XINT_bezout_pm_post
240 \romannumeral0\XINT_bezout_loop_a 1{#3#1}{#2}1001%
241 }%
242 \def\XINT_bezout_pm_post #1%
243 {%
244 \expandafter \XINT_bezout_pm_postb \expandafter
245 {\romannumeral0\xintiiopp{#1}}%
246 }%
247 \def\XINT_bezout_pm_postb #1#2#3#4#5%
248 {%
249 \space {#4}{#5}{#1}{#2}{#3}%
250 }%

plusplus

251 \def\XINT_bezout_plusplus #1#2#3#4%
252 {%
253 \expandafter\XINT_bezout_pp_post
254 \romannumeral0\XINT_bezout_loop_a 1{#3#1}{#4#2}1001%
255 }%

la parité  $(-1)^N$  est en #1, et on la jette ici.

256 \def\XINT_bezout_pp_post #1#2#3#4#5%
257 {%
258 \space {#4}{#5}{#1}{#2}{#3}%
259 }%

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

260 \def\XINT_bezout_loop_a #1#2#3%
261 {%
262 \expandafter\XINT_bezout_loop_b
263 \expandafter{\the\numexpr -#1\expandafter}%
264 \romannumeral0\XINT_div_prepare {#2}{#3}{#2}%
265 }%

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)
 $\{(-1)^n\{q(n)\{r(n)\{r(n-1)\}\alpha(n-1)\}\beta(n-1)\}\alpha(n-2)\}\beta(n-2)\}$ 

266 \def\XINT_bezout_loop_b #1#2#3#4#5#6#7#8%
267 {%
268 \expandafter \XINT_bezout_loop_c \expandafter
269 {\romannumeral0\xintiiadd{\XINT_Mul{#5}{#2}}{#7}}%
270 {\romannumeral0\xintiiadd{\XINT_Mul{#6}{#2}}{#8}}%
271 {#1}{#3}{#4}{#5}{#6}%
272 }%

```

```

{alpha(n)}{->beta(n)}{-(-1)^n}{r(n)}{r(n-1)}{alpha(n-1)}{beta(n-1)}

273 \def\XINT_bezout_loop_c #1#2%
274 {%
275     \expandafter \XINT_bezout_loop_d \expandafter
276         {#2}{#1}%
277 }%

{beta(n)}{alpha(n)}{(-1)^(n+1)}{r(n)}{r(n-1)}{alpha(n-1)}{beta(n-1)}

278 \def\XINT_bezout_loop_d #1#2#3#4#5%
279 {%
280     \XINT_bezout_loop_e #4\Z {#3}{#5}{#2}{#1}%
281 }%

r(n)\Z {(-1)^(n+1)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}

282 \def\XINT_bezout_loop_e #1#2\Z
283 {%
284     \xint_gob_til_zero #1\xint_bezout_loop_exit0\XINT_bezout_loop_f
285     {#1#2}%
286 }%

{r(n)}{(-1)^(n+1)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}

287 \def\XINT_bezout_loop_f #1#2%
288 {%
289     \XINT_bezout_loop_a {#2}{#1}%
290 }%

{(-1)^(n+1)}{r(n)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)} et itéra-
tion

291 \def\xint_bezout_loop_exit0\XINT_bezout_loop_f #1#2%
292 {%
293     \ifcase #2
294     \or \expandafter\XINT_bezout_exiteven
295     \else\expandafter\XINT_bezout_exitodd
296     \fi
297 }%
298 \def\XINT_bezout_exiteven #1#2#3#4#5%
299 {%
300     \space {#5}{#4}{#1}%
301 }%
302 \def\XINT_bezout_exitodd #1#2#3#4#5%
303 {%
304     \space {-#5}{-#4}{#1}%
305 }%

```

36.12 \xintEuclideanAlgorithm

Pour Euclide: $\{N\}\{A\}\{D=r(n)\}\{B\}\{q_1\}\{r_1\}\{q_2\}\{r_2\}\{q_3\}\{r_3\}\dots\{q_N\}\{r_N=0\}$
 $u_{<2n} = u_{<2n+3} > u_{<2n+2} > + u_{<2n+4} >$ à la n ième étape

```

306 \def\xintEuclideanAlgorithm {\romannumeral0\xinteuclideanalgorithm}%
307 \def\xinteuclideanalgorithm #1%
308 {%
309   \expandafter \XINT_euc \expandafter{\romannumeral0\xintiabs {#1}}%
310}%
311 \def\XINT_euc #1#2%
312 {%
313   \expandafter\XINT_euc_fork \romannumeral0\xintiabs {#2}\Z #1\Z
314}%

Ici #3#4=A, #1#2=B

315 \def\XINT_euc_fork #1#2\Z #3#4\Z
316 {%
317   \xint_UDzerofork
318   #1\dummy \XINT_euc_BisZero
319   #3\dummy \XINT_euc_AisZero
320   0\dummy \XINT_euc_a
321   \krof
322   {0}\{#1#2\}\{#3#4\}\{#3#4\}\{#1#2\}\}\Z
323}%

Le {} pour protéger {{A}\{B}} si on s'arrête après une étape (B divise A). On va
renvoyer:
{N}\{A\}\{D=r(n)\}\{B\}\{q_1\}\{r_1\}\{q_2\}\{r_2\}\{q_3\}\{r_3\}\dots\{q_N\}\{r_N=0\}

324 \def\XINT_euc_AisZero #1#2#3#4#5#6{ {1}\{0\}\{#2\}\{#2\}\{0\}\{0\}}%
325 \def\XINT_euc_BisZero #1#2#3#4#5#6{ {1}\{0\}\{#3\}\{#3\}\{0\}\{0\}}%

{n}\{r_n\}\{a_n\}\{\{q_n\}\{r_n\}\}\dots\{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

326 \def\XINT_euc_a #1#2#3%
327 {%
328   \expandafter\XINT_euc_b
329   \expandafter {\the\numexpr #1+1\expandafter}%
330   \romannumeral0\XINT_div_prepare {#2}\{#3\}\{#2\}%
331}%

{n+1}\{q(n+1)\}\{r(n+1)\}\{r_n\}\{\{q_n\}\{r_n\}\}\dots

332 \def\XINT_euc_b #1#2#3#4%
333 {%
334   \XINT_euc_c #3\Z {#1}\{#3\}\{#4\}\{\{#2\}\{#3\}\}%
335}%

```

$r(n+1) \setminus \mathbb{Z}$ $\{n+1\}\{r(n+1)\}\{r(n)\}\{\{q(n+1)\}\{r(n+1)\}\}\{\{q_n\}\{r_n\}\}\dots$
 Test si $r(n+1)$ est nul.

```

336 \def\XINT_euc_c #1#2\Z
337 {%
338   \xint_gob_til_zero #1\xint_euc_end0\XINT_euc_a
339 }%

{n+1}{r(n+1)}{r(n)}{\{q(n+1)\}\{r(n+1)\}}\dots\Z Ici  $r(n+1) = 0$ . On arrête on se
prépare à inverser  $\{n+1\}\{0\}\{r(n)\}\{\{q(n+1)\}\{r(n+1)\}}\dots\{\{q_1\}\{r_1\}\}\{\{A\}\{B\}\}\}\Z$ 
On veut renvoyer:  $\{N=n+1\}\{A\}\{D=r(n)\}\{B\}\{q_1\}\{r_1\}\{q_2\}\{r_2\}\{q_3\}\{r_3\}\dots\{q_N\}\{r_N=0\}$ 

340 \def\xint_euc_end0\XINT_euc_a #1#2#3#4\Z%
341 {%
342   \expandafter\xint_euc_end_
343   \romannumeral0%
344   \XINT_rord_main {}#4{\{#1\}\{#3\}}%
345   \xint_relax
346   \xint_bye\xint_bye\xint_bye\xint_bye
347   \xint_bye\xint_bye\xint_bye\xint_bye
348   \xint_relax
349 }%
350 \def\xint_euc_end_ #1#2#3%
351 {%
352   \space {\{#1\}\{#3\}\{#2\}}%
353 }%
```

36.13 \xintBezoutAlgorithm

Pour Bezout: objectif, renvoyer

$\{N\}\{A\}\{0\}\{1\}\{D=r(n)\}\{B\}\{1\}\{0\}\{q_1\}\{r_1\}\{\alpha_1=q_1\}\{\beta_1=1\}$
 $\{q_2\}\{r_2\}\{\alpha_2\}\{\beta_2\}\dots\{q_N\}\{r_N=0\}\{\alpha_N=A/D\}\{\beta_N=B/D\}$
 $\alpha_0=1, \beta_0=0, \alpha(-1)=0, \beta(-1)=1$

```

354 \def\xintBezoutAlgorithm {\romannumeral0\xintbezoutalgorithm}%
355 \def\xintbezoutalgorithm #1%
356 {%
357   \expandafter \XINT_bezalg \expandafter{\romannumeral0\xintiabs {\{#1\}}}%
358 }%
359 \def\XINT_bezalg #1#2%
360 {%
361   \expandafter\XINT_bezalg_fork \romannumeral0\xintiabs {\{#2\}}\Z #1\Z
362 }%

Ici #3#4=A, #1#2=B

363 \def\XINT_bezalg_fork #1#2\Z #3#4\Z
364 {%
365   \xint_UDzerofork
```

36 Package *xintgcd* implementation

```

366      #1\dummy \XINT_bezalg_BisZero
367      #3\dummy \XINT_bezalg_AisZero
368      0\dummy \XINT_bezalg_a
369      \krof
370      0{#1#2}{#3#4}1001{#3#4}{#1#2}{#1#2}\Z
371 }%
372 \def\XINT_bezalg_AisZero #1#2#3\Z{ {1}{0}{0}{1}{#2}{#2}{1}{0}{0}{0}{1}}%
373 \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

374 \def\XINT_bezalg_a #1#2#3%
375 {%
376     \expandafter\XINT_bezalg_b
377     \expandafter {\the\numexpr #1+1\expandafter }%
378     \romannumeral0\XINT_div_prepare {#2}{#3}{#2}%
379 }%

{n+1}{q(n+1)}{r(n+1)}{r(n)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}...

380 \def\XINT_bezalg_b #1#2#3#4#5#6#7#8%
381 {%
382     \expandafter\XINT_bezalg_c\expandafter
383     {\romannumeral0\xintiiadd {\xintiiMul {#6}{#2}}{#8}}%
384     {\romannumeral0\xintiiadd {\xintiiMul {#5}{#2}}{#7}}%
385     {#1}{#2}{#3}{#4}{#5}{#6}%
386 }%

{beta(n+1)}{alpha(n+1)}{n+1}{q(n+1)}{r(n+1)}{r(n)}{alpha(n)}{beta(n)}

387 \def\XINT_bezalg_c #1#2#3#4#5#6%
388 {%
389     \expandafter\XINT_bezalg_d\expandafter {#2}{#3}{#4}{#5}{#6}{#1}%
390 }%

{alpha(n+1)}{n+1}{q(n+1)}{r(n+1)}{r(n)}{beta(n+1)}

391 \def\XINT_bezalg_d #1#2#3#4#5#6#7#8%
392 {%
393     \XINT_bezalg_e #4\Z {#2}{#4}{#5}{#1}{#6}{#7}{#8}{#3}{#4}{#1}{#6}}%
394 }%

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.

395 \def\XINT_bezalg_e #1#2\Z
396 {%
397     \xint_gob_til_zero #1\xint_bezalg_end0\XINT_bezalg_a
398 }%

```

```

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
{ $N$ }{ $A$ }{ $0$ }{ $1$ }{ $D=r(n)$ }{ $B$ }{ $1$ }{ $0$ }{ $q_1$ }{ $r_1$ }{ $\alpha_1=q_1$ }{ $\beta_1=1$ }
{ $q_2$ }{ $r_2$ }{ $\alpha_2$ }{ $\beta_2$ }...{ $q_N$ }{ $r_N=0$ }{ $\alpha_N=A/D$ }{ $\beta_N=B/D$ }

399 \def\xint_bezalg_end0\xINT_bezalg_a #1#2#3#4#5#6#7#8\Z
400 {%
401   \expandafter\xint_bezalg_end_
402   \romannumeral0%
403   \XINT_rord_main { }#8{{#1}{{#3}}}%
404   \xint_relax
405   \xint_bye\xint_bye\xint_bye\xint_bye
406   \xint_bye\xint_bye\xint_bye\xint_bye
407   \xint_relax
408 }%

{ $N$ }{ $D$ }{ $A$ }{ $B$ }{ $q_1$ }{ $r_1$ }{ $\alpha_1=q_1$ }{ $\beta_1=1$ }{ $q_2$ }{ $r_2$ }{ $\alpha_2$ }{ $\beta_2$ }
...{ $q_N$ }{ $r_N=0$ }{ $\alpha_N=A/D$ }{ $\beta_N=B/D$ }
On veut renvoyer
{ $N$ }{ $A$ }{ $0$ }{ $1$ }{ $D=r(n)$ }{ $B$ }{ $1$ }{ $0$ }{ $q_1$ }{ $r_1$ }{ $\alpha_1=q_1$ }{ $\beta_1=1$ }
{ $q_2$ }{ $r_2$ }{ $\alpha_2$ }{ $\beta_2$ }...{ $q_N$ }{ $r_N=0$ }{ $\alpha_N=A/D$ }{ $\beta_N=B/D$ }

409 \def\xint_bezalg_end_ #1#2#3#4%
410 {%
411   \space {#1}{{#3}}{0}{{1}{{#2}{{#4}{{1}{{0}}}%
412 }%

```

36.14 \xintTypesetEuclideanAlgorithm

TYPESETTING

Organisation:

```

{ $N$ }{ $A$ }{ $D$ }{ $B$ }{ $q_1$ }{ $r_1$ }{ $q_2$ }{ $r_2$ }{ $q_3$ }{ $r_3$ }...{ $q_N$ }{ $r_N=0$ }
\U1 = N = nombre d'étapes, \U3 = PGCD, \U2 = A, \U4=B q1 = \U5, q2 = \U7 --> qn =
\U<2n+3>, rn = \U<2n+4> bn = rn. B = r0. A=r(-1)
r(n-2) = q(n)r(n-1)+r(n) (n e étape)
\U{2n} = \U{2n+3} \times \U{2n+2} + \U{2n+4}, n e étape. (avec n entre 1 et N)
1.09h uses \xintloop, and \par rather than \endgraf; and \par rather than
\hfill\break

```

```

413 \def\xintTypesetEuclideanAlgorithm #1#2%
414 {% l'algo remplace #1 et #2 par |#1| et |#2|
415   \par
416   \begingroup
417     \xintAssignArray\xintEuclideanAlgorithm {#1}{{#2}}\to\U
418     \edef\A{\U2}\edef\B{\U4}\edef\N{\U1}%
419     \setbox 0 \vbox{\halign {###$\cr \A\cr \B \cr}}%
420     \count 255 1

```

```

421 \xintloop
422 \indent\hbox to \wd 0 {\hfil$U{\numexpr 2*\count255\relax}$}%
423 ${} = \U{\numexpr 2*\count255 + 3\relax}
424 \times \U{\numexpr 2*\count255 + 2\relax}
425 + \U{\numexpr 2*\count255 + 4\relax}$%
426 \ifnum \count255 < \N
427 \par
428 \advance \count255 1
429 \repeat
430 \endgroup
431 }%

```

36.15 \xintTypesetBezoutAlgorithm

Pour Bezout on a: $\{N\}\{A\}\{0\}\{1\}\{D=r(n)\}\{B\}\{1\}\{0\}\{q_1\}\{r_1\}\{\alpha_1=q_1\}\{\beta_1=1\}$
 $\{q_2\}\{r_2\}\{\alpha_2\}\{\beta_2\}\dots\{q_N\}\{r_N=0\}\{\alpha_N=A/D\}\{\beta_N=B/D\}$ Donc $4N+8$ ter-
mes: $U_1 = N$, $U_2 = A$, $U_5 = D$, $U_6 = B$, $q_1 = U_9$, $q_n = U_{4n+5}$, n au moins 1
 $r_n = U_{4n+6}$, n au moins -1
 $\alpha(n) = U_{4n+7}$, n au moins -1
 $\beta(n) = U_{4n+8}$, n au moins -1
1.09h uses \xintloop, and \par rather than \endgraf; and no more \parindent0pt

```

432 \def\xintTypesetBezoutAlgorithm #1#2%
433 {%
434 \par
435 \begingroup
436 \xintAssignArray\xintBezoutAlgorithm {#1}{#2}\to\BEZ
437 \edef\A{\BEZ2}\edef\B{\BEZ6}\edef\N{\BEZ1}% A = |#1|, B = |#2|
438 \setbox 0 \vbox{\halign {$##$\cr \A\cr \B \cr}}%
439 \count255 1
440 \xintloop
441 \indent\hbox to \wd 0 {\hfil$\BEZ{4*\count255 - 2}$}%
442 ${} = \BEZ{4*\count255 + 5}
443 \times \BEZ{4*\count255 + 2}
444 + \BEZ{4*\count255 + 6}$\hfill\break
445 \hbox to \wd 0 {\hfil$\BEZ{4*\count255 + 7}$}%
446 ${} = \BEZ{4*\count255 + 5}
447 \times \BEZ{4*\count255 + 3}
448 + \BEZ{4*\count255 - 1}$\hfill\break
449 \hbox to \wd 0 {\hfil$\BEZ{4*\count255 + 8}$}%
450 ${} = \BEZ{4*\count255 + 5}
451 \times \BEZ{4*\count255 + 4}
452 + \BEZ{4*\count255 }$
453 \par
454 \ifnum \count255 < \N
455 \advance \count255 1
456 \repeat
457 \edef\U{\BEZ{4*\N + 4}}%
458 \edef\V{\BEZ{4*\N + 3}}%

```

```

459 \edef\D{\BEZ5}%
460 \ifodd\N
461   $\U\times\A - \V\times \B = -\D$\%
462 \else
463   $\U\times\A - \V\times\B = \D$\%
464 \fi
465 \par
466 \endgroup
467 }%
468 \XINT_restorecatcodes_endinput%

```

37 Package **xintfrac** implementation

The commenting is currently (2013/11/28) very sparse.

Contents

.1	Catcodes, ε -TeX and reload detection ..	271	.27	\xintJrr	287
.2	Confirmation of xint loading	272	.28	\xintTrunc, \xintiTrunc	289
.3	Catcodes	272	.29	\xintRound, \xintiRound	291
.4	Package identification	273	.30	\xintRound:csv	292
.5	\xintLen	273	.31	\xintDigits	293
.6	\XINT_lenrord_loop	273	.32	\xintFloat	293
.7	\XINT_outfrac	274	.33	\xintFloat:csv	296
.8	\XINT_inFrac	274	.34	\XINT_inFloat	297
.9	\XINT_frac	275	.35	\xintAdd	299
.10	\XINT_factortens, \XINT_cuz_cnt ..	277	.36	\xintSub	300
.11	\xintRaw	279	.37	\xintSum, \xintSumExpr	300
.12	\xintPraw	279	.38	\xintSum:csv	301
.13	\xintRawWithZeros	280	.39	\xintMul	301
.14	\xintFloor	280	.40	\xintSqr	301
.15	\xintCeil	281	.41	\xintPow	302
.16	\xintNumerator	281	.42	\xintFac	303
.17	\xintDenominator	281	.43	\xintPrd, \xintPrdExpr	303
.18	\xintFrac	282	.44	\xintPrd:csv	303
.19	\xintSignedFrac	282	.45	\xintDiv	304
.20	\xintFwOver	283	.46	\xintIsOne	304
.21	\xintSignedFwOver	283	.47	\xintGeq	304
.22	\xintREZ	284	.48	\xintMax	306
.23	\xintE	285	.49	\xintMaxof	306
.24	\xintIrr	285	.50	\xintMaxof:csv	307
.25	\xintNum	287	.51	\XINTinFloatMaxof	307
.26	\xintifInt	287	.52	\XINTinFloatMaxof:csv	307

.53	\xintMin.....	308	.63	\xintFloatSub.....	313
.54	\xintMinof.....	308	.64	\xintFloatMul.....	314
.55	\xintMinof:csv.....	309	.65	\xintFloatDiv.....	314
.56	\XINTinFloatMinof.....	309	.66	\XINTinFloatSum.....	315
.57	\XINTinFloatMinof:csv.....	309	.67	\XINTinFloatSum:csv.....	315
.58	\xintCmp.....	310	.68	\XINTinFloatPrd.....	316
.59	\xintAbs.....	311	.69	\XINTinFloatPrd:csv.....	316
.60	\xintOpp.....	312	.70	\xintFloatPow.....	316
.61	\xintSgn.....	312	.71	\xintFloatPower.....	319
.62	\xintFloatAdd.....	312	.72	\xintFloatSqrt.....	321

37.1 Catcodes, ε -TeX and reload detection

The code for reload detection is copied from HEIKO OBERDIEK's packages, and adapted here to check for previous loading of the master *xint* package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2  \catcode13=5      % ^^M
3  \endlinechar=13 %
4  \catcode123=1     % {
5  \catcode125=2     % }
6  \catcode64=11     % @
7  \catcode35=6      % #
8  \catcode44=12     % ,
9  \catcode45=12     % -
10 \catcode46=12     % .
11 \catcode58=12     % :
12 \def\space { }%
13 \let\z\endgroup
14 \expandafter\let\expandafter\x\csname ver@xintfrac.sty\endcsname
15 \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
16 \expandafter
17  \ifx\csname PackageInfo\endcsname\relax
18    \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
19  \else
20    \def\y#1#2{\PackageInfo{#1}{#2}}%
21  \fi
22 \expandafter
23 \ifx\csname numexpr\endcsname\relax
24   \y{xintfrac}{\numexpr not available, aborting input}%
25   \aftergroup\endinput
26 \else
27   \ifx\x\relax    % plain-TeX, first loading of xintfrac.sty
28     \ifx\w\relax % but xint.sty not yet loaded.
29       \y{xintfrac}{Package xint is required}%
30       \y{xintfrac}{Will try \string\input\space xint.sty}%
31     \def\z{\endgroup\input xint.sty\relax}%

```

```

32     \fi
33   \else
34     \def\empty {}%
35     \ifx\x\empty % LaTeX, first loading,
36     % variable is initialized, but \ProvidesPackage not yet seen
37     \ifx\w\relax % xint.sty not yet loaded.
38       \y{xintfrac}{Package xint is required}%
39       \y{xintfrac}{Will try \string\RequirePackage{xint}}%
40       \def\z{\endgroup\RequirePackage{xint}}%
41     \fi
42   \else
43     \y{xintfrac}{I was already loaded, aborting input}%
44     \aftergroup\endinput
45   \fi
46 \fi
47 \fi
48 \z%
```

37.2 Confirmation of *xint* loading

```

49 \begingroup\catcode61\catcode48\catcode32=10\relax%
50 \catcode13=5 % ^^M
51 \endlinechar=13 %
52 \catcode123=1 % {
53 \catcode125=2 % }
54 \catcode64=11 % @
55 \catcode35=6 % #
56 \catcode44=12 % ,
57 \catcode45=12 % -
58 \catcode46=12 % .
59 \catcode58=12 % :
60 \ifdefined\PackageInfo
61   \def\y#1#2{\PackageInfo{#1}{#2}}%
62 \else
63   \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
64 \fi
65 \def\empty {}%
66 \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
67 \ifx\w\relax % Plain TeX, user gave a file name at the prompt
68   \y{xintfrac}{Loading of package xint failed, aborting input}%
69   \aftergroup\endinput
70 \fi
71 \ifx\w\empty % LaTeX, user gave a file name at the prompt
72   \y{xintfrac}{Loading of package xint failed, aborting input}%
73   \aftergroup\endinput
74 \fi
75 \endgroup%
```

37.3 Catcodes

```
76 \XINTsetupcatcodes%
```

37.4 Package identification

```
77 \XINT_providespackage
78 \ProvidesPackage{xintfrac}%
79 [2013/11/28 v1.09h Expandable operations on fractions (jfb)]%
80 \chardef\xint_c_vi      6
81 \chardef\xint_c_vii     7
82 \chardef\xint_c_xviii  18
83 \mathchardef\xint_c_x^iv 10000
```

37.5 \xintLen

```
84 \def\xintLen {\romannumeral0\xintlen }%
85 \def\xintlen #1%
86 {%
87   \expandafter\xINT_flen\romannumeral0\xINT_infrac {#1}%
88 }%
89 \def\xINT_flen #1#2#3%
90 {%
91   \expandafter\space
92   \the\numexpr -1+\XINT_Abs {#1}+\XINT_Len {#2}+\XINT_Len {#3}\relax
93 }%
```

37.6 \XINT_lenrord_loop

```
94 \def\xINT_lenrord_loop #1#2#3#4#5#6#7#8#9%
95 {%  faire \romannumeral-‘0\xINT_lenrord_loop 0{ }#1\Z\W\W\W\W\W\W\W\Z
96   \xint_gob_til_W #9\xINT_lenrord_W\W
97   \expandafter\xINT_lenrord_loop\expandafter
98   {\the\numexpr #1+7}{#9#8#7#6#5#4#3#2}%
99 }%
100 \def\xINT_lenrord_W\W\expandafter\xINT_lenrord_loop\expandafter #1#2#3\Z
101 {%
102   \expandafter\xINT_lenrord_X\expandafter {#1}#2\Z
103 }%
104 \def\xINT_lenrord_X #1#2\Z
105 {%
106   \XINT_lenrord_Y #2\R\R\R\R\R\R\T {#1}%
107 }%
108 \def\xINT_lenrord_Y #1#2#3#4#5#6#7#8\T
109 {%
110   \xint_gob_til_W
111     #7\xINT_lenrord_Z \xint_c_viii
112     #6\xINT_lenrord_Z \xint_c_vii
113     #5\xINT_lenrord_Z \xint_c_vi
114     #4\xINT_lenrord_Z \xint_c_v
115     #3\xINT_lenrord_Z \xint_c_iv
116     #2\xINT_lenrord_Z \xint_c_iii
117     \W\xINT_lenrord_Z \xint_c_ii \Z
```

```

118 }%
119 \def\XINT_lenrord_Z #1#2\Z #3% retourne: {longueur}renverse\Z
120 {%
121   \expandafter{\the\numexpr #3-#1\relax}%
122 }%

```

37.7 \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*)

```

123 \def\XINT_outfrac #1#2#3%
124 {%
125   \ifcase\XINT_Sgn{#3}
126     \expandafter \XINT_outfrac_divisionbyzero
127   \or
128     \expandafter \XINT_outfrac_P
129   \else
130     \expandafter \XINT_outfrac_N
131   \fi
132   {#2}{#3}[#1]%
133 }%
134 \def\XINT_outfrac_divisionbyzero #1#2{\xintError:DivisionByZero\space #1/0}%
135 \def\XINT_outfrac_P #1#2%
136 {%
137   \ifcase\XINT_Sgn{#1}
138     \expandafter\XINT_outfrac_Zero
139   \fi
140   \space #1/#2%
141 }%
142 \def\XINT_outfrac_Zero #1[#2]{ 0/1[0]}%
143 \def\XINT_outfrac_N #1#2%
144 {%
145   \expandafter\XINT_outfrac_N_a\expandafter
146   {\romannumeral0\XINT_opp #2}{\romannumeral0\XINT_opp #1}%
147 }%
148 \def\XINT_outfrac_N_a #1#2%
149 {%
150   \expandafter\XINT_outfrac_P\expandafter {#2}{#1}%
151 }%

```

37.8 \XINT_inFrac

Extended in 1.07 to accept scientific notation on input. With lowercase e only. The *\xintexpr* parser does accept uppercase E also.

```

152 \def\XINT_inFrac {\romannumeral0\XINT_infrac }%

```

```

153 \def\XINT_infrac #1%
154 {%
155   \expandafter\XINT_infrac_ \romannumeral-‘0#1[\W]\Z\T
156 }%
157 \def\XINT_infrac_ #1[#2#3]#4\Z
158 {%
159   \xint_UDwfork
160     #2\dummy \XINT_infrac_A
161     \W\dummy \XINT_infrac_B
162   \krof
163   #1[#2#3]#4%
164 }%
165 \def\XINT_infrac_A #1[\W]\T
166 {%
167   \XINT_frac #1/\W\Z
168 }%
169 \def\XINT_infrac_B #1%
170 {%
171   \xint_gob_til_zero #1\XINT_infrac_Zero0\XINT_infrac_BB #1%
172 }%
173 \def\XINT_infrac_BB #1[\W]\T {\XINT_infrac_BC #1/\W\Z}%
174 \def\XINT_infrac_BC #1/#2#3\Z
175 {%
176   \xint_UDwfork
177     #2\dummy \XINT_infrac_BCa
178     \W\dummy {\expandafter\XINT_infrac_BCb \romannumeral-‘0#2}%
179   \krof
180   #3\Z #1\Z
181 }%
182 \def\XINT_infrac_BCa \Z #1[#2]#3\Z { {#2}{#1}{1}}%
183 \def\XINT_infrac_BCb #1[#2]/\W\Z #3\Z { {#2}{#3}{#1}}%
184 \def\XINT_infrac_Zero #1\T { {0}{0}{1}}%

```

37.9 \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

```

185 \def\XINT_frac #1/#2#3\Z
186 {%
187   \xint_UDwfork
188     #2\dummy \XINT_frac_A
189     \W\dummy {\expandafter\XINT_frac_U \romannumeral-‘0#2}%
190   \krof
191   #3e\W\Z #1e\W\Z
192 }%
193 \def\XINT_frac_U #1e#2#3\Z
194 {%

```

```

195 \xint_UDwfork
196 #2\dummy \XINT_frac_Ua
197 \W\dummy {\XINT_frac_Ub #2}%
198 \krof
199 #3\Z #1\Z
200 }%
201 \def\XINT_frac_Ua \Z #1\W\Z {\XINT_frac_B #1.\W\Z {0}}%
202 \def\XINT_frac_Ub #1/\W e\W\Z #2\Z {\XINT_frac_B #2.\W\Z {#1}}%
203 \def\XINT_frac_B #1.#2#3\Z
204 {%
205 \xint_UDwfork
206 #2\dummy \XINT_frac_Ba
207 \W\dummy {\XINT_frac_Bb #2}%
208 \krof
209 #3\Z #1\Z
210 }%
211 \def\XINT_frac_Ba \Z #1\Z {\XINT_frac_T {0}{#1}}%
212 \def\XINT_frac_Bb #1.\W\Z #2\Z
213 {%
214 \expandafter \XINT_frac_T \expandafter
215 {\romannumeral0\xintlength {#1}}{#2#1}%
216 }%
217 \def\XINT_frac_A e\W\Z {\XINT_frac_T {0}{1}{0}}%
218 \def\XINT_frac_T #1#2#3#4e#5#6\Z
219 {%
220 \xint_UDwfork
221 #5\dummy \XINT_frac-Ta
222 \W\dummy {\XINT_frac-Tb #5}%
223 \krof
224 #6\Z #4\Z {#1}{#2}{#3}%
225 }%
226 \def\XINT_frac-Ta \Z #1\Z {\XINT_frac_C #1.\W\Z {0}}%
227 \def\XINT_frac-Tb #1e\W\Z #2\Z {\XINT_frac_C #2.\W\Z {#1}}%
228 \def\XINT_frac_C #1.#2#3\Z
229 {%
230 \xint_UDwfork
231 #2\dummy \XINT_frac-Ca
232 \W\dummy {\XINT_frac-Cb #2}%
233 \krof
234 #3\Z #1\Z
235 }%
236 \def\XINT_frac-Ca \Z #1\Z {\XINT_frac_D {0}{#1}}%
237 \def\XINT_frac-Cb #1.\W\Z #2\Z
238 {%
239 \expandafter\XINT_frac_D\expandafter
240 {\romannumeral0\xintlength {#1}}{#2#1}%
241 }%
242 \def\XINT_frac_D #1#2#3#4#5#6%
243 {%

```

```

244 \expandafter \XINT_frac_E \expandafter
245 {\the\numexpr -#1+#3+#4-#6\expandafter}\expandafter
246 {\romannumeral0\XINT_num_loop #2%
247 \xint_relax\xint_relax\xint_relax\xint_relax
248 \xint_relax\xint_relax\xint_relax\xint_relax\Z }%
249 {\romannumeral0\XINT_num_loop #5%
250 \xint_relax\xint_relax\xint_relax\xint_relax
251 \xint_relax\xint_relax\xint_relax\xint_relax\Z }%
252 }%
253 \def\XINT_frac_E #1#2#3%
254 {%
255 \expandafter \XINT_frac_F #3\Z {#2}{#1}%
256 }%
257 \def\XINT_frac_F #1%
258 {%
259 \xint_UDzerominusfork
260 #1-\dummy \XINT_frac_Gdivisionbyzero
261 0#1\dummy \XINT_frac_Gneg
262 0-\dummy {\XINT_frac_Gpos #1}%
263 \krof
264 }%
265 \def\XINT_frac_Gdivisionbyzero #1\Z #2#3%
266 {%
267 \xintError:DivisionByZero\space {0}{#2}{0}%
268 }%
269 \def\XINT_frac_Gneg #1\Z #2#3%
270 {%
271 \expandafter\XINT_frac_H \expandafter{\romannumeral0\XINT_opp #2}{#3}{#1}%
272 }%
273 \def\XINT_frac_H #1#2{ {#2}{#1}}%
274 \def\XINT_frac_Gpos #1\Z #2#3{ {#3}{#2}{#1}}%

```

37.10 \XINT_factortens, \XINT_cuz_cnt

```

275 \def\XINT_factortens #1%
276 {%
277 \expandafter\XINT_cuz_cnt_loop\expandafter
278 {\expandafter}\romannumeral0\XINT_rord_main {}#1%
279 \xint_relax
280 \xint_bye\xint_bye\xint_bye\xint_bye
281 \xint_bye\xint_bye\xint_bye\xint_bye
282 \xint_relax
283 \R\R\R\R\R\R\R\Z
284 }%
285 \def\XINT_cuz_cnt #1%
286 {%
287 \XINT_cuz_cnt_loop {}#1\R\R\R\R\R\R\R\Z
288 }%
289 \def\XINT_cuz_cnt_loop #1#2#3#4#5#6#7#8#9%

```

37 Package *xintfrac* implementation

```

290 {%
291   \xint_gob_til_R #9\XINT_cuz_cnt_toofara \R
292   \expandafter\XINT_cuz_cnt_checka\expandafter
293   {\the\numexpr #1+8\relax}{#2#3#4#5#6#7#8#9}%
294 }%
295 \def\XINT_cuz_cnt_toofara\R
296   \expandafter\XINT_cuz_cnt_checka\expandafter #1#2%
297 {%
298   \XINT_cuz_cnt_toofarb {#1}#2%
299 }%
300 \def\XINT_cuz_cnt_toofarb #1#2\Z {\XINT_cuz_cnt_toofarc #2\Z {#1}}%
301 \def\XINT_cuz_cnt_toofarc #1#2#3#4#5#6#7#8%
302 {%
303   \xint_gob_til_R #2\XINT_cuz_cnt_toofard 7%
304       #3\XINT_cuz_cnt_toofard 6%
305       #4\XINT_cuz_cnt_toofard 5%
306       #5\XINT_cuz_cnt_toofard 4%
307       #6\XINT_cuz_cnt_toofard 3%
308       #7\XINT_cuz_cnt_toofard 2%
309       #8\XINT_cuz_cnt_toofard 1%
310       \Z #1#2#3#4#5#6#7#8%
311 }%
312 \def\XINT_cuz_cnt_toofard #1#2\Z #3\R #4\Z #5%
313 {%
314   \expandafter\XINT_cuz_cnt_toofare
315   \the\numexpr #3\relax \R\R\R\R\R\R\R\R\Z
316   {\the\numexpr #5-#1\relax}\R\Z
317 }%
318 \def\XINT_cuz_cnt_toofare #1#2#3#4#5#6#7#8%
319 {%
320   \xint_gob_til_R #2\XINT_cuz_cnt_stopc 1%
321       #3\XINT_cuz_cnt_stopc 2%
322       #4\XINT_cuz_cnt_stopc 3%
323       #5\XINT_cuz_cnt_stopc 4%
324       #6\XINT_cuz_cnt_stopc 5%
325       #7\XINT_cuz_cnt_stopc 6%
326       #8\XINT_cuz_cnt_stopc 7%
327       \Z #1#2#3#4#5#6#7#8%
328 }%
329 \def\XINT_cuz_cnt_checka #1#2%
330 {%
331   \expandafter\XINT_cuz_cnt_checkb\the\numexpr #2\relax \Z {#1}%
332 }%
333 \def\XINT_cuz_cnt_checkb #1%
334 {%
335   \xint_gob_til_zero #1\expandafter\XINT_cuz_cnt_loop\xint_gob_til_Z
336   0\XINT_cuz_cnt_stopa #1%
337 }%
338 \def\XINT_cuz_cnt_stopa #1\Z

```



```

339 {%
340   \XINT_cuz_cnt_stopb #1\R\R\R\R\R\R\R\R\Z %
341 }%
342 \def\XINT_cuz_cnt_stopb #1#2#3#4#5#6#7#8#9%
343 {%
344   \xint_gob_til_R #2\XINT_cuz_cnt_stopc 1%
345       #3\XINT_cuz_cnt_stopc 2%
346       #4\XINT_cuz_cnt_stopc 3%
347       #5\XINT_cuz_cnt_stopc 4%
348       #6\XINT_cuz_cnt_stopc 5%
349       #7\XINT_cuz_cnt_stopc 6%
350       #8\XINT_cuz_cnt_stopc 7%
351       #9\XINT_cuz_cnt_stopc 8%
352       \Z #1#2#3#4#5#6#7#8#9%
353 }%
354 \def\XINT_cuz_cnt_stopc #1#2\Z #3\R #4\Z #5%
355 {%
356   \expandafter\XINT_cuz_cnt_stopd\expandafter
357   {\the\numexpr #5-#1}\#3%
358 }%
359 \def\XINT_cuz_cnt_stopd #1#2\R #3\Z
360 {%
361   \expandafter\space\expandafter
362   {\romannumeral0\XINT_rord_main }\#2%
363   \xint_relax
364   \xint_bye\xint_bye\xint_bye\xint_bye
365   \xint_bye\xint_bye\xint_bye\xint_bye
366   \xint_relax }\#1}%
367 }%

```

37.11 \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.

```

368 \def\xintRaw {\romannumeral0\xintraw }%
369 \def\xintraw
370 {%
371   \expandafter\XINT_raw\romannumeral0\XINT_infrac
372 }%
373 \def\XINT_raw #1#2#3{ #2/#3[#1]}%

```

37.12 \xintPraw

1.09b: these `[n]`'s and especially the possible `/1` are truly annoying at times.

```

374 \def\xintPraw {\romannumeral0\xintpraw }%
375 \def\xintpraw

```

```

376 {%
377   \expandafter\XINT_praw\romannumeral0\XINT_infrac
378 }%
379 \def\XINT_praw #1%
380 {%
381   \ifnum #1=\xint_c_ \expandafter\XINT_praw_a\fi \XINT_praw_A {#1}%
382 }%
383 \def\XINT_praw_A #1#2#3%
384 {%
385   \if\XINT_isOne{#3}1\expandafter\xint_firstoftwo
386     \else\expandafter\xint_secondoftwo
387   \fi { #2[#1]}{ #2/#3[#1]}%
388 }%
389 \def\XINT_praw_a\XINT_praw_A #1#2#3%
390 {%
391   \if\XINT_isOne{#3}1\expandafter\xint_firstoftwo
392     \else\expandafter\xint_secondoftwo
393   \fi { #2}{ #2/#3}%
394 }%

```

37.13 \xintRawWithZeros

This was called \xintRaw in versions earlier than 1.07

```

395 \def\xintRawWithZeros {\romannumeral0\xintrawwithzeros }%
396 \def\xintrawwithzeros
397 {%
398   \expandafter\XINT_rawz\romannumeral0\XINT_infrac
399 }%
400 \def\XINT_rawz #1%
401 {%
402   \ifcase\XINT_Sgn {#1}
403     \expandafter\XINT_rawz_Ba
404   \or
405     \expandafter\XINT_rawz_A
406   \else
407     \expandafter\XINT_rawz_Ba
408   \fi
409   {#1}%
410 }%
411 \def\XINT_rawz_A #1#2#3{\xint_dsh {#2}{-#1}/#3}%
412 \def\XINT_rawz_Ba #1#2#3{\expandafter\XINT_rawz_Bb
413   \expandafter{\romannumeral0\xint_dsh {#3}{#1}}{#2}}%
414 \def\XINT_rawz_Bb #1#2{ #2/#1}%

```

37.14 \xintFloor

1.09a

```

415 \def\xintFloor {\romannumeral0\xintfloor }%
416 \def\xintfloor #1{\expandafter\XINT_floor
417     \romannumeral0\xintraewithzeros {#1}.}%
418 \def\XINT_floor #1/#2.{\xintiipro {#1}{#2}}%

```

37.15 \xintCeil

1.09a

```

419 \def\xintCeil {\romannumeral0\xintceil }%
420 \def\xintceil #1{\xintiio { \xintFloor {\xintOpp{#1}}}}%

```

37.16 \xintNumerator

```

421 \def\xintNumerator {\romannumeral0\xintnumerator }%
422 \def\xintnumerator
423 {%
424     \expandafter\XINT_numer\romannumeral0\XINT_infrac
425 }%
426 \def\XINT_numer #1%
427 {%
428     \ifcase\XINT_Sgn {#1}
429     \expandafter\XINT_numer_B
430     \or
431     \expandafter\XINT_numer_A
432     \else
433     \expandafter\XINT_numer_B
434     \fi
435     {#1}%
436 }%
437 \def\XINT_numer_A #1#2#3{\xint_dsh {#2}{-#1}}%
438 \def\XINT_numer_B #1#2#3{ #2}%

```

37.17 \xintDenominator

```

439 \def\xintDenominator {\romannumeral0\xintdenominator }%
440 \def\xintdenominator
441 {%
442     \expandafter\XINT_denom\romannumeral0\XINT_infrac
443 }%
444 \def\XINT_denom #1%
445 {%
446     \ifcase\XINT_Sgn {#1}
447     \expandafter\XINT_denom_B
448     \or
449     \expandafter\XINT_denom_A
450     \else
451     \expandafter\XINT_denom_B
452     \fi

```

```

453   {#1}%
454 }%
455 \def\XINT_denom_A #1#2#3{ #3}%
456 \def\XINT_denom_B #1#2#3{\xint_dsh {#3}{#1}}%

```

37.18 \xintFrac

```

457 \def\xintFrac {\romannumeral0\xintfrac }%
458 \def\xintfrac #1%
459 {%
460   \expandafter\XINT_fracfrac_A\romannumeral0\XINT_infrac {#1}%
461 }%
462 \def\XINT_fracfrac_A #1{\XINT_fracfrac_B #1\Z }%
463 \catcode'\^=7
464 \def\XINT_fracfrac_B #1#2\Z
465 {%
466   \xint_gob_til_zero #1\XINT_fracfrac_C 0\XINT_fracfrac_D {10^{#1#2}}%
467 }%
468 \def\XINT_fracfrac_C #1#2#3#4#5%
469 {%
470   \ifcase\XINT_isOne {#5}
471   \or \xint_afterfi {\expandafter\xint_firstoftwo_andstop\xint_gobble_ii }%
472   \fi
473   \space
474   \frac {#4}{#5}%
475 }%
476 \def\XINT_fracfrac_D #1#2#3%
477 {%
478   \ifcase\XINT_isOne {#3}
479   \or \XINT_fracfrac_E
480   \fi
481   \space
482   \frac {#2}{#3}#1%
483 }%
484 \def\XINT_fracfrac_E \fi #1#2#3#4{\fi \space #3\cdot }%

```

37.19 \xintSignedFrac

```

485 \def\xintSignedFrac {\romannumeral0\xintsignedfrac }%
486 \def\xintsignedfrac #1%
487 {%
488   \expandafter\XINT_sgnfrac_a\romannumeral0\XINT_infrac {#1}%
489 }%
490 \def\XINT_sgnfrac_a #1#2%
491 {%
492   \XINT_sgnfrac_b #2\Z {#1}%
493 }%
494 \def\XINT_sgnfrac_b #1%
495 {%
496   \xint_UDsignfork
497   #1\dummy \XINT_sgnfrac_N

```

```

498     -\dummy {\XINT_sgnfrac_P #1}%
499     \krof
500 }%
501 \def\XINT_sgnfrac_P #1\Z #2%
502 {%
503     \XINT_fracfrac_A {#2}{#1}%
504 }%
505 \def\XINT_sgnfrac_N
506 {%
507     \expandafter\xint_minus_andstop\romannumeral0\XINT_sgnfrac_P
508 }%

```

37.20 \xintFwOver

```

509 \def\xintFwOver {\romannumeral0\xintfwover }%
510 \def\xintfwover #1%
511 {%
512     \expandafter\XINT_fwover_A\romannumeral0\XINT_infrac {#1}%
513 }%
514 \def\XINT_fwover_A #1{\XINT_fwover_B #1\Z }%
515 \def\XINT_fwover_B #1#2\Z
516 {%
517     \xint_gob_til_zero #1\XINT_fwover_C 0\XINT_fwover_D {10^{#1#2}}%
518 }%
519 \catcode'\^=11
520 \def\XINT_fwover_C #1#2#3#4#5%
521 {%
522     \ifcase\XINT_isOne {#5}
523         \xint_afterfi { {#4\over #5}}%
524     \or
525         \xint_afterfi { #4}%
526     \fi
527 }%
528 \def\XINT_fwover_D #1#2#3%
529 {%
530     \ifcase\XINT_isOne {#3}
531         \xint_afterfi { {#2\over #3}}%
532     \or
533         \xint_afterfi { #2\cdot }%
534     \fi
535     #1%
536 }%

```

37.21 \xintSignedFwOver

```

537 \def\xintSignedFwOver {\romannumeral0\xintsignedfwover }%
538 \def\xintsignedfwover #1%
539 {%
540     \expandafter\XINT_sgnfwover_a\romannumeral0\XINT_infrac {#1}%
541 }%
542 \def\XINT_sgnfwover_a #1#2%

```

```

543 {%
544   \XINT_sgnfwover_b #2\Z {#1}%
545 }%
546 \def\XINT_sgnfwover_b #1%
547 {%
548   \xint_UDsignfork
549     #1\dummy \XINT_sgnfwover_N
550     -\dummy {\XINT_sgnfwover_P #1}%
551   \krof
552 }%
553 \def\XINT_sgnfwover_P #1\Z #2%
554 {%
555   \XINT_fwover_A {#2}{#1}%
556 }%
557 \def\XINT_sgnfwover_N
558 {%
559   \expandafter\xint_minus_andstop\romannumeral0\XINT_sgnfwover_P
560 }%

```

37.22 \xintREZ

```

561 \def\xintREZ {\romannumeral0\xintrez }%
562 \def\xintrez
563 {%
564   \expandafter\XINT_rez_A\romannumeral0\XINT_infrac
565 }%
566 \def\XINT_rez_A #1#2%
567 {%
568   \XINT_rez_AB #2\Z {#1}%
569 }%
570 \def\XINT_rez_AB #1%
571 {%
572   \xint_UDzerominusfork
573     #1-\dummy \XINT_rez_zero
574     0#1\dummy \XINT_rez_neg
575     0-\dummy {\XINT_rez_B #1}%
576   \krof
577 }%
578 \def\XINT_rez_zero #1\Z #2#3{ 0/1[0]}%
579 \def\XINT_rez_neg {\expandafter\xint_minus_andstop\romannumeral0\XINT_rez_B }%
580 \def\XINT_rez_B #1\Z
581 {%
582   \expandafter\XINT_rez_C\romannumeral0\XINT_factortens {#1}%
583 }%
584 \def\XINT_rez_C #1#2#3#4%
585 {%
586   \expandafter\XINT_rez_D\romannumeral0\XINT_factortens {#4}{#3}{#2}{#1}%
587 }%
588 \def\XINT_rez_D #1#2#3#4#5%
589 {%

```

```

590 \expandafter\XINT_rez_E\expandafter
591 {\the\numexpr #3+#4-#2}{#1}{#5}%
592 }%
593 \def\XINT_rez_E #1#2#3{ #3/#2[#1]}%

```

37.23 \xintE

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

```

594 \def\xintE {\romannumeral0\xinte }%
595 \def\xinte #1%
596 {%
597 \expandafter\XINT_e \romannumeral0\XINT_infrac {#1}%
598 }%
599 \def\XINT_e #1#2#3#4%
600 {%
601 \expandafter\XINT_e_end\expandafter{\the\numexpr #1+#4}{#2}{#3}%
602 }%
603 \def\xintfE {\romannumeral0\xintfe }%
604 \def\xintfe #1%
605 {%
606 \expandafter\XINT_fe \romannumeral0\XINT_infrac {#1}%
607 }%
608 \def\XINT_fe #1#2#3#4%
609 {%
610 \expandafter\XINT_e_end\expandafter{\the\numexpr #1+\xintNum{#4}}{#2}{#3}%
611 }%
612 \def\XINT_e_end #1#2#3{ #2/#3[#1]}%
613 \let\XINTinFloatfE\xintfE

```

37.24 \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.

```

614 \def\xintIrr {\romannumeral0\xintirr }%
615 \def\xintirr #1%
616 {%
617 \expandafter\XINT_irr_start\romannumeral0\xintrawwithzeros {#1}\Z
618 }%
619 \def\XINT_irr_start #1#2/#3\Z
620 {%
621 \ifcase\XINT_isOne {#3}
622 \xint_afterfi
623 {\xint_UDsignfork
624 #1\dummy \XINT_irr_negative
625 -\dummy {\XINT_irr_nonneg #1}%

```

```

626         \krof}%
627     \or
628         \xint_afterfi{\XINT_irr_denomisine #1}%
629     \fi
630     #2\Z {#3}%
631 }%
632 \def\XINT_irr_denomisine #1\Z #2{ #1/1}% changed in 1.08
633 \def\XINT_irr_negative #1\Z #2{\XINT_irr_D #1\Z #2\Z \xint_minus_andstop}%
634 \def\XINT_irr_nonneg #1\Z #2{\XINT_irr_D #1\Z #2\Z \space}%
635 \def\XINT_irr_D #1#2\Z #3#4\Z
636 {%
637     \xint_UDzerosfork
638     #3#1\dummy \XINT_irr_indeterminate
639     #30\dummy \XINT_irr_divisionbyzero
640     #10\dummy \XINT_irr_zero
641     00\dummy \XINT_irr_loop_a
642     \krof
643     {#3#4}{#1#2}{#3#4}{#1#2}%
644 }%
645 \def\XINT_irr_indeterminate #1#2#3#4#5{\xintError:NaN\space 0/0}%
646 \def\XINT_irr_divisionbyzero #1#2#3#4#5{\xintError:DivisionByZero #5#2/0}%
647 \def\XINT_irr_zero #1#2#3#4#5{ 0/1}% changed in 1.08
648 \def\XINT_irr_loop_a #1#2%
649 {%
650     \expandafter\XINT_irr_loop_d
651     \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
652 }%
653 \def\XINT_irr_loop_d #1#2%
654 {%
655     \XINT_irr_loop_e #2\Z
656 }%
657 \def\XINT_irr_loop_e #1#2\Z
658 {%
659     \xint_gob_til_zero #1\xint_irr_loop_exit0\XINT_irr_loop_a {#1#2}%
660 }%
661 \def\xint_irr_loop_exit0\XINT_irr_loop_a #1#2#3#4%
662 {%
663     \expandafter\XINT_irr_loop_exitb\expandafter
664     {\romannumeral0\xinti quo {#3}{#2}}%
665     {\romannumeral0\xinti quo {#4}{#2}}%
666 }%
667 \def\XINT_irr_loop_exitb #1#2%
668 {%
669     \expandafter\XINT_irr_finish\expandafter {#2}{#1}%
670 }%
671 \def\XINT_irr_finish #1#2#3{#3#1/#2}% changed in 1.08

```


37.25 \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.

```

672 \def\xintNum {\romannumeral0\xintnum }%
673 \def\xintnum #1{\expandafter\XINT_intcheck\romannumeral0\xintirr {#1}\Z }%
674 \def\XINT_intcheck #1/#2\Z
675 {%
676   \ifcase\XINT_isOne {#2}
677     \xintError:NotAnInteger
678   \fi\space #1%
679 }%

```

37.26 \xintifInt

1.09e. xintfrac.sty only

```

680 \def\xintifInt {\romannumeral0\xintifint }%
681 \def\xintifint #1{\expandafter\XINT_ifint\romannumeral0\xintirr {#1}\Z }%
682 \def\XINT_ifint #1/#2\Z
683 {%
684   \if\XINT_isOne {#2}1%
685   \xint_afterfi{\expandafter\space\xint_firstoftwo}%
686   \else
687   \xint_afterfi{\expandafter\space\xint_secondoftwo}%
688   \fi
689 }%

```

37.27 \xintJrr

Modified similarly as \xintIrr in release 1.05. 1.08 version does not remove a /1 denominator.

```

690 \def\xintJrr {\romannumeral0\xintjrr }%
691 \def\xintjrr #1%
692 {%
693   \expandafter\XINT_jrr_start\romannumeral0\xintrawwithzeros {#1}\Z
694 }%
695 \def\XINT_jrr_start #1#2/#3\Z
696 {%
697   \ifcase\XINT_isOne {#3}
698     \xint_afterfi
699     {\xint_UDsignfork
700      #1\dummy \XINT_jrr_negative

```

37 Package *xintfrac* implementation

```

701          -\dummy {\XINT_jrr_nonneg #1}%
702          \krof}%
703      \or
704      \xint_afterfi{\XINT_jrr_denomisine #1}%
705      \fi
706      #2\Z {#3}%
707 }%
708 \def\XINT_jrr_denomisine #1\Z #2{ #1/1}% changed in 1.08
709 \def\XINT_jrr_negative #1\Z #2{\XINT_jrr_D #1\Z #2\Z \xint_minus_andstop }%
710 \def\XINT_jrr_nonneg #1\Z #2{\XINT_jrr_D #1\Z #2\Z \space}%
711 \def\XINT_jrr_D #1#2\Z #3#4\Z
712 {%
713     \xint_UDzerosfork
714     #3#1\dummy \XINT_jrr_indeterminate
715     #30\dummy \XINT_jrr_divisionbyzero
716     #10\dummy \XINT_jrr_zero
717     00\dummy \XINT_jrr_loop_a
718     \krof
719     {#3#4}{#1#2}1001%
720 }%
721 \def\XINT_jrr_indeterminate #1#2#3#4#5#6#7{\xintError:NaN\space 0/0}%
722 \def\XINT_jrr_divisionbyzero #1#2#3#4#5#6#7{\xintError:DivisionByZero #7#2/0}%
723 \def\XINT_jrr_zero #1#2#3#4#5#6#7{ 0/1}% changed in 1.08
724 \def\XINT_jrr_loop_a #1#2%
725 {%
726     \expandafter\XINT_jrr_loop_b
727     \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
728 }%
729 \def\XINT_jrr_loop_b #1#2#3#4#5#6#7%
730 {%
731     \expandafter \XINT_jrr_loop_c \expandafter
732     {\romannumeral0\xintiiadd{\XINT_Mul{#4}{#1}}{#6}}%
733     {\romannumeral0\xintiiadd{\XINT_Mul{#5}{#1}}{#7}}%
734     {#2}{#3}{#4}{#5}%
735 }%
736 \def\XINT_jrr_loop_c #1#2%
737 {%
738     \expandafter \XINT_jrr_loop_d \expandafter{#2}{#1}%
739 }%
740 \def\XINT_jrr_loop_d #1#2#3#4%
741 {%
742     \XINT_jrr_loop_e #3\Z {#4}{#2}{#1}%
743 }%
744 \def\XINT_jrr_loop_e #1#2\Z
745 {%
746     \xint_gob_til_zero #1\xint_jrr_loop_exit0\XINT_jrr_loop_a {#1#2}%
747 }%
748 \def\xint_jrr_loop_exit0\XINT_jrr_loop_a #1#2#3#4#5#6%
749 {%

```

```

750 \XINT_irr_finish {#3}{#4}%
751 }%

```

37.28 \xintTrunc, \xintiTrunc

Modified in 1.06 to give the first argument to a \numexpr. 1.09f fixes the overhead added in 1.09a to some inner routines when \xintquo was redefined to use \xintnum, whereas it should not. Now called \xinti quo, by the way.

```

752 \def\xintTrunc {\romannumeral0\xinttrunc }%
753 \def\xintiTrunc {\romannumeral0\xintitrunc }%
754 \def\xinttrunc #1%
755 {%
756   \expandafter\XINT_trunc\expandafter {\the\numexpr #1}%
757 }%
758 \def\XINT_trunc #1#2%
759 {%
760   \expandafter\XINT_trunc_G
761   \romannumeral0\expandafter\XINT_trunc_A
762   \romannumeral0\XINT_infrac {#2}{#1}{#1}%
763 }%
764 \def\xintitrunc #1%
765 {%
766   \expandafter\XINT_itrunc\expandafter {\the\numexpr #1}%
767 }%
768 \def\XINT_itrunc #1#2%
769 {%
770   \expandafter\XINT_itrunc_G
771   \romannumeral0\expandafter\XINT_trunc_A
772   \romannumeral0\XINT_infrac {#2}{#1}{#1}%
773 }%
774 \def\XINT_trunc_A #1#2#3#4%
775 {%
776   \expandafter\XINT_trunc_checkifzero
777   \expandafter{\the\numexpr #1+#4}#2\Z {#3}%
778 }%
779 \def\XINT_trunc_checkifzero #1#2#3\Z
780 {%
781   \xint_gob_til_zero #2\XINT_trunc_iszero0\XINT_trunc_B {#1}{#2#3}%
782 }%
783 \def\XINT_trunc_iszero #1#2#3#4#5{ 0\Z 0}%
784 \def\XINT_trunc_B #1%
785 {%
786   \ifcase\XINT_Sgn {#1}
787     \expandafter\XINT_trunc_D
788   \or
789     \expandafter\XINT_trunc_D
790   \else
791     \expandafter\XINT_trunc_C

```

```

792 \fi
793 {#1}%
794 }%
795 \def\XINT_trunc_C #1#2#3%
796 {%
797 \expandafter \XINT_trunc_E
798 \romannumeral0\xint_dsh {#3}{#1}\Z #2\Z
799 }%
800 \def\XINT_trunc_D #1#2%
801 {%
802 \expandafter \XINT_trunc_DE \expandafter
803 {\romannumeral0\xint_dsh {#2}{-#1}}%
804 }%
805 \def\XINT_trunc_DE #1#2{\XINT_trunc_E #2\Z #1\Z }%
806 \def\XINT_trunc_E #1#2\Z #3#4\Z
807 {%
808 \xint_UDsignsfork
809 #1#3\dummy \XINT_trunc_minusminus
810 #1-\dummy {\XINT_trunc_minusplus #3}%
811 #3-\dummy {\XINT_trunc_plusminus #1}%
812 --\dummy {\XINT_trunc_plusplus #3#1}%
813 \krof
814 {#4}{#2}%
815 }%
816 \def\XINT_trunc_minusminus #1#2{\xintiigo {#1}{#2}\Z \space}%
817 \def\XINT_trunc_minusplus #1#2#3{\xintiigo {#1#2}{#3}\Z \xint_minus_andstop}%
818 \def\XINT_trunc_plusminus #1#2#3{\xintiigo {#2}{#1#3}\Z \xint_minus_andstop}%
819 \def\XINT_trunc_plusplus #1#2#3#4{\xintiigo {#1#3}{#2#4}\Z \space}%
820 \def\XINT_itrunc_G #1#2\Z #3#4%
821 {%
822 \xint_gob_til_zero #1\XINT_trunc_zero 0\xint_firstoftwo {#3#1#2}0%
823 }%
824 \def\XINT_trunc_G #1\Z #2#3%
825 {%
826 \xint_gob_til_zero #2\XINT_trunc_zero 0%
827 \expandafter\XINT_trunc_H\expandafter
828 {\the\numexpr\romannumeral0\xintlength {#1}-#3}{#3}{#1}#2%
829 }%
830 \def\XINT_trunc_zero 0#10{ 0}%
831 \def\XINT_trunc_H #1#2%
832 {%
833 \ifnum #1 > 0
834 \xint_afterfi {\XINT_trunc_Ha {#2}}%
835 \else
836 \xint_afterfi {\XINT_trunc_Hb {-#1}}% -0,--1,--2, ....
837 \fi
838 }%
839 \def\XINT_trunc_Ha
840 {%

```

```

841 \expandafter\XINT_trunc_Haa\romannumeral0\xintdecsplit
842 }%
843 \def\XINT_trunc_Haa #1#2#3%
844 {%
845     #3#1.#2%
846 }%
847 \def\XINT_trunc_Hb #1#2#3%
848 {%
849     \expandafter #3\expandafter0\expandafter.%
850     \romannumeral0\XINT_dsx_zero loop {#1}{}Z {}#2% #1=-0 possible!
851 }%

```

37.29 \xintRound, \xintiRound

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

```

852 \def\xintRound {\romannumeral0\xintround }%
853 \def\xintiRound {\romannumeral0\xintiround }%
854 \def\xintround #1%
855 {%
856     \expandafter\XINT_round\expandafter {\the\numexpr #1}%
857 }%
858 \def\XINT_round
859 {%
860     \expandafter\XINT_trunc_G\romannumeral0\XINT_round_A
861 }%
862 \def\xintiround #1%
863 {%
864     \expandafter\XINT_iround\expandafter {\the\numexpr #1}%
865 }%
866 \def\XINT_iround
867 {%
868     \expandafter\XINT_itrunc_G\romannumeral0\XINT_round_A
869 }%
870 \def\XINT_round_A #1#2%
871 {%
872     \expandafter\XINT_round_B
873     \romannumeral0\expandafter\XINT_trunc_A
874     \romannumeral0\XINT_infrac {#2}{\the\numexpr #1+1\relax}{#1}%
875 }%
876 \def\XINT_round_B #1\Z
877 {%
878     \expandafter\XINT_round_C
879     \romannumeral0\XINT_rord_main {}#1%
880     \xint_relax
881     \xint_bye\xint_bye\xint_bye\xint_bye
882     \xint_bye\xint_bye\xint_bye\xint_bye
883     \xint_relax
884     \Z

```

```

885 }%
886 \def\XINT_round_C #1%
887 {%
888   \ifnum #1<5
889     \expandafter\XINT_round_Daa
890   \else
891     \expandafter\XINT_round_Dba
892   \fi
893 }%
894 \def\XINT_round_Daa #1%
895 {%
896   \xint_gob_til_Z #1\XINT_round_Daz\Z \XINT_round_Da #1%
897 }%
898 \def\XINT_round_Daz\Z \XINT_round_Da \Z { 0\Z }%
899 \def\XINT_round_Da #1\Z
900 {%
901   \XINT_rord_main {}#1%
902   \xint_relax
903   \xint_bye\xint_bye\xint_bye\xint_bye
904   \xint_bye\xint_bye\xint_bye\xint_bye
905   \xint_relax \Z
906 }%
907 \def\XINT_round_Dba #1%
908 {%
909   \xint_gob_til_Z #1\XINT_round_Dbz\Z \XINT_round_Db #1%
910 }%
911 \def\XINT_round_Dbz\Z \XINT_round_Db \Z { 1\Z }%
912 \def\XINT_round_Db #1\Z
913 {%
914   \XINT_addm_A 0{}1000\W\X\Y\Z #1000\W\X\Y\Z \Z
915 }%

```

37.30 \xintRound:csv

1.09a. For use by \xintthenumexpr.

```

916 \def\xintRound:csv #1{\expandafter\XINT_round:_a\romannumeral-'0#1,,^}%
917 \def\XINT_round:_a {\XINT_round:_b {}}%
918 \def\XINT_round:_b #1#2,%
919   {\expandafter\XINT_round:_c\romannumeral-'0#2,{#1}}%
920 \def\XINT_round:_c #1{\if #1,\expandafter\XINT_round:_f
921   \else\expandafter\XINT_round:_d\fi #1}%
922 \def\XINT_round:_d #1,%
923   {\expandafter\XINT_round:_e\romannumeral0\xintiround 0{#1},}%
924 \def\XINT_round:_e #1,#2{\XINT_round:_b {#2,#1}}%
925 \def\XINT_round:_f ,#1#2^{\xint_gobble_i #1}%

```

37.31 \xintDigits

The `mathchardef` used to be called `\XINT_digits`, but for reasons originating in `\xintNewExpr`, release 1.09a uses `\XINTdigits` without underscore.

```

926 \mathchardef\XINTdigits 16
927 \def\xintDigits #1#2%
928   {\afterassignment \xint_gobble_i \mathchardef\XINTdigits=}%
929 \def\xinttheDigits {\number\XINTdigits }%
```

37.32 \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`. Here again some inner macros used the `\xintquo` with extra `\xintnum` overhead in 1.09a, reverted in 1.09f.

```

930 \def\xintFloat   {\romannumeral0\xintfloat }%
931 \def\xintfloat #1{\XINT_float_chkopt #1\Z }%
932 \def\XINT_float_chkopt #1%
933 {%
934   \ifx [#1\expandafter\XINT_float_opt
935     \else\expandafter\XINT_float_noopt
936   \fi #1%
937 }%
938 \def\XINT_float_noopt #1\Z
939 {%
940   \expandafter\XINT_float_a\expandafter\XINTdigits
941   \romannumeral0\XINT_infrac {#1}\XINT_float_Q
942 }%
943 \def\XINT_float_opt [\Z #1]#2%
944 {%
945   \expandafter\XINT_float_a\expandafter
946   {\the\numexpr #1\expandafter}%
947   \romannumeral0\XINT_infrac {#2}\XINT_float_Q
948 }%
949 \def\XINT_float_a #1#2#3% #1=P, #2=n, #3=A, #4=B
950 {%
951   \XINT_float_fork #3\Z {#1}{#2}% #1 = precision, #2=n
952 }%
953 \def\XINT_float_fork #1%
954 {%
955   \xint_UDzerominusfork
956   #1-\dummy \XINT_float_zero
957   0#1\dummy \XINT_float_J
958   0-\dummy {\XINT_float_K #1}%
959   \krof
```

```

960 }%
961 \def\XINT_float_zero #1\Z #2#3#4#5{ 0.e0}%
962 \def\XINT_float_J {\expandafter\xint_minus_andstop\romannumeral0\XINT_float_K }%
963 \def\XINT_float_K #1\Z #2% #1=A, #2=P, #3=n, #4=B
964 {%
965   \expandafter\XINT_float_L\expandafter
966   {\the\numexpr\xintLength{#1}\expandafter}\expandafter
967   {\the\numexpr #2+\xint_c_ii}{#1}{#2}%
968 }%
969 \def\XINT_float_L #1#2%
970 {%
971   \ifnum #1>#2
972     \expandafter\XINT_float_Ma
973   \else
974     \expandafter\XINT_float_Mc
975   \fi {#1}{#2}%
976 }%
977 \def\XINT_float_Ma #1#2#3%
978 {%
979   \expandafter\XINT_float_Mb\expandafter
980   {\the\numexpr #1-#2\expandafter\expandafter\expandafter}%
981   \expandafter\expandafter\expandafter
982   {\expandafter\xint_firstoftwo
983     \romannumeral0\XINT_split_fromleft_loop {#2}{#3}\W\W\W\W\W\W\W\W\Z
984     }{#2}%
985 }%
986 \def\XINT_float_Mb #1#2#3#4#5#6% #2=A', #3=P+2, #4=P, #5=n, #6=B
987 {%
988   \expandafter\XINT_float_N\expandafter
989   {\the\numexpr\xintLength{#6}\expandafter}\expandafter
990   {\the\numexpr #3\expandafter}\expandafter
991   {\the\numexpr #1+#5}%
992   {#6}{#3}{#2}{#4}%
993 }% long de B, P+2, n', B, |A'|=P+2, A', P
994 \def\XINT_float_Mc #1#2#3#4#5#6%
995 {%
996   \expandafter\XINT_float_N\expandafter
997   {\romannumeral0\xintlength{#6}}{#2}{#5}{#6}{#1}{#3}{#4}%
998 }% long de B, P+2, n, B, |A|, A, P
999 \def\XINT_float_N #1#2%
1000 {%
1001   \ifnum #1>#2
1002     \expandafter\XINT_float_O
1003   \else
1004     \expandafter\XINT_float_P
1005   \fi {#1}{#2}%
1006 }%
1007 \def\XINT_float_O #1#2#3#4%
1008 {%

```



```

1009 \expandafter\XINT_float_P\expandafter
1010 {\the\numexpr #2\expandafter}\expandafter
1011 {\the\numexpr #2\expandafter}\expandafter
1012 {\the\numexpr #3-#1+#2\expandafter\expandafter\expandafter}%
1013 \expandafter\expandafter\expandafter
1014 {\expandafter\xint_firstoftwo
1015 \romannumeral0\XINT_split_fromleft_loop {#2}{#4\W\W\W\W\W\W\W\W\Z
1016 }%
1017}% |B|,P+2,n,B,|A|,A,P
1018\def\XINT_float_P #1#2#3#4#5#6#7#8%
1019{%
1020 \expandafter #8\expandafter {\the\numexpr #1-#5+#2-\xint_c_i}%
1021 {#6}{#4}{#7}{#3}%
1022}% |B|-|A|+P+1,A,B,P,n
1023\def\XINT_float_Q #1%
1024{%
1025 \ifnum #1<\xint_c_
1026 \expandafter\XINT_float_Ri
1027 \else
1028 \expandafter\XINT_float_Rii
1029 \fi {#1}%
1030}%
1031\def\XINT_float_Ri #1#2#3%
1032{%
1033 \expandafter\XINT_float_Sa
1034 \romannumeral0\xintiique {#2}%
1035 {\XINT_dsx_addzerosnofuss {-#1}{#3}}\Z {#1}%
1036}%
1037\def\XINT_float_Rii #1#2#3%
1038{%
1039 \expandafter\XINT_float_Sa
1040 \romannumeral0\xintiique
1041 {\XINT_dsx_addzerosnofuss {#1}{#2}}{#3}\Z {#1}%
1042}%
1043\def\XINT_float_Sa #1%
1044{%
1045 \if #19%
1046 \xint_afterfi {\XINT_float_Sb\XINT_float_Wb}%
1047 \else
1048 \xint_afterfi {\XINT_float_Sb\XINT_float_Wa}%
1049 \fi #1%
1050}%
1051\def\XINT_float_Sb #1#2\Z #3#4%
1052{%
1053 \expandafter\XINT_float_T\expandafter
1054 {\the\numexpr #4+\xint_c_i\expandafter}%
1055 \romannumeral-'\0\XINT_lenrord_loop 0{}#2\Z\W\W\W\W\W\W\W\Z #1{#3}{#4}%
1056}%
1057\def\XINT_float_T #1#2#3%

```

```

1058 {%
1059   \ifnum #2>#1
1060     \xint_afterfi{\XINT_float_U\XINT_float_Xb}%
1061   \else
1062     \xint_afterfi{\XINT_float_U\XINT_float_Xa #3}%
1063   \fi
1064 }%
1065 \def\XINT_float_U #1#2%
1066 {%
1067   \ifnum #2<\xint_c_v
1068     \expandafter\XINT_float_Va
1069   \else
1070     \expandafter\XINT_float_Vb
1071   \fi #1%
1072 }%
1073 \def\XINT_float_Va #1#2\Z #3%
1074 {%
1075   \expandafter#1%
1076   \romannumeral0\expandafter\XINT_float_Wa
1077   \romannumeral0\XINT_rord_main {}#2%
1078   \xint_relax
1079   \xint_bye\xint_bye\xint_bye\xint_bye
1080   \xint_bye\xint_bye\xint_bye\xint_bye
1081   \xint_relax \Z
1082 }%
1083 \def\XINT_float_Vb #1#2\Z #3%
1084 {%
1085   \expandafter #1%
1086   \romannumeral0\expandafter #3%
1087   \romannumeral0\XINT_addm_A 0{ }1000\W\X\Y\Z #2000\W\X\Y\Z \Z
1088 }%
1089 \def\XINT_float_Wa #1{ #1.}%
1090 \def\XINT_float_Wb #1#2%
1091   {\if #11\xint_afterfi{ 10.}\else\xint_afterfi{ #1.#2}\fi }%
1092 \def\XINT_float_Xa #1\Z #2#3#4%
1093 {%
1094   \expandafter\XINT_float_Y\expandafter
1095   {\the\numexpr #3+#4-#2}{#1}%
1096 }%
1097 \def\XINT_float_Xb #1\Z #2#3#4%
1098 {%
1099   \expandafter\XINT_float_Y\expandafter
1100   {\the\numexpr #3+#4+\xint_c_i-#2}{#1}%
1101 }%
1102 \def\XINT_float_Y #1#2{ #2e#1}%

```

37.33 \xintFloat:csv

1.09a. For use by \xintthefloatexpr.

```

1103 \def\xintFloat:csv #1{\expandafter\XINT_float:_a\romannumeral-‘0#1,,^}%
1104 \def\XINT_float:_a {\XINT_float:_b {}}}%
1105 \def\XINT_float:_b #1#2,%
1106         {\expandafter\XINT_float:_c\romannumeral-‘0#2,{#1}}}%
1107 \def\XINT_float:_c #1{\if #1,\expandafter\XINT_float:_f
1108         \else\expandafter\XINT_float:_d\fi #1}%
1109 \def\XINT_float:_d #1,%
1110         {\expandafter\XINT_float:_e\romannumeral0\xintfloat {#1},}%
1111 \def\XINT_float:_e #1,#2{\XINT_float:_b {#2,#1}}}%
1112 \def\XINT_float:_f ,#1#2^{\xint_gobble_i #1}%

```

37.34 \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° 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.

Here again some inner macros used the `\xintiquo` with extra `\xintnum` overhead in 1.09a, reverted in 1.09f.

```

1113 \def\XINTinFloat {\romannumeral0\XINT_inFloat }%
1114 \def\XINT_inFloat [#1]#2%
1115 {%
1116     \expandafter\XINT_infloat_a\expandafter
1117     {\the\numexpr #1\expandafter}%
1118     \romannumeral0\XINT_infrac {#2}\XINT_infloat_Q
1119 }%
1120 \def\XINT_infloat_a #1#2#3% #1=P, #2=n, #3=A, #4=B
1121 {%
1122     \XINT_infloat_fork #3\Z {#1}{#2}% #1 = precision, #2=n
1123 }%
1124 \def\XINT_infloat_fork #1%
1125 {%
1126     \xint_UDzerominusfork
1127     #1-\dummy \XINT_infloat_zero
1128     0#1\dummy \XINT_infloat_J
1129     0-\dummy {\XINT_float_K #1}%
1130     \krof
1131 }%
1132 \def\XINT_infloat_zero #1\Z #2#3#4#5{ 0[0]}%
1133 \def\XINT_infloat_J {\expandafter-\romannumeral0\XINT_float_K }%
1134 \def\XINT_infloat_Q #1%
1135 {%
1136     \ifnum #1<\xint_c_
1137         \expandafter\XINT_infloat_Ri

```

```

1138 \else
1139 \expandafter\XINT_infloat_Rii
1140 \fi {#1}%
1141 }%
1142 \def\XINT_infloat_Ri #1#2#3%
1143 {%
1144 \expandafter\XINT_infloat_S\expandafter
1145 {\romannumeral0\xintiigo {#2}%
1146 {\XINT_dsx_addzerosnofuss {-#1}{#3}}}{#1}%
1147 }%
1148 \def\XINT_infloat_Rii #1#2#3%
1149 {%
1150 \expandafter\XINT_infloat_S\expandafter
1151 {\romannumeral0\xintiigo
1152 {\XINT_dsx_addzerosnofuss {#1}{#2}}{#3}}{#1}%
1153 }%
1154 \def\XINT_infloat_S #1#2#3%
1155 {%
1156 \expandafter\XINT_infloat_T\expandafter
1157 {\the\numexpr #3+\xint_c_i\expandafter}%
1158 \romannumeral-'\XINT_lenrord_loop 0{}#1\Z\W\W\W\W\W\W\W\W\Z
1159 {#2}%
1160 }%
1161 \def\XINT_infloat_T #1#2#3%
1162 {%
1163 \ifnum #2>#1
1164 \xint_afterfi{\XINT_infloat_U\XINT_infloat_Wb}%
1165 \else
1166 \xint_afterfi{\XINT_infloat_U\XINT_infloat_Wa #3}%
1167 \fi
1168 }%
1169 \def\XINT_infloat_U #1#2%
1170 {%
1171 \ifnum #2<\xint_c_v
1172 \expandafter\XINT_infloat_Va
1173 \else
1174 \expandafter\XINT_infloat_Vb
1175 \fi #1%
1176 }%
1177 \def\XINT_infloat_Va #1#2\Z
1178 {%
1179 \expandafter#1%
1180 \romannumeral0\XINT_rord_main {}#2%
1181 \xint_relax
1182 \xint_bye\xint_bye\xint_bye\xint_bye
1183 \xint_bye\xint_bye\xint_bye\xint_bye
1184 \xint_relax \Z
1185 }%
1186 \def\XINT_infloat_Vb #1#2\Z

```

```

1187 {%
1188   \expandafter #1%
1189   \romannumeral0\XINT_addm_A 0{1000\W\X\Y\Z #2000\W\X\Y\Z \Z
1190 }%
1191 \def\XINT_infloat_Wa #1\Z #2#3%
1192 {%
1193   \expandafter\XINT_infloat_X\expandafter
1194   {\the\numexpr #3+\xint_c_i-#2}{#1}%
1195 }%
1196 \def\XINT_infloat_Wb #1\Z #2#3%
1197 {%
1198   \expandafter\XINT_infloat_X\expandafter
1199   {\the\numexpr #3+\xint_c_ii-#2}{#1}%
1200 }%
1201 \def\XINT_infloat_X #1#2{ #2[#1]}%

```

37.35 \xintAdd

```

1202 \def\xintAdd {\romannumeral0\xintadd }%
1203 \def\xintadd #1%
1204 {%
1205   \expandafter\xint_fadd\expandafter {\romannumeral0\XINT_infrac {#1}}%
1206 }%
1207 \def\xint_fadd #1#2{\expandafter\XINT_fadd_A\romannumeral0\XINT_infrac{#2}#1}%
1208 \def\XINT_fadd_A #1#2#3#4%
1209 {%
1210   \ifnum #4 > #1
1211     \xint_afterfi {\XINT_fadd_B {#1}}%
1212   \else
1213     \xint_afterfi {\XINT_fadd_B {#4}}%
1214   \fi
1215   {#1}{#4}{#2}{#3}%
1216 }%
1217 \def\XINT_fadd_B #1#2#3#4#5#6#7%
1218 {%
1219   \expandafter\XINT_fadd_C\expandafter
1220   {\romannumeral0\xintiimul {#7}{#5}}%
1221   {\romannumeral0\xintiimul
1222   {\romannumeral0\xintiimul {\xintDSH {\the\numexpr -#3+#1\relax}{#6}}{#5}}%
1223   {\romannumeral0\xintiimul {#7}{\xintDSH {\the\numexpr -#2+#1\relax}{#4}}}%
1224   }%
1225   {#1}%
1226 }%
1227 \def\XINT_fadd_C #1#2#3%
1228 {%
1229   \expandafter\XINT_fadd_D\expandafter {#2}{#3}{#1}%
1230 }%
1231 \def\XINT_fadd_D #1#2{\XINT_outfrac {#2}{#1}}%

```

37.36 \xintSub

```

1232 \def\xintSub {\romannumeral0\xintsub }%
1233 \def\xintsub #1%
1234 {%
1235   \expandafter\xint_fsub\expandafter {\romannumeral0\XINT_infrac {#1}}%
1236 }%
1237 \def\xint_fsub #1#2%
1238   {\expandafter\XINT_fsub_A\romannumeral0\XINT_infrac {#2}#1}%
1239 \def\XINT_fsub_A #1#2#3#4%
1240 {%
1241   \ifnum #4 > #1
1242     \xint_afterfi {\XINT_fsub_B {#1}}%
1243   \else
1244     \xint_afterfi {\XINT_fsub_B {#4}}%
1245   \fi
1246   {#1}{#4}{#2}{#3}%
1247 }%
1248 \def\XINT_fsub_B #1#2#3#4#5#6#7%
1249 {%
1250   \expandafter\XINT_fsub_C\expandafter
1251   {\romannumeral0\xintiimul {#7}{#5}}%
1252   {\romannumeral0\xintiisub
1253   {\romannumeral0\xintiimul {\xintDSH {\the\numexpr -#3+#1\relax}{#6}}{#5}}%
1254   {\romannumeral0\xintiimul {#7}{\xintDSH {\the\numexpr -#2+#1\relax}{#4}}}%
1255   }%
1256   {#1}%
1257 }%
1258 \def\XINT_fsub_C #1#2#3%
1259 {%
1260   \expandafter\XINT_fsub_D\expandafter {#2}{#3}{#1}%
1261 }%
1262 \def\XINT_fsub_D #1#2{\XINT_outfrac {#2}{#1}}%

```

37.37 \xintSum, \xintSumExpr

```

1263 \def\xintSum {\romannumeral0\xintsum }%
1264 \def\xintsum #1{\xintsumexpr #1\relax }%
1265 \def\xintSumExpr {\romannumeral0\xintsumexpr }%
1266 \def\xintsumexpr {\expandafter\XINT_fsumexpr\romannumeral-‘0}%
1267 \def\XINT_fsumexpr {\XINT_fsum_loop_a {0/1[0]}}%
1268 \def\XINT_fsum_loop_a #1#2%
1269 {%
1270   \expandafter\XINT_fsum_loop_b \romannumeral-‘0#2\Z {#1}%
1271 }%
1272 \def\XINT_fsum_loop_b #1%
1273 {%
1274   \xint_gob_til_relax #1\XINT_fsum_finished\relax
1275   \XINT_fsum_loop_c #1%
1276 }%

```

```

1277 \def\XINT_fsum_loop_c #1\Z #2%
1278 {%
1279   \expandafter\XINT_fsum_loop_a\expandafter{\romannumeral0\xintadd {#2}{#1}}%
1280 }%
1281 \def\XINT_fsum_finished #1\Z #2{ #2}%

```

37.38 \xintSum:csv

1.09a. For use by \xintexpr.

```

1282 \def\xintSum:csv #1{\expandafter\XINT_sum:_a\romannumeral-‘0#1,,^}%
1283 \def\XINT_sum:_a {\XINT_sum:_b {0/1[0]}}%
1284 \def\XINT_sum:_b #1#2,{\expandafter\XINT_sum:_c\romannumeral-‘0#2,{#1}}%
1285 \def\XINT_sum:_c #1{\if #1,\expandafter\XINT_sum:_e
1286   \else\expandafter\XINT_sum:_d\fi #1}%
1287 \def\XINT_sum:_d #1,#2{\expandafter\XINT_sum:_b\expandafter
1288   {\romannumeral0\xintadd {#2}{#1}}}%
1289 \def\XINT_sum:_e ,#1#2^{#1}% allows empty list

```

37.39 \xintMul

```

1290 \def\xintMul {\romannumeral0\xintmul }%
1291 \def\xintmul #1%
1292 {%
1293   \expandafter\xint_fmulo\expandafter {\romannumeral0\XINT_infrac {#1}}%
1294 }%
1295 \def\xint_fmulo #1#2%
1296   {\expandafter\XINT_fmulo_A\romannumeral0\XINT_infrac {#2}{#1}}%
1297 \def\XINT_fmulo_A #1#2#3#4#5#6%
1298 {%
1299   \expandafter\XINT_fmulo_B
1300   \expandafter{\the\numexpr #1+#4\expandafter}%
1301   \expandafter{\romannumeral0\xintiimul {#6}{#3}}%
1302   {\romannumeral0\xintiimul {#5}{#2}}%
1303 }%
1304 \def\XINT_fmulo_B #1#2#3%
1305 {%
1306   \expandafter \XINT_fmulo_C \expandafter{#3}{#1}{#2}%
1307 }%
1308 \def\XINT_fmulo_C #1#2{\XINT_outfrac {#2}{#1}}%

```

37.40 \xintSqr

```

1309 \def\xintSqr {\romannumeral0\xintsqr }%
1310 \def\xintsqr #1%
1311 {%
1312   \expandafter\xint_fsqr\expandafter{\romannumeral0\XINT_infrac {#1}}%
1313 }%
1314 \def\xint_fsqr #1{\XINT_fmulo_A #1#1}%

```

37.41 \xintPow

Modified in 1.06 to give the exponent to a \numexpr.

With 1.07 and for use within the \xintexpr parser, we must allow fractions (which are integers in disguise) as input to the exponent, so we must have a variant which uses \xintNum and not only \numexpr for normalizing the input. Hence the \xintfPow here. 1.08b: well actually I think that with xintfrac.sty loaded the exponent should always be allowed to be a fraction giving an integer. So I do as for \xintFac, and remove here the duplicated. The \xintexpr can thus use directly \xintPow.

```

1315 \def\xintPow {\romannumeral0\xintpow }%
1316 \def\xintpow #1%
1317 {%
1318   \expandafter\xint_fpow\expandafter {\romannumeral0\xint_infrac {#1}}%
1319 }%
1320 \def\xint_fpow #1#2%
1321 {%
1322   \expandafter\xint_fpow_fork\the\numexpr \xintNum{#2}\relax\Z #1%
1323 }%
1324 \def\xint_fpow_fork #1#2\Z
1325 {%
1326   \xint_UDzerominusfork
1327   #1-\dummy \XINT_fpow_zero
1328   0#1\dummy \XINT_fpow_neg
1329   0-\dummy {\XINT_fpow_pos #1}%
1330   \krof
1331   {#2}%
1332 }%
1333 \def\xint_fpow_zero #1#2#3#4%
1334 {%
1335   \space 1/1[0]%
1336 }%
1337 \def\xint_fpow_pos #1#2#3#4#5%
1338 {%
1339   \expandafter\xint_fpow_pos_A\expandafter
1340   {\the\numexpr #1#2*#3\expandafter}\expandafter
1341   {\romannumeral0\xintiipow {#5}{#1#2}}%
1342   {\romannumeral0\xintiipow {#4}{#1#2}}%
1343 }%
1344 \def\xint_fpow_neg #1#2#3#4%
1345 {%
1346   \expandafter\xint_fpow_pos_A\expandafter
1347   {\the\numexpr -#1*#2\expandafter}\expandafter
1348   {\romannumeral0\xintiipow {#3}{#1}}%
1349   {\romannumeral0\xintiipow {#4}{#1}}%
1350 }%
1351 \def\xint_fpow_pos_A #1#2#3%
1352 {%

```



```

1353 \expandafter\XINT_fpow_pos_B\expandafter {#3}{#1}{#2}%
1354 }%
1355 \def\XINT_fpow_pos_B #1#2{\XINT_outfrac {#2}{#1}}%

```

37.42 \xintFac

1.07: to be used by the \xintexpr scanner which needs to be able to apply \xintFac to a fraction which is an integer in disguise; so we use \xintNum and not only \numexpr. Je modifie cela dans 1.08b, au lieu d'avoir un \xintfFac spécialement pour \xintexpr, tout simplement j'étends \xintFac comme les autres macros, pour qu'elle utilise \xintNum.

```

1356 \def\xintFac {\romannumeral0\xintfac }%
1357 \def\xintfac #1%
1358 {%
1359 \expandafter\XINT_fac_fork\expandafter{\the\numexpr \xintNum{#1}}%
1360 }%

```

37.43 \xintPrd, \xintPrdExpr

```

1361 \def\xintPrd {\romannumeral0\xintprd }%
1362 \def\xintprd #1{\xintprdexpr #1\relax }%
1363 \def\xintPrdExpr {\romannumeral0\xintprdexpr }%
1364 \def\xintprdexpr {\expandafter\XINT_fprdexpr \romannumeral-‘0}%
1365 \def\XINT_fprdexpr {\XINT_fprod_loop_a {1/1[0]}}%
1366 \def\XINT_fprod_loop_a #1#2%
1367 {%
1368 \expandafter\XINT_fprod_loop_b \romannumeral-‘0#2\Z {#1}%
1369 }%
1370 \def\XINT_fprod_loop_b #1%
1371 {%
1372 \xint_gob_til_relax #1\XINT_fprod_finished\relax
1373 \XINT_fprod_loop_c #1%
1374 }%
1375 \def\XINT_fprod_loop_c #1\Z #2%
1376 {%
1377 \expandafter\XINT_fprod_loop_a\expandafter{\romannumeral0\xintmul {#1}{#2}}%
1378 }%
1379 \def\XINT_fprod_finished #1\Z #2{ #2}%

```

37.44 \xintPrd:csv

1.09a. For use by \xintexpr.

```

1380 \def\xintPrd:csv #1{\expandafter\XINT_prd:_a\romannumeral-‘0#1,,^}%
1381 \def\XINT_prd:_a {\XINT_prd:_b {1/1[0]}}%
1382 \def\XINT_prd:_b #1#2,{\expandafter\XINT_prd:_c\romannumeral-‘0#2,{#1}}%
1383 \def\XINT_prd:_c #1{\if #1,\expandafter\XINT_prd:_e
1384 \else\expandafter\XINT_prd:_d\fi #1}%

```

37 Package *xintfrac* implementation

```
1385 \def\XINT_prd:_d #1,#2{\expandafter\XINT_prd:_b\expandafter
1386           {\romannumeral0\xintmul {#2}{#1}}}%
1387 \def\XINT_prd:_e ,#1#2^{#1}% allows empty list
```

37.45 \xintDiv

```
1388 \def\xintDiv {\romannumeral0\xintdiv }%
1389 \def\xintdiv #1%
1390 {%
1391   \expandafter\xint_fdiv\expandafter {\romannumeral0\XINT_infrac {#1}}%
1392 }%
1393 \def\xint_fdiv #1#2%
1394   {\expandafter\XINT_fdiv_A\romannumeral0\XINT_infrac {#2}{#1}}%
1395 \def\XINT_fdiv_A #1#2#3#4#5#6%
1396 {%
1397   \expandafter\XINT_fdiv_B
1398   \expandafter{\the\numexpr #4-#1\expandafter}%
1399   \expandafter{\romannumeral0\xintiimul {#2}{#6}}%
1400   {\romannumeral0\xintiimul {#3}{#5}}%
1401 }%
1402 \def\XINT_fdiv_B #1#2#3%
1403 {%
1404   \expandafter\XINT_fdiv_C
1405   \expandafter{#3}{#1}{#2}%
1406 }%
1407 \def\XINT_fdiv_C #1#2{\XINT_outfrac {#2}{#1}}%
```

37.46 \xintIsOne

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

```
1408 \def\xintIsOne {\romannumeral0\xintisone }%
1409 \def\xintisone #1{\expandafter\XINT_fracisone
1410       \romannumeral0\xintrawwithzeros{#1}\Z }%
1411 \def\XINT_fracisone #1/#2\Z{\xintsgnfork{\XINT_Cmp {#1}{#2}}{0}{1}{0}}%
```

37.47 \xintGeq

Rewritten completely in 1.08a to be less dumb when comparing fractions having big powers of tens.

```
1412 \def\xintGeq {\romannumeral0\xintgeq }%
1413 \def\xintgeq #1%
1414 {%
1415   \expandafter\xint_fgeq\expandafter {\romannumeral0\xintabs {#1}}%
1416 }%
1417 \def\xint_fgeq #1#2%
1418 {%
```

```

1419 \expandafter\XINT_fgeq_A \romannumeral0\xintabs {#2}#1%
1420 }%
1421 \def\XINT_fgeq_A #1%
1422 {%
1423 \xint_gob_til_zero #1\XINT_fgeq_Zii 0%
1424 \XINT_fgeq_B #1%
1425 }%
1426 \def\XINT_fgeq_Zii 0\XINT_fgeq_B #1[#2]#3[#4]{ 1}%
1427 \def\XINT_fgeq_B #1/#2[#3]#4#5/#6[#7]%
1428 {%
1429 \xint_gob_til_zero #4\XINT_fgeq_Zi 0%
1430 \expandafter\XINT_fgeq_C\expandafter
1431 {\the\numexpr #7-#3\expandafter}\expandafter
1432 {\romannumeral0\xintiimul {#4#5}{#2}}%
1433 {\romannumeral0\xintiimul {#6}{#1}}%
1434 }%
1435 \def\XINT_fgeq_Zi 0#1#2#3#4#5#6#7{ 0}%
1436 \def\XINT_fgeq_C #1#2#3%
1437 {%
1438 \expandafter\XINT_fgeq_D\expandafter
1439 {#3}{#1}{#2}%
1440 }%
1441 \def\XINT_fgeq_D #1#2#3%
1442 {%
1443 \xintSgnFork
1444 {\xintiiSgn{\the\numexpr #2+\xintLength{#3}-\xintLength{#1}\relax}}%
1445 { 0}{\XINT_fgeq_E #2\Z {#3}{#1}}{ 1}%
1446 }%
1447 \def\XINT_fgeq_E #1%
1448 {%
1449 \xint_UDsignfork
1450 #1\dummy \XINT_fgeq_Fd
1451 -\dummy {\XINT_fgeq_Fn #1}%
1452 \krof
1453 }%
1454 \def\XINT_fgeq_Fd #1\Z #2#3%
1455 {%
1456 \expandafter\XINT_fgeq_Fe\expandafter
1457 {\romannumeral0\XINT_dsx_addzerosnofuss {#1}{#3}}{#2}%
1458 }%
1459 \def\XINT_fgeq_Fe #1#2{\XINT_geq_pre {#2}{#1}}%
1460 \def\XINT_fgeq_Fn #1\Z #2#3%
1461 {%
1462 \expandafter\XINT_geq_pre\expandafter
1463 {\romannumeral0\XINT_dsx_addzerosnofuss {#1}{#2}}{#3}%
1464 }%

```

37.48 \xintMax

Rewritten completely in 1.08a.

```

1465 \def\xintMax {\romannumeral0\xintmax }%
1466 \def\xintmax #1%
1467 {%
1468   \expandafter\xint_fmax\expandafter {\romannumeral0\xintra {#1}}%
1469 }%
1470 \def\xint_fmax #1#2%
1471 {%
1472   \expandafter\XINT_fmax_A\romannumeral0\xintra {#2}#1%
1473 }%
1474 \def\XINT_fmax_A #1#2/#3[#4]#5#6/#7[#8]%
1475 {%
1476   \xint_UDsignsfork
1477     #1#5\dummy \XINT_fmax_minusminus
1478     -#5\dummy \XINT_fmax_firstneg
1479     #1-\dummy \XINT_fmax_secondneg
1480     --\dummy \XINT_fmax_nonneg_a
1481   \krof
1482   #1#5{#2/#3[#4]}{#6/#7[#8]}%
1483 }%
1484 \def\XINT_fmax_minusminus --%
1485   {\expandafter\xint_minus_andstop\romannumeral0\XINT_fmin_nonneg_b }%
1486 \def\XINT_fmax_firstneg #1-#2#3{ #1#2}%
1487 \def\XINT_fmax_secondneg -#1#2#3{ #1#3}%
1488 \def\XINT_fmax_nonneg_a #1#2#3#4%
1489 {%
1490   \XINT_fmax_nonneg_b {#1#3}{#2#4}%
1491 }%
1492 \def\XINT_fmax_nonneg_b #1#2%
1493 {%
1494   \ifcase\romannumeral0\XINT_fgeq_A #1#2
1495     \xint_afterfi{ #1}%
1496   \or \xint_afterfi{ #2}%
1497   \fi
1498 }%

```

37.49 \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 \xintra.

```

1499 \def\xintMaxof {\romannumeral0\xintmaxof }%
1500 \def\xintmaxof #1{\expandafter\XINT_maxof_a\romannumeral-‘0#1\relax }%
1501 \def\XINT_maxof_a #1{\expandafter\XINT_maxof_b\romannumeral0\xintra{#1}\Z }%
1502 \def\XINT_maxof_b #1\Z #2%
1503   {\expandafter\XINT_maxof_c\romannumeral-‘0#2\Z {#1}\Z}%

```

```

1504 \def\XINT_maxof_c #1%
1505     {\xint_gob_til_relax #1\XINT_maxof_e\relax\XINT_maxof_d #1}%
1506 \def\XINT_maxof_d #1\Z
1507     {\expandafter\XINT_maxof_b\romannumeral0\xintmax {#1}}%
1508 \def\XINT_maxof_e #1\Z #2\Z { #2}%

```

37.50 \xintMaxof:csv

1.09a. For use by \xintexpr.

```

1509 \def\xintMaxof:csv #1{\expandafter\XINT_maxof:_b\romannumeral-‘0#1,,}%
1510 \def\XINT_maxof:_b #1,#2,{\expandafter\XINT_maxof:_c\romannumeral-‘0#2,{#1},}%
1511 \def\XINT_maxof:_c #1{\if #1,\expandafter\XINT_maxof:_e
1512     \else\expandafter\XINT_maxof:_d\fi #1}%
1513 \def\XINT_maxof:_d #1,{\expandafter\XINT_maxof:_b\romannumeral0\xintmax {#1}}%
1514 \def\XINT_maxof:_e ,#1,{#1}%

```

37.51 \XINTinFloatMaxof

1.09a, for use by \xintNewFloatExpr. Name changed in 1.09h

```

1515 \def\XINTinFloatMaxof {\romannumeral0\XINTinfloatmaxof}%
1516 \def\XINTinfloatmaxof #1{\expandafter\XINT_flmaxof:_a\romannumeral-‘0#1\relax}%
1517 \def\XINT_flmaxof:_a #1{\expandafter\XINT_flmaxof:_b
1518     \romannumeral0\XINT_inFloat [\XINTdigits]{#1}\Z}%
1519 \def\XINT_flmaxof:_b #1\Z #2%
1520     {\expandafter\XINT_flmaxof:_c\romannumeral-‘0#2\Z {#1}\Z}%
1521 \def\XINT_flmaxof:_c #1%
1522     {\xint_gob_til_relax #1\XINT_flmaxof_e\relax\XINT_flmaxof_d #1}%
1523 \def\XINT_flmaxof_d #1\Z
1524     {\expandafter\XINT_flmaxof:_b\romannumeral0\xintmax
1525     {\XINTinFloat [\XINTdigits]{#1}}}%
1526 \def\XINT_flmaxof_e #1\Z #2\Z { #2}%

```

37.52 \XINTinFloatMaxof:csv

1.09a. For use by \xintfloatexpr. Name changed in 1.09h

```

1527 \def\XINTinFloatMaxof:csv #1{\expandafter\XINT_flmaxof:_a\romannumeral-‘0#1,,}%
1528 \def\XINT_flmaxof:_a #1,{\expandafter\XINT_flmaxof:_b
1529     \romannumeral0\XINT_inFloat [\XINTdigits]{#1},}%
1530 \def\XINT_flmaxof:_b #1,#2,%
1531     {\expandafter\XINT_flmaxof:_c\romannumeral-‘0#2,{#1},}%
1532 \def\XINT_flmaxof:_c #1{\if #1,\expandafter\XINT_flmaxof:_e
1533     \else\expandafter\XINT_flmaxof:_d\fi #1}%
1534 \def\XINT_flmaxof:_d #1,%
1535     {\expandafter\XINT_flmaxof:_b\romannumeral0\xintmax
1536     {\XINTinFloat [\XINTdigits]{#1}}}%
1537 \def\XINT_flmaxof:_e ,#1,{#1}%

```

37.53 \xintMin

Rewritten completely in 1.08a.

```

1538 \def\xintMin {\romannumeral0\xintmin }%
1539 \def\xintmin #1%
1540 {%
1541   \expandafter\xint_fmin\expandafter {\romannumeral0\xintra {#1}}%
1542 }%
1543 \def\xint_fmin #1#2%
1544 {%
1545   \expandafter\XINT_fmin_A\romannumeral0\xintra {#2}#1%
1546 }%
1547 \def\XINT_fmin_A #1#2/#3[#4]#5#6/#7[#8]%
1548 {%
1549   \xint_UDsignsfork
1550     #1#5\dummy \XINT_fmin_minusminus
1551     -#5\dummy \XINT_fmin_firstneg
1552     #1-\dummy \XINT_fmin_secondneg
1553     --\dummy \XINT_fmin_nonneg_a
1554   \krof
1555   #1#5{#2/#3[#4]}{#6/#7[#8]}%
1556 }%
1557 \def\XINT_fmin_minusminus --%
1558   {\expandafter\xint_minus_andstop\romannumeral0\XINT_fmax_nonneg_b }%
1559 \def\XINT_fmin_firstneg #1-#2#3{ -#3}%
1560 \def\XINT_fmin_secondneg -#1#2#3{ -#2}%
1561 \def\XINT_fmin_nonneg_a #1#2#3#4%
1562 {%
1563   \XINT_fmin_nonneg_b {#1#3}{#2#4}%
1564 }%
1565 \def\XINT_fmin_nonneg_b #1#2%
1566 {%
1567   \ifcase\romannumeral0\XINT_fgeq_A #1#2
1568     \xint_afterfi{ #2}%
1569   \or \xint_afterfi{ #1}%
1570   \fi
1571 }%

```

37.54 \xintMinof

```

1572 \def\xintMinof {\romannumeral0\xintminof }%
1573 \def\xintminof #1{\expandafter\XINT_minof_a\romannumeral-‘0#1\relax }%
1574 \def\XINT_minof_a #1{\expandafter\XINT_minof_b\romannumeral0\xintra{#1}\Z }%
1575 \def\XINT_minof_b #1\Z #2%
1576   {\expandafter\XINT_minof_c\romannumeral-‘0#2\Z {#1}\Z}%
1577 \def\XINT_minof_c #1%
1578   {\xint_gob_til_relax #1\XINT_minof_e\relax\XINT_minof_d #1}%
1579 \def\XINT_minof_d #1\Z

```

```

1580      {\expandafter\XINT_minof_b\romannumeral0\xintmin {#1}}%
1581 \def\XINT_minof_e #1\Z #2\Z { #2}%

```

37.55 \xintMinof:csv

1.09a. For use by \xintexpr.

```

1582 \def\xintMinof:csv #1{\expandafter\XINT_minof:_b\romannumeral-‘0#1,,}%
1583 \def\XINT_minof:_b #1,#2,{\expandafter\XINT_minof:_c\romannumeral-‘0#2,{#1},}%
1584 \def\XINT_minof:_c #1{\if #1,\expandafter\XINT_minof:_e
1585      \else\expandafter\XINT_minof:_d\fi #1}%
1586 \def\XINT_minof:_d #1,{\expandafter\XINT_minof:_b\romannumeral0\xintmin {#1}}%
1587 \def\XINT_minof:_e ,#1,{#1}%

```

37.56 \XINTinFloatMinof

1.09a, for use by \xintNewFloatExpr. Name changed in 1.09h

```

1588 \def\XINTinFloatMinof {\romannumeral0\XINTinfloatminof}%
1589 \def\XINTinfloatminof #1{\expandafter\XINT_flminof_a\romannumeral-‘0#1\relax}%
1590 \def\XINT_flminof_a #1{\expandafter\XINT_flminof_b
1591      \romannumeral0\XINT_inFloat [\XINTdigits]{#1}\Z}%
1592 \def\XINT_flminof_b #1\Z #2%
1593      {\expandafter\XINT_flminof_c\romannumeral-‘0#2\Z {#1}\Z}%
1594 \def\XINT_flminof_c #1%
1595      {\xint_gob_til_relax #1\XINT_flminof_e\relax\XINT_flminof_d #1}%
1596 \def\XINT_flminof_d #1\Z
1597      {\expandafter\XINT_flminof_b\romannumeral0\xintmin
1598      {\XINTinFloat [\XINTdigits]{#1}}}%
1599 \def\XINT_flminof_e #1\Z #2\Z { #2}%

```

37.57 \XINTinFloatMinof:csv

1.09a. For use by \xintfloatexpr. Name changed in 1.09h

```

1600 \def\XINTinFloatMinof:csv #1{\expandafter\XINT_flminof:_a\romannumeral-‘0#1,,}%
1601 \def\XINT_flminof:_a #1,{\expandafter\XINT_flminof:_b
1602      \romannumeral0\XINT_inFloat [\XINTdigits]{#1},}%
1603 \def\XINT_flminof:_b #1,#2,%
1604      {\expandafter\XINT_flminof:_c\romannumeral-‘0#2,{#1},}%
1605 \def\XINT_flminof:_c #1{\if #1,\expandafter\XINT_flminof:_e
1606      \else\expandafter\XINT_flminof:_d\fi #1}%
1607 \def\XINT_flminof:_d #1,%
1608      {\expandafter\XINT_flminof:_b\romannumeral0\xintmin
1609      {\XINTinFloat [\XINTdigits]{#1}}}%
1610 \def\XINT_flminof:_e ,#1,{#1}%

```

37.58 \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...

```

1611 \def\xintCmp {\romannumeral0\xintcmp }%
1612 \def\xintcmp #1%
1613 {%
1614   \expandafter\xint_fcmp\expandafter {\romannumeral0\xintra {#1}}%
1615 }%
1616 \def\xint_fcmp #1#2%
1617 {%
1618   \expandafter\XINT_fcmp_A\romannumeral0\xintra {#2}#1%
1619 }%
1620 \def\XINT_fcmp_A #1#2/#3[#4]#5#6/#7[#8]%
1621 {%
1622   \xint_UDsignsfork
1623     #1#5\dummy \XINT_fcmp_minusminus
1624     -#5\dummy \XINT_fcmp_firstneg
1625     #1-\dummy \XINT_fcmp_secondneg
1626     --\dummy \XINT_fcmp_nonneg_a
1627   \krof
1628   #1#5{#2/#3[#4]}{#6/#7[#8]}%
1629 }%
1630 \def\XINT_fcmp_minusminus --#1#2{\XINT_fcmp_B #2#1}%
1631 \def\XINT_fcmp_firstneg #1-#2#3{ -1}%
1632 \def\XINT_fcmp_secondneg -#1#2#3{ 1}%
1633 \def\XINT_fcmp_nonneg_a #1#2%
1634 {%
1635   \xint_UDzerosfork
1636     #1#2\dummy \XINT_fcmp_zerozero
1637     0#2\dummy \XINT_fcmp_firstzero
1638     #10\dummy \XINT_fcmp_secondzero
1639     00\dummy \XINT_fcmp_pos
1640   \krof
1641   #1#2%
1642 }%
1643 \def\XINT_fcmp_zerozero #1#2#3#4{ 0}% 1.08b had some [ and ] here!!!
1644 \def\XINT_fcmp_firstzero #1#2#3#4{ -1}% incredibly I never saw that until
1645 \def\XINT_fcmp_secondzero #1#2#3#4{ 1}% preparing 1.09a.
1646 \def\XINT_fcmp_pos #1#2#3#4%
1647 {%
1648   \XINT_fcmp_B #1#3#2#4%
1649 }%
1650 \def\XINT_fcmp_B #1/#2[#3]#4/#5[#6]%
1651 {%

```



```

1652 \expandafter\XINT_fcmp_C\expandafter
1653 {\the\numexpr #6-#3\expandafter}\expandafter
1654 {\romannumeral0\xintiimul {#4}{#2}}%
1655 {\romannumeral0\xintiimul {#5}{#1}}%
1656 }%
1657 \def\XINT_fcmp_C #1#2#3%
1658 {%
1659 \expandafter\XINT_fcmp_D\expandafter
1660 {#3}{#1}{#2}%
1661 }%
1662 \def\XINT_fcmp_D #1#2#3%
1663 {%
1664 \xintSgnFork
1665 {\xintiiSgn{\the\numexpr #2+\xintLength{#3}-\xintLength{#1}\relax}}%
1666 {-1}{\XINT_fcmp_E #2\Z {#3}{#1}}{ 1}%
1667 }%
1668 \def\XINT_fcmp_E #1%
1669 {%
1670 \xint_UDsignfork
1671 #1\dummy \XINT_fcmp_Fd
1672 -\dummy {\XINT_fcmp_Fn #1}%
1673 \krof
1674 }%
1675 \def\XINT_fcmp_Fd #1\Z #2#3%
1676 {%
1677 \expandafter\XINT_fcmp_Fe\expandafter
1678 {\romannumeral0\XINT_dsx_addzerosnofuss {#1}{#3}}{#2}%
1679 }%
1680 \def\XINT_fcmp_Fe #1#2{\XINT_cmp_pre {#2}{#1}}%
1681 \def\XINT_fcmp_Fn #1\Z #2#3%
1682 {%
1683 \expandafter\XINT_cmp_pre\expandafter
1684 {\romannumeral0\XINT_dsx_addzerosnofuss {#1}{#2}}{#3}%
1685 }%

```

37.59 \xintAbs

```

1686 \def\xintAbs {\romannumeral0\xintabs }%
1687 \def\xintabs #1%
1688 {%
1689 \expandafter\xint_fabs\romannumeral0\XINT_infrac {#1}%
1690 }%
1691 \def\xint_fabs #1#2%
1692 {%
1693 \expandafter\XINT_outfrac\expandafter
1694 {\the\numexpr #1\expandafter}\expandafter
1695 {\romannumeral0\XINT_abs #2}%
1696 }%

```

37.60 \xintOpp

```

1697 \def\xintOpp {\romannumeral0\xintopp }%
1698 \def\xintopp #1%
1699 {%
1700   \expandafter\xint_fopp\romannumeral0\XINT_infrac {#1}%
1701 }%
1702 \def\xint_fopp #1#2%
1703 {%
1704   \expandafter\XINT_outfrac\expandafter
1705   {\the\numexpr #1\expandafter}\expandafter
1706   {\romannumeral0\XINT_opp #2}%
1707 }%

```

37.61 \xintSgn

```

1708 \def\xintSgn {\romannumeral0\xintsgn }%
1709 \def\xintsgn #1%
1710 {%
1711   \expandafter\xint_fsgn\romannumeral0\XINT_infrac {#1}%
1712 }%
1713 \def\xint_fsgn #1#2#3{\xintiisgn {#2}}%

```

37.62 \xintFloatAdd

1.07

```

1714 \def\xintFloatAdd {\romannumeral0\xintfloatadd }%
1715 \def\xintfloatadd #1{\XINT_fladd_chkopt \xintfloat #1\Z }%
1716 \def\XINTinFloatAdd {\romannumeral0\XINTinfloatadd }%
1717 \def\XINTinfloatadd #1{\XINT_fladd_chkopt \XINT_inFloat #1\Z }%
1718 \def\XINT_fladd_chkopt #1#2%
1719 {%
1720   \ifx [#2\expandafter\XINT_fladd_opt
1721   \else\expandafter\XINT_fladd_noopt
1722   \fi #1#2%
1723 }%
1724 \def\XINT_fladd_noopt #1#2\Z #3%
1725 {%
1726   #1[\XINTdigits]{\XINT_FL_Add {\XINTdigits+2}{#2}{#3}}%
1727 }%
1728 \def\XINT_fladd_opt #1[\Z #2]#3#4%
1729 {%
1730   #1[#2]{\XINT_FL_Add {#2+2}{#3}{#4}}%
1731 }%
1732 \def\XINT_FL_Add #1#2%
1733 {%
1734   \expandafter\XINT_FL_Add_a\expandafter{\the\numexpr #1\expandafter}%
1735   \expandafter{\romannumeral0\XINT_inFloat [#1]{#2}}%
1736 }%
1737 \def\XINT_FL_Add_a #1#2#3%

```

```

1738 {%
1739   \expandafter\XINT_FL_Add_b\romannumeral0\XINT_inFloat [#1]{#3}#2{#1}%
1740 }%
1741 \def\XINT_FL_Add_b #1%
1742 {%
1743   \xint_gob_til_zero #1\XINT_FL_Add_zero 0\XINT_FL_Add_c #1%
1744 }%
1745 \def\XINT_FL_Add_c #1[#2]#3%
1746 {%
1747   \xint_gob_til_zero #3\XINT_FL_Add_zerobis 0\XINT_FL_Add_d #1[#2]#3%
1748 }%
1749 \def\XINT_FL_Add_d #1[#2]#3[#4]#5%
1750 {%
1751   \xintSgnFork {\ifnum \numexpr #2-#4-#5>1 \expandafter 1%
1752                 \else\ifnum \numexpr #4-#2-#5>1
1753                   \xint_afterfi {\expandafter-\expandafter1}%
1754                   \else \expandafter\expandafter\expandafter0%
1755                   \fi
1756                 \fi}%
1757   {#3[#4]}\xintAdd {#1[#2]}\{#3[#4]}\{#1[#2]}}%
1758 }%
1759 \def\XINT_FL_Add_zero 0\XINT_FL_Add_c 0[0]#1[#2]#3{#1[#2]}}%
1760 \def\XINT_FL_Add_zerobis 0\XINT_FL_Add_d #1[#2]0[0]#3{#1[#2]}}%

```

37.63 \xintFloatSub

1.07

```

1761 \def\xintFloatSub {\romannumeral0\xintfloatsub }%
1762 \def\xintfloatsub #1{\XINT_flsub_chkopt \xintfloat #1\Z }%
1763 \def\XINTinFloatSub {\romannumeral0\XINTinfloatsub }%
1764 \def\XINTinfloatsub #1{\XINT_flsub_chkopt \XINT_inFloat #1\Z }%
1765 \def\XINT_flsub_chkopt #1#2%
1766 {%
1767   \ifx [#2\expandafter\XINT_flsub_opt
1768     \else\expandafter\XINT_flsub_noopt
1769     \fi #1#2%
1770 }%
1771 \def\XINT_flsub_noopt #1#2\Z #3%
1772 {%
1773   #1[\XINTdigits]{\XINT_FL_Add {\XINTdigits+2}{#2}\xintOpp{#3}}}%
1774 }%
1775 \def\XINT_flsub_opt #1[\Z #2]#3#4%
1776 {%
1777   #1[#2]{\XINT_FL_Add {#2+2}{#3}\xintOpp{#4}}}%
1778 }%

```

37.64 \xintFloatMul

1.07

```

1779 \def\xintFloatMul    {\romannumeral0\xintfloatmul}%
1780 \def\xintfloatmul    #1{\XINT_flmul_chkopt \xintfloat #1\Z }%
1781 \def\XINTinFloatMul  {\romannumeral0\XINTinfloatmul }%
1782 \def\XINTinfloatmul #1{\XINT_flmul_chkopt \XINT_inFloat #1\Z }%
1783 \def\XINT_flmul_chkopt #1#2%
1784 {%
1785   \ifx [#2\expandafter\XINT_flmul_opt
1786     \else\expandafter\XINT_flmul_noopt
1787     \fi #1#2%
1788 }%
1789 \def\XINT_flmul_noopt #1#2\Z #3%
1790 {%
1791   #1[\XINTdigits]{\XINT_FL_Mul {\XINTdigits+2}{#2}{#3}}%
1792 }%
1793 \def\XINT_flmul_opt #1[\Z #2]#3#4%
1794 {%
1795   #1[#2]{\XINT_FL_Mul {#2+2}{#3}{#4}}%
1796 }%
1797 \def\XINT_FL_Mul #1#2%
1798 {%
1799   \expandafter\XINT_FL_Mul_a\expandafter{\the\numexpr #1\expandafter}%
1800   \expandafter{\romannumeral0\XINT_inFloat [#1]{#2}}%
1801 }%
1802 \def\XINT_FL_Mul_a #1#2#3%
1803 {%
1804   \expandafter\XINT_FL_Mul_b\romannumeral0\XINT_inFloat [#1]{#3}#2%
1805 }%
1806 \def\XINT_FL_Mul_b #1[#2]#3[#4]{\xintE{\xintiiMul {#1}{#3}}{#2+#4}}%

```

37.65 \xintFloatDiv

1.07

```

1807 \def\xintFloatDiv    {\romannumeral0\xintfloatdiv}%
1808 \def\xintfloatdiv    #1{\XINT_fldiv_chkopt \xintfloat #1\Z }%
1809 \def\XINTinFloatDiv  {\romannumeral0\XINTinfloatdiv }%
1810 \def\XINTinfloatdiv #1{\XINT_fldiv_chkopt \XINT_inFloat #1\Z }%
1811 \def\XINT_fldiv_chkopt #1#2%
1812 {%
1813   \ifx [#2\expandafter\XINT_fldiv_opt
1814     \else\expandafter\XINT_fldiv_noopt
1815     \fi #1#2%
1816 }%
1817 \def\XINT_fldiv_noopt #1#2\Z #3%
1818 {%

```

```

1819 #1[\XINTdigits]{\XINT_FL_Div {\XINTdigits+2}{#2}{#3}}%
1820 }%
1821 \def\XINT_fldiv_opt #1[\Z #2]#3#4%
1822 {%
1823 #1[#2]{\XINT_FL_Div {#2+2}{#3}{#4}}%
1824 }%
1825 \def\XINT_FL_Div #1#2%
1826 {%
1827 \expandafter\XINT_FL_Div_a\expandafter{\the\numexpr #1\expandafter}%
1828 \expandafter{\romannumeral0\XINT_inFloat [#1]{#2}}%
1829 }%
1830 \def\XINT_FL_Div_a #1#2#3%
1831 {%
1832 \expandafter\XINT_FL_Div_b\romannumeral0\XINT_inFloat [#1]{#3}#2%
1833 }%
1834 \def\XINT_FL_Div_b #1[#2]#3[#4]{\xintE{#3/#1}{#4-#2}}%

```

37.66 \XINTinFloatSum

1.09a: quick write-up, for use by \xintfloatexpr, will need to be thought through again. Renamed (and slightly modified) in 1.09h. Should be extended for optional precision. Should be rewritten for optimization.

```

1835 \def\XINTinFloatSum {\romannumeral0\XINTinfloatsum }%
1836 \def\XINTinfloatsum #1{\expandafter\XINT_floatsum_a\romannumeral-‘0#1\relax }%
1837 \def\XINT_floatsum_a #1{\expandafter\XINT_floatsum_b
1838 \romannumeral0\XINT_inFloat[\XINTdigits]{#1}\Z }%
1839 \def\XINT_floatsum_b #1\Z #2%
1840 {\expandafter\XINT_floatsum_c\romannumeral-‘0#2\Z {#1}\Z}%
1841 \def\XINT_floatsum_c #1%
1842 {\xint_gob_til_relax #1\XINT_floatsum_e\relax\XINT_floatsum_d #1}%
1843 \def\XINT_floatsum_d #1\Z
1844 {\expandafter\XINT_floatsum_b\romannumeral0\XINTinfloatadd {#1}}%
1845 \def\XINT_floatsum_e #1\Z #2\Z { #2}%

```

37.67 \XINTinFloatSum:csv

1.09a. For use by \xintfloatexpr. Renamed in 1.09h

```

1846 \def\XINTinFloatSum:csv #1{\expandafter\XINT_floatsum:_a\romannumeral-‘0#1,,^}%
1847 \def\XINT_floatsum:_a {\XINT_floatsum:_b {0[0]}}% was 0/1[0] prior to 1.09h
1848 \def\XINT_floatsum:_b #1#2,%
1849 {\expandafter\XINT_floatsum:_c\romannumeral-‘0#2,{#1}}%
1850 \def\XINT_floatsum:_c #1{\if #1,\expandafter\XINT_floatsum:_e
1851 \else\expandafter\XINT_floatsum:_d\fi #1}%
1852 \def\XINT_floatsum:_d #1,#2{\expandafter\XINT_floatsum:_b\expandafter
1853 {\romannumeral0\XINTinfloatadd {#2}{#1}}}%
1854 \def\XINT_floatsum:_e ,#1#2^{#1}% allows empty list

```

37.68 \XINTinFloatPrd

1.09a: quick write-up, for use by `\xintfloatexpr`, will need to be thought through again. Renamed (and slightly modified) in 1.09h. Should be extended for optional precision. Should be rewritten for optimization.

```

1855 \def\XINTinFloatPrd {\romannumeral0\XINTinfloatprd}%
1856 \def\XINTinfloatprd #1{\expandafter\XINT_floatprd_a\romannumeral-‘0#1\relax}%
1857 \def\XINT_floatprd_a #1{\expandafter\XINT_floatprd_b
1858 \romannumeral0\XINT_inFloat[\XINTdigits]{#1}\Z}%
1859 \def\XINT_floatprd_b #1\Z #2%
1860 {\expandafter\XINT_floatprd_c\romannumeral-‘0#2\Z {#1}\Z}%
1861 \def\XINT_floatprd_c #1%
1862 {\xint_gob_til_relax #1\XINT_floatprd_e\relax\XINT_floatprd_d #1}%
1863 \def\XINT_floatprd_d #1\Z
1864 {\expandafter\XINT_floatprd_b\romannumeral0\XINTinfloatmul {#1}}%
1865 \def\XINT_floatprd_e #1\Z #2\Z { #2}%

```

37.69 \XINTinFloatPrd:csv

1.09a. For use by `\xintfloatexpr`. Renamed in 1.09h

```

1866 \def\XINTinFloatPred:csv #1{\expandafter\XINT_floatprd:_a\romannumeral-‘0#1,,^}%
1867 \def\XINT_floatprd:_a {\XINT_floatprd:_b {1[0]}}% was 1/1[0] prior to 1.09h
1868 \def\XINT_floatprd:_b #1#2,%
1869 {\expandafter\XINT_floatprd:_c\romannumeral-‘0#2,{#1}}%
1870 \def\XINT_floatprd:_c #1{\if #1,\expandafter\XINT_floatprd:_e
1871 \else\expandafter\XINT_floatprd:_d\fi #1}%
1872 \def\XINT_floatprd:_d #1,#2{\expandafter\XINT_floatprd:_b\expandafter
1873 {\romannumeral0\XINTinfloatmul {#2}{#1}}}%
1874 \def\XINT_floatprd:_e ,#1#2^{#1}% allows empty list

```

37.70 \xintFloatPow

1.07

```

1875 \def\xintFloatPow {\romannumeral0\xintfloatpow}%
1876 \def\xintfloatpow #1{\XINT_flpow_chkopt \xintfloat #1\Z}%
1877 \def\XINTinFloatPow {\romannumeral0\XINTinfloatpow}%
1878 \def\XINTinfloatpow #1{\XINT_flpow_chkopt \XINT_inFloat #1\Z}%
1879 \def\XINT_flpow_chkopt #1#2%
1880 {%
1881 \ifx [#2\expandafter\XINT_flpow_opt
1882 \else\expandafter\XINT_flpow_noopt
1883 \fi
1884 #1#2%
1885}%
1886 \def\XINT_flpow_noopt #1#2\Z #3%
1887 {%

```

```

1888 \expandafter\XINT_flpow_checkB_start\expandafter
1889         {\the\numexpr #3\expandafter}\expandafter
1890         {\the\numexpr \XINTdigits}{#2}{#1[\XINTdigits]}}%
1891 }%
1892 \def\XINT_flpow_opt #1[\Z #2]#3#4%
1893 {%
1894     \expandafter\XINT_flpow_checkB_start\expandafter
1895         {\the\numexpr #4\expandafter}\expandafter
1896         {\the\numexpr #2}{#3}{#1[#2]}}%
1897 }%
1898 \def\XINT_flpow_checkB_start #1{\XINT_flpow_checkB_a #1\Z }%
1899 \def\XINT_flpow_checkB_a #1%
1900 {%
1901     \xint_UDzerominusfork
1902     #1-\dummy \XINT_flpow_BisZero
1903     0#1\dummy {\XINT_flpow_checkB_b 1}%
1904     0-\dummy {\XINT_flpow_checkB_b 0#1}%
1905     \krof
1906 }%
1907 \def\XINT_flpow_BisZero \Z #1#2#3{#3{1/1[0]}}}%
1908 \def\XINT_flpow_checkB_b #1#2\Z #3%
1909 {%
1910     \expandafter\XINT_flpow_checkB_c \expandafter
1911     {\romannumeral0\xintlength{#2}}{#3}{#2}#1%
1912 }%
1913 \def\XINT_flpow_checkB_c #1#2%
1914 {%
1915     \expandafter\XINT_flpow_checkB_d \expandafter
1916     {\the\numexpr \expandafter\xintlength\expandafter
1917         {\the\numexpr #1*20/3}+#1+#2+1}}%
1918 }%
1919 \def\XINT_flpow_checkB_d #1#2#3#4%
1920 {%
1921     \expandafter \XINT_flpow_a
1922     \romannumeral0\XINT_inFloat [#1]{#4}{#1}{#2}#3%
1923 }%
1924 \def\XINT_flpow_a #1%
1925 {%
1926     \xint_UDzerominusfork
1927     #1-\dummy \XINT_flpow_zero
1928     0#1\dummy {\XINT_flpow_b 1}%
1929     0-\dummy {\XINT_flpow_b 0#1}%
1930     \krof
1931 }%
1932 \def\XINT_flpow_zero [#1]#2#3#4#5%
1933 {%
1934     \if #41 \xint_afterfi {\xintError:DivisionByZero\space 1.e2147483647}%
1935     \else \xint_afterfi { 0.e0}\fi
1936 }%

```

```

1937 \def\XINT_flpow_b #1#2[#3]#4#5%
1938 {%
1939   \XINT_flpow_c {#4}{#5}{#2[#3]}{#1*\ifodd #5 1\else 0\fi}%
1940 }%
1941 \def\XINT_flpow_c #1#2#3#4%
1942 {%
1943   \XINT_flpow_loop {#1}{#2}{#3}{#1}\XINT_flpow_prd
1944   \xint_relax
1945   \xint_bye\xint_bye\xint_bye\xint_bye
1946   \xint_bye\xint_bye\xint_bye\xint_bye
1947   \xint_relax {#4}%
1948 }%
1949 \def\XINT_flpow_loop #1#2#3%
1950 {%
1951   \ifnum #2 = 1
1952     \expandafter\XINT_flpow_loop_end
1953   \else
1954     \xint_afterfi{\expandafter\XINT_flpow_loop_a
1955       \expandafter{\the\numexpr 2*(#2/2)-#2\expandafter }% b mod 2
1956       \expandafter{\the\numexpr #2-#2/2\expandafter }% [b/2]
1957       \expandafter{\romannumeral0\XINTinfloatmul [#1]{#3}{#3}}}%
1958   \fi
1959   {#1}{#3}%
1960 }%
1961 \def\XINT_flpow_loop_a #1#2#3#4%
1962 {%
1963   \ifnum #1 = 1
1964     \expandafter\XINT_flpow_loop
1965   \else
1966     \expandafter\XINT_flpow_loop_throwaway
1967   \fi
1968   {#4}{#2}{#3}%
1969 }%
1970 \def\XINT_flpow_loop_throwaway #1#2#3#4%
1971 {%
1972   \XINT_flpow_loop {#1}{#2}{#3}%
1973 }%
1974 \def\XINT_flpow_loop_end #1{\romannumeral0\XINT_rord_main {} \relax }%
1975 \def\XINT_flpow_prd #1#2%
1976 {%
1977   \XINT_flpow_prd_getnext {#2}{#1}%
1978 }%
1979 \def\XINT_flpow_prd_getnext #1#2#3%
1980 {%
1981   \XINT_flpow_prd_checkiffinished #3\Z {#1}{#2}%
1982 }%
1983 \def\XINT_flpow_prd_checkiffinished #1%
1984 {%
1985   \xint_gob_til_relax #1\XINT_flpow_prd_end\relax

```



```

1986 \XINT_flpow_prd_compute #1%
1987 }%
1988 \def\XINT_flpow_prd_compute #1\Z #2#3%
1989 {%
1990 \expandafter\XINT_flpow_prd_getnext\expandafter
1991 {\romannumeral0\XINTinfloatmul [#3]{#1}{#2}}{#3}%
1992 }%
1993 \def\XINT_flpow_prd_end\relax\XINT_flpow_prd_compute
1994 \relax\Z #1#2#3%
1995 {%
1996 \expandafter\XINT_flpow_conclude \the\numexpr #3\relax #1%
1997 }%
1998 \def\XINT_flpow_conclude #1#2[#3]#4%
1999 {%
2000 \expandafter\XINT_flpow_conclude_really\expandafter
2001 {\the\numexpr\if #41 -\fi#3\expandafter}%
2002 \xint_UDzerofork
2003 #4\dummy {{#2}}%
2004 0\dummy {{1/#2}}%
2005 \krof #1%
2006 }%
2007 \def\XINT_flpow_conclude_really #1#2#3#4%
2008 {%
2009 \xint_UDzerofork
2010 #3\dummy {#4{#2[#1]}}%
2011 0\dummy {#4{-#2[#1]}}%
2012 \krof
2013 }%

```

37.71 \xintFloatPower

1.07

```

2014 \def\xintFloatPower {\romannumeral0\xintfloatpower}%
2015 \def\xintfloatpower #1{\XINT_flpower_chkopt \xintfloat #1\Z }%
2016 \def\XINTinFloatPower {\romannumeral0\XINTinfloatpower}%
2017 \def\XINTinfloatpower #1{\XINT_flpower_chkopt \XINT_inFloat #1\Z }%
2018 \def\XINT_flpower_chkopt #1#2%
2019 {%
2020 \ifx [#2\expandafter\XINT_flpower_opt
2021 \else\expandafter\XINT_flpower_noopt
2022 \fi
2023 #1#2%
2024 }%
2025 \def\XINT_flpower_noopt #1#2\Z #3%
2026 {%
2027 \expandafter\XINT_flpower_checkB_start\expandafter
2028 {\the\numexpr \XINTdigits\expandafter}\expandafter
2029 {\romannumeral0\xintnum{#3}}{#2}{#1[\XINTdigits]}%

```

```

2030 }%
2031 \def\XINT_flpower_opt #1[\Z #2]#3#4%
2032 {%
2033   \expandafter\XINT_flpower_checkB_start\expandafter
2034     {\the\numexpr #2\expandafter}\expandafter
2035     {\romannumeral0\xintnum{#4}}{#3}{#1[#2]}%
2036 }%
2037 \def\XINT_flpower_checkB_start #1#2{\XINT_flpower_checkB_a #2\Z {#1}}%
2038 \def\XINT_flpower_checkB_a #1%
2039 {%
2040   \xint_UDzerominusfork
2041     #1-\dummy \XINT_flpower_BisZero
2042     0#1\dummy {\XINT_flpower_checkB_b 1}%
2043     0-\dummy {\XINT_flpower_checkB_b 0#1}%
2044   \krof
2045 }%
2046 \def\XINT_flpower_BisZero \Z #1#2#3{#3{1/1[0]}}%
2047 \def\XINT_flpower_checkB_b #1#2\Z #3%
2048 {%
2049   \expandafter\XINT_flpower_checkB_c \expandafter
2050     {\romannumeral0\xintlengh{#2}}{#3}{#2}#1%
2051 }%
2052 \def\XINT_flpower_checkB_c #1#2%
2053 {%
2054   \expandafter\XINT_flpower_checkB_d \expandafter
2055     {\the\numexpr \expandafter\xintLength\expandafter
2056       {\the\numexpr #1*20/3}+#1+#2+1}%
2057 }%
2058 \def\XINT_flpower_checkB_d #1#2#3#4%
2059 {%
2060   \expandafter \XINT_flpower_a
2061   \romannumeral0\XINT_inFloat [#1]{#4}{#1}{#2}#3%
2062 }%
2063 \def\XINT_flpower_a #1%
2064 {%
2065   \xint_UDzerominusfork
2066     #1-\dummy \XINT_flpower_zero
2067     0#1\dummy {\XINT_flpower_b 1}%
2068     0-\dummy {\XINT_flpower_b 0#1}%
2069   \krof
2070 }%
2071 \def\XINT_flpower_zero [#1]#2#3#4#5%
2072 {%
2073   \if #41
2074     \xint_afterfi {\xintError:DivisionByZero\space 1.e2147483647}%
2075   \else \xint_afterfi { 0.e0}\fi
2076 }%
2077 \def\XINT_flpower_b #1#2[#3]#4#5%
2078 {%

```

```

2079 \XINT_flpower_c {#4}{#5}{#2[#3]}{#1*\xintiiOdd {#5}}%
2080 }%
2081 \def\XINT_flpower_c #1#2#3#4%
2082 {%
2083 \XINT_flpower_loop {#1}{#2}{#3}{#1}\XINT_flpow_prd
2084 \xint_relax
2085 \xint_bye\xint_bye\xint_bye\xint_bye
2086 \xint_bye\xint_bye\xint_bye\xint_bye
2087 \xint_relax {#4}%
2088 }%
2089 \def\XINT_flpower_loop #1#2#3%
2090 {%
2091 \ifcase\XINT_isOne {#2}
2092 \xint_afterfi{\expandafter\XINT_flpower_loop_x\expandafter
2093 {\romannumeral0\XINTinfloatmul [#1]{#3}{#3}}%
2094 {\romannumeral0\xintdivision {#2}{2}}}%
2095 \or \expandafter\XINT_flpow_loop_end
2096 \fi
2097 {#1}{#3}}%
2098 }%
2099 \def\XINT_flpower_loop_x #1#2{\expandafter\XINT_flpower_loop_a #2{#1}}%
2100 \def\XINT_flpower_loop_a #1#2#3#4%
2101 {%
2102 \ifnum #2 = 1
2103 \expandafter\XINT_flpower_loop
2104 \else
2105 \expandafter\XINT_flpower_loop_throwaway
2106 \fi
2107 {#4}{#1}{#3}}%
2108 }%
2109 \def\XINT_flpower_loop_throwaway #1#2#3#4%
2110 {%
2111 \XINT_flpower_loop {#1}{#2}{#3}%
2112 }%

```

37.72 \xintFloatSqrt

1.08

```

2113 \def\xintFloatSqrt {\romannumeral0\xintfloatsqrt }%
2114 \def\xintfloatsqrt #1{\XINT_flsqrt_chkopt \xintfloat #1\Z }%
2115 \def\XINTinFloatSqrt {\romannumeral0\XINTinfloatsqrt }%
2116 \def\XINTinfloatsqrt #1{\XINT_flsqrt_chkopt \XINT_inFloat #1\Z }%
2117 \def\XINT_flsqrt_chkopt #1#2%
2118 {%
2119 \ifx [#2\expandafter\XINT_flsqrt_opt
2120 \else\expandafter\XINT_flsqrt_noopt
2121 \fi #1#2%
2122 }%

```

```

2123 \def\XINT_flgqrt_noopt #1#2\Z
2124 {%
2125   #1[\XINTdigits]{\XINT_FL_sqrt \XINTdigits {#2}}%
2126 }%
2127 \def\XINT_flgqrt_opt #1[\Z #2]#3%
2128 {%
2129   #1[#2]{\XINT_FL_sqrt {#2}{#3}}%
2130 }%
2131 \def\XINT_FL_sqrt #1%
2132 {%
2133   \ifnum\numexpr #1<\xint_c_xviii
2134     \xint_afterfi {\XINT_FL_sqrt_a\xint_c_xviii}%
2135   \else
2136     \xint_afterfi {\XINT_FL_sqrt_a {#1+\xint_c_i}}%
2137   \fi
2138 }%
2139 \def\XINT_FL_sqrt_a #1#2%
2140 {%
2141   \expandafter\XINT_FL_sqrt_checkifzeroorneg
2142   \romannumeral0\XINT_inFloat [#1]{#2}%
2143 }%
2144 \def\XINT_FL_sqrt_checkifzeroorneg #1%
2145 {%
2146   \xint_UDzerominusfork
2147   #1-\dummy \XINT_FL_sqrt_iszero
2148   0#1\dummy \XINT_FL_sqrt_isneg
2149   0-\dummy {\XINT_FL_sqrt_b #1}%
2150   \krof
2151 }%
2152 \def\XINT_FL_sqrt_iszero #1[#2]{0[0]}%
2153 \def\XINT_FL_sqrt_isneg #1[#2]{\xintError:RootOfNegative 0[0]}%
2154 \def\XINT_FL_sqrt_b #1[#2]%
2155 {%
2156   \ifodd #2
2157     \xint_afterfi{\XINT_FL_sqrt_c 01}%
2158   \else
2159     \xint_afterfi{\XINT_FL_sqrt_c {}0}%
2160   \fi
2161   {#1}{#2}%
2162 }%
2163 \def\XINT_FL_sqrt_c #1#2#3#4%
2164 {%
2165   \expandafter\XINT_flgqrt\expandafter {\the\numexpr #4-#2}{#3#1}%
2166 }%
2167 \def\XINT_flgqrt #1#2%
2168 {%
2169   \expandafter\XINT_sqrt_a
2170   \expandafter{\romannumeral0\xintlength {#2}}\XINT_flgqrt_big_d {#2}{#1}%
2171 }%

```

```

2172 \def\XINT_flsqrt_big_d #1\or #2\fi #3%
2173 {%
2174   \fi
2175   \ifodd #3
2176     \xint_afterfi{\expandafter\XINT_flsqrt_big_eB}%
2177   \else
2178     \xint_afterfi{\expandafter\XINT_flsqrt_big_eA}%
2179   \fi
2180   \expandafter {\the\numexpr (#3-\xint_c_i)/\xint_c_ii }{#1}%
2181 }%
2182 \def\XINT_flsqrt_big_eA #1#2#3%
2183 {%
2184   \XINT_flsqrt_big_eA_a #3\Z {#2}{#1}{#3}%
2185 }%
2186 \def\XINT_flsqrt_big_eA_a #1#2#3#4#5#6#7#8#9\Z
2187 {%
2188   \XINT_flsqrt_big_eA_b {#1#2#3#4#5#6#7#8}%
2189 }%
2190 \def\XINT_flsqrt_big_eA_b #1#2%
2191 {%
2192   \expandafter\XINT_flsqrt_big_f
2193   \romannumeral0\XINT_flsqrt_small_e {#2001}{#1}%
2194 }%
2195 \def\XINT_flsqrt_big_eB #1#2#3%
2196 {%
2197   \XINT_flsqrt_big_eB_a #3\Z {#2}{#1}{#3}%
2198 }%
2199 \def\XINT_flsqrt_big_eB_a #1#2#3#4#5#6#7#8#9%
2200 {%
2201   \XINT_flsqrt_big_eB_b {#1#2#3#4#5#6#7#8#9}%
2202 }%
2203 \def\XINT_flsqrt_big_eB_b #1#2\Z #3%
2204 {%
2205   \expandafter\XINT_flsqrt_big_f
2206   \romannumeral0\XINT_flsqrt_small_e {#30001}{#1}%
2207 }%
2208 \def\XINT_flsqrt_small_e #1#2%
2209 {%
2210   \expandafter\XINT_flsqrt_small_f\expandafter
2211   {\the\numexpr #1*#1-#2-\xint_c_i}{#1}%
2212 }%
2213 \def\XINT_flsqrt_small_f #1#2%
2214 {%
2215   \expandafter\XINT_flsqrt_small_g\expandafter
2216   {\the\numexpr (#1+#2)/(2*#2)-\xint_c_i }{#1}{#2}%
2217 }%
2218 \def\XINT_flsqrt_small_g #1%
2219 {%
2220   \ifnum #1>\xint_c_

```

```

2221 \expandafter\XINT_flsqrt_small_h
2222 \else
2223 \expandafter\XINT_flsqrt_small_end
2224 \fi
2225 {#1}%
2226 }%
2227 \def\XINT_flsqrt_small_h #1#2#3%
2228 {%
2229 \expandafter\XINT_flsqrt_small_f\expandafter
2230 {\the\numexpr #2-\xint_c_ii*#1*#3+#1*#1\expandafter}\expandafter
2231 {\the\numexpr #3-#1}%
2232 }%
2233 \def\XINT_flsqrt_small_end #1#2#3%
2234 {%
2235 \expandafter\space\expandafter
2236 {\the\numexpr \xint_c_i+#3*\xint_c_x^iv-
2237 (#2*\xint_c_x^iv+#3)/(\xint_c_ii*#3)}%
2238 }%
2239 \def\XINT_flsqrt_big_f #1%
2240 {%
2241 \expandafter\XINT_flsqrt_big_fa\expandafter
2242 {\romannumeral0\xintiisqr {#1}}{#1}%
2243 }%
2244 \def\XINT_flsqrt_big_fa #1#2#3#4%
2245 {%
2246 \expandafter\XINT_flsqrt_big_fb\expandafter
2247 {\romannumeral0\XINT_dsx_addzerosnofuss
2248 {\numexpr #3-\xint_c_viii\relax}{#2}}%
2249 {\romannumeral0\xintiisub
2250 {\XINT_dsx_addzerosnofuss
2251 {\numexpr \xint_c_ii*(#3-\xint_c_viii)\relax}{#1}}{#4}}%
2252 {#3}%
2253 }%
2254 \def\XINT_flsqrt_big_fb #1#2%
2255 {%
2256 \expandafter\XINT_flsqrt_big_g\expandafter {#2}{#1}%
2257 }%
2258 \def\XINT_flsqrt_big_g #1#2%
2259 {%
2260 \expandafter\XINT_flsqrt_big_j
2261 \romannumeral0\xintiidivision
2262 {#1}{\romannumeral0\XINT_dbl_pos #2\R\R\R\R\R\R\R\Z \W\W\W\W\W\W\W }{#2}%
2263 }%
2264 \def\XINT_flsqrt_big_j #1%
2265 {%
2266 \ifcase\XINT_Sgn {#1}
2267 \expandafter \XINT_flsqrt_big_end_a
2268 \or \expandafter \XINT_flsqrt_big_k
2269 \fi {#1}%

```

```

2270 }%
2271 \def\XINT_flsqrt_big_k #1#2#3%
2272 {%
2273   \expandafter\XINT_flsqrt_big_l\expandafter
2274   {\romannumeral0\XINT_sub_pre {#3}{#1}}%
2275   {\romannumeral0\xintiiadd {#2}{\romannumeral0\XINT_sqr {#1}}}%
2276 }%
2277 \def\XINT_flsqrt_big_l #1#2%
2278 {%
2279   \expandafter\XINT_flsqrt_big_g\expandafter
2280   {#2}{#1}%
2281 }%
2282 \def\XINT_flsqrt_big_end_a #1#2#3#4#5%
2283 {%
2284   \expandafter\XINT_flsqrt_big_end_b\expandafter
2285   {\the\numexpr -#4+#5/\xint_c_ii\expandafter}\expandafter
2286   {\romannumeral0\xintiisub
2287     {\XINT_dsx_addzerosnofuss {#4}{#3}}%
2288     {\xintHalf{\xintiiQuo{\XINT_dsx_addzerosnofuss {#4}{#2}}{#3}}}%
2289 }%
2290 \def\XINT_flsqrt_big_end_b #1#2{#2[#1]}%
2291 \XINT_restorecatcodes_endinput%

```

38 Package *xintseries* implementation

The commenting is currently (2013/11/28) very sparse.

Contents

.1	Catcodes, ε -T _E X and reload detection .. 325	.8	\xintPowerSeriesX..... 330
.2	Confirmation of <i>xintfrac</i> loading ... 326	.9	\xintRationalSeries..... 330
.3	Catcodes 327	.10	\xintRationalSeriesX..... 331
.4	Package identification 327	.11	\xintFxFtPowerSeries..... 332
.5	\xintSeries..... 327	.12	\xintFxFtPowerSeriesX..... 333
.6	\xintiSeries..... 328	.13	\xintFloatPowerSeries..... 334
.7	\xintPowerSeries..... 329	.14	\xintFloatPowerSeriesX..... 335

38.1 Catcodes, ε -T_EX 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.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2 \catcode13=5 % ^^M
3 \endlinechar=13 %

```

```

4 \catcode123=1 % {
5 \catcode125=2 % }
6 \catcode64=11 % @
7 \catcode35=6 % #
8 \catcode44=12 % ,
9 \catcode45=12 % -
10 \catcode46=12 % .
11 \catcode58=12 % :
12 \def\space { }%
13 \let\z\endgroup
14 \expandafter\let\expandafter\x\csname ver@xintseries.sty\endcsname
15 \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
16 \expandafter
17 \ifx\csname PackageInfo\endcsname\relax
18 \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
19 \else
20 \def\y#1#2{\PackageInfo{#1}{#2}}%
21 \fi
22 \expandafter
23 \ifx\csname numexpr\endcsname\relax
24 \y{xintseries}{numexpr not available, aborting input}%
25 \aftergroup\endinput
26 \else
27 \ifx\x\relax % plain-TeX, first loading of xintseries.sty
28 \ifx\w\relax % but xintfrac.sty not yet loaded.
29 \y{xintseries}{Package xintfrac is required}%
30 \y{xintseries}{Will try \string\input\space xintfrac.sty}%
31 \def\z{\endgroup\input xintfrac.sty\relax}%
32 \fi
33 \else
34 \def\empty {}%
35 \ifx\x\empty % LaTeX, first loading,
36 % variable is initialized, but \ProvidesPackage not yet seen
37 \ifx\w\relax % xintfrac.sty not yet loaded.
38 \y{xintseries}{Package xintfrac is required}%
39 \y{xintseries}{Will try \string\RequirePackage{xintfrac}}%
40 \def\z{\endgroup\RequirePackage{xintfrac}}%
41 \fi
42 \else
43 \y{xintseries}{I was already loaded, aborting input}%
44 \aftergroup\endinput
45 \fi
46 \fi
47 \fi
48 \z%

```

38.2 Confirmation of *xintfrac* loading

```

49 \begingroup\catcode61\catcode48\catcode32=10\relax%

```



```

50 \catcode13=5      % ^^M
51 \endlinechar=13 %
52 \catcode123=1     % {
53 \catcode125=2     % }
54 \catcode64=11     % @
55 \catcode35=6      % #
56 \catcode44=12     % ,
57 \catcode45=12     % -
58 \catcode46=12     % .
59 \catcode58=12     % :
60 \ifdefined\PackageInfo
61   \def\y#1#2{\PackageInfo{#1}{#2}}%
62   \else
63     \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
64   \fi
65 \def\empty {}%
66 \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
67 \ifx\w\relax % Plain TeX, user gave a file name at the prompt
68   \y{xintseries}{Loading of package xintfrac failed, aborting input}%
69   \aftergroup\endinput
70 \fi
71 \ifx\w\empty % LaTeX, user gave a file name at the prompt
72   \y{xintseries}{Loading of package xintfrac failed, aborting input}%
73   \aftergroup\endinput
74 \fi
75 \endgroup%

```

38.3 Catcodes

```
76 \XINTsetupcatcodes%
```

38.4 Package identification

```

77 \XINT_providespackage
78 \ProvidesPackage{xintseries}%
79 [2013/11/28 v1.09h Expandable partial sums with xint package (jfb)]%

```

38.5 \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.

```

80 \def\xintSeries {\romannumeral0\xintseries }%
81 \def\xintseries #1#2%
82 {%
83   \expandafter\XINT_series\expandafter
84   {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
85 }%
86 \def\XINT_series #1#2#3%
87 {%

```

```

88 \ifnum #2<#1
89 \xint_afterfi { 0/1[0]}%
90 \else
91 \xint_afterfi {\XINT_series_loop {#1}{0}{#2}{#3}}%
92 \fi
93}%
94\def\XINT_series_loop #1#2#3#4%
95{%
96 \ifnum #3>#1 \else \XINT_series_exit \fi
97 \expandafter\XINT_series_loop\expandafter
98 {\the\numexpr #1+1\expandafter }\expandafter
99 {\romannumeral0\xintadd {#2}{#4{#1}}}%
100 {#3}{#4}%
101}%
102\def\XINT_series_exit \fi #1#2#3#4#5#6#7#8%
103{%
104 \fi\xint_gobble_ii #6%
105}%

```

38.6 \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.

```

106\def\xintiSeries {\romannumeral0\xintiseries}%
107\def\xintiseries #1#2%
108{%
109 \expandafter\XINT_iseries\expandafter
110 {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
111}%
112\def\XINT_iseries #1#2#3%
113{%
114 \ifnum #2<#1
115 \xint_afterfi { 0}%
116 \else
117 \xint_afterfi {\XINT_iseries_loop {#1}{0}{#2}{#3}}%
118 \fi
119}%
120\def\XINT_iseries_loop #1#2#3#4%
121{%
122 \ifnum #3>#1 \else \XINT_iseries_exit \fi
123 \expandafter\XINT_iseries_loop\expandafter
124 {\the\numexpr #1+1\expandafter }\expandafter
125 {\romannumeral0\xintiiadd {#2}{#4{#1}}}%
126 {#3}{#4}%
127}%
128\def\XINT_iseries_exit \fi #1#2#3#4#5#6#7#8%
129{%

```

```

130   \fi\xint_gobble_ii #6%
131 }%

```

38.7 \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.

```

132 \def\xintPowerSeries {\romannumeral0\xintpowerseries }%
133 \def\xintpowerseries #1#2%
134 {%
135   \expandafter\XINT_powseries\expandafter
136   {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
137 }%
138 \def\XINT_powseries #1#2#3#4%
139 {%
140   \ifnum #2<#1
141     \xint_afterfi { 0/1[0]}%
142   \else
143     \xint_afterfi
144     {\XINT_powseries_loop_i {#3{#2}}{#1}{#2}{#3}{#4}}%
145   \fi
146 }%
147 \def\XINT_powseries_loop_i #1#2#3#4#5%
148 {%
149   \ifnum #3>#2 \else\XINT_powseries_exit_i\fi
150   \expandafter\XINT_powseries_loop_ii\expandafter
151   {\the\numexpr #3-1\expandafter}\expandafter
152   {\romannumeral0\xintmul {#1}{#5}}{#2}{#4}{#5}%
153 }%
154 \def\XINT_powseries_loop_ii #1#2#3#4%
155 {%
156   \expandafter\XINT_powseries_loop_i\expandafter
157   {\romannumeral0\xintadd {#4{#1}}{#2}}{#3}{#1}{#4}%
158 }%
159 \def\XINT_powseries_exit_i\fi #1#2#3#4#5#6#7#8#9%
160 {%
161   \fi \XINT_powseries_exit_ii #6{#7}%
162 }%
163 \def\XINT_powseries_exit_ii #1#2#3#4#5#6%
164 {%
165   \xintmul{\xintPow {#5}{#6}}{#4}%
166 }%

```

38.8 \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.

```

167 \def\xintPowerSeriesX {\romannumeral0\xintpowerseriesx }%
168 \def\xintpowerseriesx #1#2%
169 {%
170   \expandafter\XINT_powseriesx\expandafter
171   {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
172 }%
173 \def\XINT_powseriesx #1#2#3#4%
174 {%
175   \ifnum #2<#1
176     \xint_afterfi { 0/1[0]}%
177   \else
178     \xint_afterfi
179     {\expandafter\XINT_powseriesx_pre\expandafter
180      {\romannumeral-'0#4}{#1}{#2}{#3}%
181     }%
182   \fi
183 }%
184 \def\XINT_powseriesx_pre #1#2#3#4%
185 {%
186   \XINT_powseries_loop_i {#4}{#3}}{#2}{#3}{#4}{#1}%
187 }%

```

38.9 \xintRationalSeries

This computes $F(a)+\dots+F(b)$ 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.

```

188 \def\xintRationalSeries {\romannumeral0\xintratseries }%
189 \def\xintratseries #1#2%
190 {%
191   \expandafter\XINT_ratseries\expandafter
192   {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
193 }%
194 \def\XINT_ratseries #1#2#3#4%
195 {%
196   \ifnum #2<#1

```

```

197     \xint_afterfi { 0/1[0]}%
198   \else
199     \xint_afterfi
200     {\XINT_ratseries_loop {#2}{1}{#1}{#4}{#3}}%
201   \fi
202 }%
203 \def\XINT_ratseries_loop #1#2#3#4%
204 {%
205   \ifnum #1>#3 \else\XINT_ratseries_exit_i\fi
206   \expandafter\XINT_ratseries_loop\expandafter
207   {\the\numexpr #1-1\expandafter}\expandafter
208   {\romannumeral0\xintadd {1}{\xintMul {#2}{#4{#1}}}}{#3}{#4}%
209 }%
210 \def\XINT_ratseries_exit_i\fi #1#2#3#4#5#6#7#8%
211 {%
212   \fi \XINT_ratseries_exit_ii #6%
213 }%
214 \def\XINT_ratseries_exit_ii #1#2#3#4#5%
215 {%
216   \XINT_ratseries_exit_iii #5%
217 }%
218 \def\XINT_ratseries_exit_iii #1#2#3#4%
219 {%
220   \xintmul{#2}{#4}%
221 }%

```

38.10 \xintRationalSeriesX

a, b, initial, ratiofunction, x

This computes $F(a, x) + \dots + F(b, x)$ 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.

```

222 \def\xintRationalSeriesX {\romannumeral0\xintratseriesx }%
223 \def\xintratseriesx #1#2%
224 {%
225   \expandafter\XINT_ratseriesx\expandafter
226   {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
227 }%
228 \def\XINT_ratseriesx #1#2#3#4#5%
229 {%
230   \ifnum #2<#1
231     \xint_afterfi { 0/1[0]}%
232   \else
233     \xint_afterfi

```

```

234      {\expandafter\XINT_ratseriesx_pre\expandafter
235        {\romannumeral-‘0#5}{#2}{#1}{#4}{#3}%
236      }%
237    \fi
238 }%
239 \def\XINT_ratseriesx_pre #1#2#3#4#5%
240 {%
241   \XINT_ratseries_loop {#2}{1}{#3}{#4}{#1}}{#5{#1}}%
242 }%

```

38.11 \xintFxFtPowerSeries

I am not too 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.

```

243 \def\xintFxFtPowerSeries {\romannumeral0\xintfxptpowerseries }%
244 \def\xintfxptpowerseries #1#2%
245 {%
246   \expandafter\XINT_fppowseries\expandafter
247   {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
248 }%
249 \def\XINT_fppowseries #1#2#3#4#5%
250 {%
251   \ifnum #2<#1
252     \xint_afterfi { 0}%
253   \else
254     \xint_afterfi
255     {\expandafter\XINT_fppowseries_loop_pre\expandafter
256       {\romannumeral0\xinttrunc {#5}{\xintPow {#4}{#1}}}%
257       {#1}{#4}{#2}{#3}{#5}%
258     }%
259   \fi
260 }%
261 \def\XINT_fppowseries_loop_pre #1#2#3#4#5#6%
262 {%
263   \ifnum #4>#2 \else\XINT_fppowseries_dont_i \fi
264   \expandafter\XINT_fppowseries_loop_i\expandafter
265   {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
266   {\romannumeral0\xintitrunc {#6}{\xintMul {#5{#2}}{#1}}}%
267   {#1}{#3}{#4}{#5}{#6}%
268 }%
269 \def\XINT_fppowseries_dont_i \fi\expandafter\XINT_fppowseries_loop_i
270   {\fi \expandafter\XINT_fppowseries_dont_ii }%
271 \def\XINT_fppowseries_dont_ii #1#2#3#4#5#6#7{\xinttrunc {#7}{#2[-#7]}}%
272 \def\XINT_fppowseries_loop_i #1#2#3#4#5#6#7%
273 {%
274   \ifnum #5>#1 \else \XINT_fppowseries_exit_i \fi

```

```

275 \expandafter\XINT_fppowseries_loop_ii\expandafter
276 {\romannumeral0\xinttrunc {#7}{\xintMul {#3}{#4}}}%
277 {#1}{#4}{#2}{#5}{#6}{#7}%
278 }%
279 \def\XINT_fppowseries_loop_ii #1#2#3#4#5#6#7%
280 {%
281 \expandafter\XINT_fppowseries_loop_i\expandafter
282 {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
283 {\romannumeral0\xintiiadd {#4}{\xintiTrunc {#7}{\xintMul {#6{#2}}{#1}}}}%
284 {#1}{#3}{#5}{#6}{#7}%
285 }%
286 \def\XINT_fppowseries_exit_i\fi\expandafter\XINT_fppowseries_loop_ii
287 {\fi \expandafter\XINT_fppowseries_exit_ii }%
288 \def\XINT_fppowseries_exit_ii #1#2#3#4#5#6#7%
289 {%
290 \xinttrunc {#7}
291 {\xintiiadd {#4}{\xintiTrunc {#7}{\xintMul {#6{#2}}{#1}}}{-#7}}%
292 }%

```

38.12 \xintFxFtPowerSeriesX

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.

```

293 \def\xintFxFtPowerSeriesX {\romannumeral0\xintfxptpowerseriesx }%
294 \def\xintfxptpowerseriesx #1#2%
295 {%
296 \expandafter\XINT_fppowseriesx\expandafter
297 {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
298 }%
299 \def\XINT_fppowseriesx #1#2#3#4#5%
300 {%
301 \ifnum #2<#1
302 \xint_afterfi { 0}%
303 \else
304 \xint_afterfi
305 {\expandafter \XINT_fppowseriesx_pre \expandafter
306 {\romannumeral- '0#4}{#1}{#2}{#3}{#5}%
307 }%
308 \fi
309 }%
310 \def\XINT_fppowseriesx_pre #1#2#3#4#5%
311 {%
312 \expandafter\XINT_fppowseries_loop_pre\expandafter
313 {\romannumeral0\xinttrunc {#5}{\xintPow {#1}{#2}}}%
314 {#2}{#1}{#3}{#4}{#5}%
315 }%

```

38.13 \xintFloatPowerSeries

1.08a. I still have to re-visit \xintFxFtPowerSeries; temporarily I just adapted the code to the case of floats.

```

316 \def\xintFloatPowerSeries {\romannumeral0\xintfloatpowerseries }%
317 \def\xintfloatpowerseries #1{\XINT_flpowseries_chkopt #1\Z }%
318 \def\XINT_flpowseries_chkopt #1%
319 {%
320   \ifx [#1\expandafter\XINT_flpowseries_opt
321     \else\expandafter\XINT_flpowseries_noopt
322   \fi
323   #1%
324 }%
325 \def\XINT_flpowseries_noopt #1\Z #2%
326 {%
327   \expandafter\XINT_flpowseries\expandafter
328   {\the\numexpr #1\expandafter}\expandafter
329   {\the\numexpr #2}\XINTdigits
330 }%
331 \def\XINT_flpowseries_opt [\Z #1]#2#3%
332 {%
333   \expandafter\XINT_flpowseries\expandafter
334   {\the\numexpr #2\expandafter}\expandafter
335   {\the\numexpr #3\expandafter}{\the\numexpr #1}%
336 }%
337 \def\XINT_flpowseries #1#2#3#4#5%
338 {%
339   \ifnum #2<#1
340     \xint_afterfi { 0.e0}%
341   \else
342     \xint_afterfi
343     {\expandafter\XINT_flpowseries_loop_pre\expandafter
344      {\romannumeral0\XINTinfloatpow [#3]{#5}{#1}}%
345      {#1}{#5}{#2}{#4}{#3}%
346     }%
347   \fi
348 }%
349 \def\XINT_flpowseries_loop_pre #1#2#3#4#5#6%
350 {%
351   \ifnum #4>#2 \else\XINT_flpowseries_dont_i \fi
352   \expandafter\XINT_flpowseries_loop_i\expandafter
353   {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
354   {\romannumeral0\XINTinfloatmul [#6]{#5{#2}}{#1}}%
355   {#1}{#3}{#4}{#5}{#6}%
356 }%
357 \def\XINT_flpowseries_dont_i \fi\expandafter\XINT_flpowseries_loop_i
358   {\fi \expandafter\XINT_flpowseries_dont_ii }%
359 \def\XINT_flpowseries_dont_ii #1#2#3#4#5#6#7{\xintfloat [#7]{#2}}%

```



```

360 \def\XINT_flpowseries_loop_i #1#2#3#4#5#6#7%
361 {%
362   \ifnum #5>#1 \else \XINT_flpowseries_exit_i \fi
363   \expandafter\XINT_flpowseries_loop_ii\expandafter
364   {\romannumeral0\XINTinfloatmul [#7]{#3}{#4}}%
365   {#1}{#4}{#2}{#5}{#6}{#7}%
366 }%
367 \def\XINT_flpowseries_loop_ii #1#2#3#4#5#6#7%
368 {%
369   \expandafter\XINT_flpowseries_loop_i\expandafter
370   {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
371   {\romannumeral0\XINTinfloatadd [#7]{#4}%
372    {\XINTinfloatmul [#7]{#6{#2}}{#1}}}%
373   {#1}{#3}{#5}{#6}{#7}%
374 }%
375 \def\XINT_flpowseries_exit_i\fi\expandafter\XINT_flpowseries_loop_ii
376   {\fi \expandafter\XINT_flpowseries_exit_ii }%
377 \def\XINT_flpowseries_exit_ii #1#2#3#4#5#6#7%
378 {%
379   \xintfloatadd [#7]{#4}{\XINTinfloatmul [#7]{#6{#2}}{#1}}%
380 }%

```

38.14 \xintFloatPowerSeriesX

1.08a

```

381 \def\xintFloatPowerSeriesX {\romannumeral0\xintfloatpowerseriesx }%
382 \def\xintfloatpowerseriesx #1{\XINT_flpowseriesx_chkopt #1\Z }%
383 \def\XINT_flpowseriesx_chkopt #1%
384 {%
385   \ifx [#1\expandafter\XINT_flpowseriesx_opt
386     \else\expandafter\XINT_flpowseriesx_noopt
387   \fi
388   #1%
389 }%
390 \def\XINT_flpowseriesx_noopt #1\Z #2%
391 {%
392   \expandafter\XINT_flpowseriesx\expandafter
393   {\the\numexpr #1\expandafter}\expandafter
394   {\the\numexpr #2}\XINTdigits
395 }%
396 \def\XINT_flpowseriesx_opt [\Z #1]#2#3%
397 {%
398   \expandafter\XINT_flpowseriesx\expandafter
399   {\the\numexpr #2\expandafter}\expandafter
400   {\the\numexpr #3\expandafter}{\the\numexpr #1}%
401 }%
402 \def\XINT_flpowseriesx #1#2#3#4#5%
403 {%

```

```

404 \ifnum #2<#1
405 \xint_afterfi { 0.e0}%
406 \else
407 \xint_afterfi
408 {\expandafter \XINT_flpowseriesx_pre \expandafter
409 {\romannumeral-‘0#5}{#1}{#2}{#4}{#3}%
410 }%
411 \fi
412 }%
413 \def\XINT_flpowseriesx_pre #1#2#3#4#5%
414 {%
415 \expandafter\XINT_flpowseries_loop_pre\expandafter
416 {\romannumeral0\XINTinfloatpow [#5]{#1}{#2}}%
417 {#2}{#1}{#3}{#4}{#5}%
418 }%
419 \XINT_restorecatcodes_endinput%

```

39 Package **xintcfrac** implementation

The commenting is currently (2013/11/28) very sparse.

Contents

.1	Catcodes, ε - \TeX and reload detection ..	336	.15	\xintiCstoF	346
.2	Confirmation of xintfrac loading ...	338	.16	\xintGctoF	346
.3	Catcodes	338	.17	\xintiGctoF	347
.4	Package identification	338	.18	\xintCstoCv	348
.5	\xintCFrac	338	.19	\xintiCstoCv	349
.6	\xintGCFrac	340	.20	\xintGctoCv	350
.7	\xintGctoGCx	341	.21	\xintiGctoCv	351
.8	\xintFtoCs	341	.22	\xintCntoF	353
.9	\xintFtoCx	342	.23	\xintGCntoF	353
.10	\xintFtoGC	343	.24	\xintCntoCs	354
.11	\xintFtoCC	343	.25	\xintCntoGC	355
.12	\xintFtoCv	344	.26	\xintGCntoGC	356
.13	\xintFtoCCv	345	.27	\xintCstoGC	356
.14	\xintCstoF	345	.28	\xintGctoGC	357

39.1 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.

```
1 \begingroup\catcode61\catcode48\catcode32=10\relax%
```

```

2 \catcode13=5 % ^^M
3 \endlinechar=13 %
4 \catcode123=1 % {
5 \catcode125=2 % }
6 \catcode64=11 % @
7 \catcode35=6 % #
8 \catcode44=12 % ,
9 \catcode45=12 % -
10 \catcode46=12 % .
11 \catcode58=12 % :
12 \def\space { }%
13 \let\z\endgroup
14 \expandafter\let\expandafter\x\csname ver@xintcfrac.sty\endcsname
15 \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
16 \expandafter
17 \ifx\csname PackageInfo\endcsname\relax
18 \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
19 \else
20 \def\y#1#2{\PackageInfo{#1}{#2}}%
21 \fi
22 \expandafter
23 \ifx\csname numexpr\endcsname\relax
24 \y{xintcfrac}{numexpr not available, aborting input}%
25 \aftergroup\endinput
26 \else
27 \ifx\x\relax % plain-TeX, first loading of xintcfrac.sty
28 \ifx\w\relax % but xintfrac.sty not yet loaded.
29 \y{xintcfrac}{Package xintfrac is required}%
30 \y{xintcfrac}{Will try \string\input\space xintfrac.sty}%
31 \def\z{\endgroup\input xintfrac.sty\relax}%
32 \fi
33 \else
34 \def\empty {}%
35 \ifx\x\empty % LaTeX, first loading,
36 % variable is initialized, but \ProvidesPackage not yet seen
37 \ifx\w\relax % xintfrac.sty not yet loaded.
38 \y{xintcfrac}{Package xintfrac is required}%
39 \y{xintcfrac}{Will try \string\RequirePackage{xintfrac}}%
40 \def\z{\endgroup\RequirePackage{xintfrac}}%
41 \fi
42 \else
43 \y{xintcfrac}{I was already loaded, aborting input}%
44 \aftergroup\endinput
45 \fi
46 \fi
47 \fi
48 \z%

```

39.2 Confirmation of *xintcfrac* loading

```

49 \begingroup\catcode61\catcode48\catcode32=10\relax%
50 \catcode13=5 % ^^M
51 \endlinechar=13 %
52 \catcode123=1 % {
53 \catcode125=2 % }
54 \catcode64=11 % @
55 \catcode35=6 % #
56 \catcode44=12 % ,
57 \catcode45=12 % -
58 \catcode46=12 % .
59 \catcode58=12 % :
60 \ifdefined\PackageInfo
61 \def\y#1#2{\PackageInfo{#1}{#2}}%
62 \else
63 \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
64 \fi
65 \def\empty {}%
66 \expandafter\let\expandafter\w\csname ver@xintcfrac.sty\endcsname
67 \ifx\w\relax % Plain TeX, user gave a file name at the prompt
68 \y{xintcfrac}{Loading of package xintcfrac failed, aborting input}%
69 \aftergroup\endinput
70 \fi
71 \ifx\w\empty % LaTeX, user gave a file name at the prompt
72 \y{xintcfrac}{Loading of package xintcfrac failed, aborting input}%
73 \aftergroup\endinput
74 \fi
75 \endgroup%

```

39.3 Catcodes

```

76 \XINTsetupcatcodes%

```

39.4 Package identification

```

77 \XINT_providespackage
78 \ProvidesPackage{xintcfrac}%
79 [2013/11/28 v1.09h Expandable continued fractions with xint package (jfb)]%

```

39.5 *\xintCFrac*

```

80 \def\xintCFrac {\romannumeral0\xintcfrac}%
81 \def\xintcfrac #1%
82 {%
83 \XINT_cfrac_opt_a #1\Z
84}%
85 \def\XINT_cfrac_opt_a #1%
86 {%
87 \ifx[#1\XINT_cfrac_opt_b\fi \XINT_cfrac_noopt #1%
88}%
89 \def\XINT_cfrac_noopt #1\Z

```

```

90 {%
91   \expandafter\XINT_cfrac_A\romannumeral0\xintraewithzeros {#1}\Z
92   \relax\relax
93 }%
94 \def\XINT_cfrac_opt_b\fi\XINT_cfrac_noopt [\Z #1]%
95 {%
96   \fi\csname XINT_cfrac_opt#1\endcsname
97 }%
98 \def\XINT_cfrac_optl #1%
99 {%
100   \expandafter\XINT_cfrac_A\romannumeral0\xintraewithzeros {#1}\Z
101   \relax\hfill
102 }%
103 \def\XINT_cfrac_optc #1%
104 {%
105   \expandafter\XINT_cfrac_A\romannumeral0\xintraewithzeros {#1}\Z
106   \relax\relax
107 }%
108 \def\XINT_cfrac_optr #1%
109 {%
110   \expandafter\XINT_cfrac_A\romannumeral0\xintraewithzeros {#1}\Z
111   \hfill\relax
112 }%
113 \def\XINT_cfrac_A #1/#2\Z
114 {%
115   \expandafter\XINT_cfrac_B\romannumeral0\xintiidivision {#1}{#2}{#2}%
116 }%
117 \def\XINT_cfrac_B #1#2%
118 {%
119   \XINT_cfrac_C #2\Z {#1}%
120 }%
121 \def\XINT_cfrac_C #1%
122 {%
123   \xint_gob_til_zero #1\XINT_cfrac_integer 0\XINT_cfrac_D #1%
124 }%
125 \def\XINT_cfrac_integer 0\XINT_cfrac_D 0#1\Z #2#3#4#5{ #2}%
126 \def\XINT_cfrac_D #1\Z #2#3{\XINT_cfrac_loop_a {#1}{#3}{#1}{{#2}}}%
127 \def\XINT_cfrac_loop_a
128 {%
129   \expandafter\XINT_cfrac_loop_d\romannumeral0\XINT_div_prepare
130 }%
131 \def\XINT_cfrac_loop_d #1#2%
132 {%
133   \XINT_cfrac_loop_e #2.{#1}%
134 }%
135 \def\XINT_cfrac_loop_e #1%
136 {%
137   \xint_gob_til_zero #1\xint_cfrac_loop_exit0\XINT_cfrac_loop_f #1%
138 }%

```

```

139 \def\XINT_cfrac_loop_f #1.#2#3#4%
140 {%
141   \XINT_cfrac_loop_a {#1}{#3}{#1}{{#2}#4}%
142 }%
143 \def\xint_cfrac_loop_exit0\XINT_cfrac_loop_f #1.#2#3#4#5#6%
144   {\XINT_cfrac_T #5#6{#2}#4\Z }%
145 \def\XINT_cfrac_T #1#2#3#4%
146 {%
147   \xint_gob_til_Z #4\XINT_cfrac_end\Z\XINT_cfrac_T #1#2{#4+\cfrac{#1#2}{#3}}%
148 }%
149 \def\XINT_cfrac_end\Z\XINT_cfrac_T #1#2#3%
150 {%
151   \XINT_cfrac_end_b #3%
152 }%
153 \def\XINT_cfrac_end_b \Z+\cfrac#1#2{ #2}%

```

39.6 \xintGCFrac

```

154 \def\xintGCFrac {\romannumeral0\xintgcfrac }%
155 \def\xintgcfrac #1{\XINT_gcfrac_opt_a #1\Z }%
156 \def\XINT_gcfrac_opt_a #1%
157 {%
158   \ifx[#1\XINT_gcfrac_opt_b\fi \XINT_gcfrac_noopt #1%
159 }%
160 \def\XINT_gcfrac_noopt #1\Z
161 {%
162   \XINT_gcfrac #1+\W/\relax\relax
163 }%
164 \def\XINT_gcfrac_opt_b\fi\XINT_gcfrac_noopt [\Z #1]%
165 {%
166   \fi\csname XINT_gcfrac_opt#1\endcsname
167 }%
168 \def\XINT_gcfrac_optl #1%
169 {%
170   \XINT_gcfrac #1+\W/\relax\hfill
171 }%
172 \def\XINT_gcfrac_optc #1%
173 {%
174   \XINT_gcfrac #1+\W/\relax\relax
175 }%
176 \def\XINT_gcfrac_optr #1%
177 {%
178   \XINT_gcfrac #1+\W/\hfill\relax
179 }%
180 \def\XINT_gcfrac
181 {%
182   \expandafter\XINT_gcfrac_enter\romannumeral-‘0%
183 }%
184 \def\XINT_gcfrac_enter {\XINT_gcfrac_loop {}}%
185 \def\XINT_gcfrac_loop #1#2+#3/%

```

```

186 {%
187   \xint_gob_til_W #3\XINT_gcfrac_endloop\W
188   \XINT_gcfrac_loop {{{#3}{#2}#1}%
189 }%
190 \def\XINT_gcfrac_endloop\W\XINT_gcfrac_loop #1#2#3%
191 {%
192   \XINT_gcfrac_T #2#3#1\Z\Z
193 }%
194 \def\XINT_gcfrac_T #1#2#3#4{\XINT_gcfrac_U #1#2{\xintFrac{#4}}}%
195 \def\XINT_gcfrac_U #1#2#3#4#5%
196 {%
197   \xint_gob_til_Z #5\XINT_gcfrac_end\Z\XINT_gcfrac_U
198   #1#2{\xintFrac{#5}%
199     \ifcase\xintSgn{#4}
200     +\or+\else-\fi
201     \cfrac{#1\xintFrac{\xintAbs{#4}}#2}{#3}}%
202 }%
203 \def\XINT_gcfrac_end\Z\XINT_gcfrac_U #1#2#3%
204 {%
205   \XINT_gcfrac_end_b #3%
206 }%
207 \def\XINT_gcfrac_end_b #1\cfrac#2#3{ #3}%

```

39.7 \xintGctoGCx

```

208 \def\xintGctoGCx {\romannumeral0\xintgctogcx }%
209 \def\xintgctogcx #1#2#3%
210 {%
211   \expandafter\XINT_gctgcx_start\expandafter {\romannumeral-‘0#3}{#1}{#2}%
212 }%
213 \def\XINT_gctgcx_start #1#2#3{\XINT_gctgcx_loop_a }{{#2}{#3}#1+\W/}%
214 \def\XINT_gctgcx_loop_a #1#2#3#4+#5/%
215 {%
216   \xint_gob_til_W #5\XINT_gctgcx_end\W
217   \XINT_gctgcx_loop_b {{#1}{#4}}{#2{#5}#3}{#2}{#3}%
218 }%
219 \def\XINT_gctgcx_loop_b #1#2%
220 {%
221   \XINT_gctgcx_loop_a {#1#2}%
222 }%
223 \def\XINT_gctgcx_end\W\XINT_gctgcx_loop_b #1#2#3#4{ #1}%

```

39.8 \xintFtoCs

```

224 \def\xintFtoCs {\romannumeral0\xintftocs }%
225 \def\xintftocs #1%
226 {%
227   \expandafter\XINT_ftc_A\romannumeral0\xintrawwithzeros {#1}\Z
228 }%
229 \def\XINT_ftc_A #1/#2\Z
230 {%

```

39 Package *xintcfrac* implementation

```

231 \expandafter\XINT_ftc_B\romannumeral0\xintiidivision {#1}{#2}{#2}%
232 }%
233 \def\XINT_ftc_B #1#2%
234 {%
235 \XINT_ftc_C #2.{#1}%
236 }%
237 \def\XINT_ftc_C #1%
238 {%
239 \xint_gob_til_zero #1\XINT_ftc_integer 0\XINT_ftc_D #1%
240 }%
241 \def\XINT_ftc_integer 0\XINT_ftc_D 0#1.#2#3{ #2}%
242 \def\XINT_ftc_D #1.#2#3{\XINT_ftc_loop_a {#1}{#3}{#1}{#2,}}%
243 \def\XINT_ftc_loop_a
244 {%
245 \expandafter\XINT_ftc_loop_d\romannumeral0\XINT_div_prepare
246 }%
247 \def\XINT_ftc_loop_d #1#2%
248 {%
249 \XINT_ftc_loop_e #2.{#1}%
250 }%
251 \def\XINT_ftc_loop_e #1%
252 {%
253 \xint_gob_til_zero #1\xint_ftc_loop_exit0\XINT_ftc_loop_f #1%
254 }%
255 \def\XINT_ftc_loop_f #1.#2#3#4%
256 {%
257 \XINT_ftc_loop_a {#1}{#3}{#1}{#4#2,}%
258 }%
259 \def\xint_ftc_loop_exit0\XINT_ftc_loop_f #1.#2#3#4{ #4#2}%

```

39.9 \xintFtoCx

```

260 \def\xintFtoCx {\romannumeral0\xintftocx }%
261 \def\xintftocx #1#2%
262 {%
263 \expandafter\XINT_ftcx_A\romannumeral0\xintraawithzeros {#2}\Z {#1}%
264 }%
265 \def\XINT_ftcx_A #1/#2\Z
266 {%
267 \expandafter\XINT_ftcx_B\romannumeral0\xintiidivision {#1}{#2}{#2}%
268 }%
269 \def\XINT_ftcx_B #1#2%
270 {%
271 \XINT_ftcx_C #2.{#1}%
272 }%
273 \def\XINT_ftcx_C #1%
274 {%
275 \xint_gob_til_zero #1\XINT_ftcx_integer 0\XINT_ftcx_D #1%
276 }%
277 \def\XINT_ftcx_integer 0\XINT_ftcx_D 0#1.#2#3#4{ #2}%

```



```

278 \def\XINT_ftcx_D #1.#2#3#4{\XINT_ftcx_loop_a {#1}{#3}{#1}{#2#4}{#4}}%
279 \def\XINT_ftcx_loop_a
280 {%
281   \expandafter\XINT_ftcx_loop_d\romannumeral0\XINT_div_prepare
282 }%
283 \def\XINT_ftcx_loop_d #1#2%
284 {%
285   \XINT_ftcx_loop_e #2.{#1}%
286 }%
287 \def\XINT_ftcx_loop_e #1%
288 {%
289   \xint_gob_til_zero #1\xint_ftcx_loop_exit0\XINT_ftcx_loop_f #1%
290 }%
291 \def\XINT_ftcx_loop_f #1.#2#3#4#5%
292 {%
293   \XINT_ftcx_loop_a {#1}{#3}{#1}{#4{#2}#5}{#5}%
294 }%
295 \def\xint_ftcx_loop_exit0\XINT_ftcx_loop_f #1.#2#3#4#5{ #4{#2}}%

```

39.10 \xintFtoGC

```

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

```

39.11 \xintFtoCC

```

298 \def\xintFtoCC {\romannumeral0\xintftocc }%
299 \def\xintftocc #1%
300 {%
301   \expandafter\XINT_ftcc_A\expandafter {\romannumeral0\xintrawwithzeros {#1}}%
302 }%
303 \def\XINT_ftcc_A #1%
304 {%
305   \expandafter\XINT_ftcc_B
306   \romannumeral0\xintrawwithzeros {\xintAdd {1/2[0]}{#1[0]}}\Z {#1[0]}%
307 }%
308 \def\XINT_ftcc_B #1/#2\Z
309 {%
310   \expandafter\XINT_ftcc_C\expandafter {\romannumeral0\xintiiquo {#1}{#2}}%
311 }%
312 \def\XINT_ftcc_C #1#2%
313 {%
314   \expandafter\XINT_ftcc_D\romannumeral0\xintsub {#2}{#1}\Z {#1}%
315 }%
316 \def\XINT_ftcc_D #1%
317 {%
318   \xint_UDzerominusfork
319   #1-\dummy \XINT_ftcc_integer
320   0#1\dummy \XINT_ftcc_En
321   0-\dummy {\XINT_ftcc_Ep #1}%
322   \krof

```

```

323 }%
324 \def\XINT_ftcc_Ep #1\Z #2%
325 {%
326   \expandafter\XINT_ftcc_loop_a\expandafter
327   {\romannumeral0\xintdiv {1[0]}\{#1}\{#2+1/}%
328 }%
329 \def\XINT_ftcc_En #1\Z #2%
330 {%
331   \expandafter\XINT_ftcc_loop_a\expandafter
332   {\romannumeral0\xintdiv {1[0]}\{#1}\{#2+-1/}%
333 }%
334 \def\XINT_ftcc_integer #1\Z #2{ #2}%
335 \def\XINT_ftcc_loop_a #1%
336 {%
337   \expandafter\XINT_ftcc_loop_b
338   \romannumeral0\xintrawithzeros {\xintAdd {1/2[0]}\{#1}\Z {#1}%
339 }%
340 \def\XINT_ftcc_loop_b #1/#2\Z
341 {%
342   \expandafter\XINT_ftcc_loop_c\expandafter
343   {\romannumeral0\xintiigo {#1}\{#2}%
344 }%
345 \def\XINT_ftcc_loop_c #1#2%
346 {%
347   \expandafter\XINT_ftcc_loop_d
348   \romannumeral0\xintsub {#2}\{#1[0]}\Z {#1}%
349 }%
350 \def\XINT_ftcc_loop_d #1%
351 {%
352   \xint_UDzerominusfork
353   #1-\dummy \XINT_ftcc_end
354   0#1\dummy \XINT_ftcc_loop_N
355   0-\dummy {\XINT_ftcc_loop_P #1}%
356   \krof
357 }%
358 \def\XINT_ftcc_end #1\Z #2#3{ #3#2}%
359 \def\XINT_ftcc_loop_P #1\Z #2#3%
360 {%
361   \expandafter\XINT_ftcc_loop_a\expandafter
362   {\romannumeral0\xintdiv {1[0]}\{#1}\{#3#2+1/}%
363 }%
364 \def\XINT_ftcc_loop_N #1\Z #2#3%
365 {%
366   \expandafter\XINT_ftcc_loop_a\expandafter
367   {\romannumeral0\xintdiv {1[0]}\{#1}\{#3#2+-1/}%
368 }%

```

39.12 \xintFtoCv

```

369 \def\xintFtoCv {\romannumeral0\xintftocv }%

```

```

370 \def\xintftocv #1%
371 {%
372   \xinticstocv {\xintFtoCs {#1}}%
373 }%

```

39.13 \xintFtoCCv

```

374 \def\xintFtoCCv {\romannumeral0\xintftoccv }%
375 \def\xintftoccv #1%
376 {%
377   \xintigctocv {\xintFtoCC {#1}}%
378 }%

```

39.14 \xintCstoF

```

379 \def\xintCstoF {\romannumeral0\xintcstof }%
380 \def\xintcstof #1%
381 {%
382   \expandafter\XINT_cstf_prep \romannumeral-‘0#1,\W,%
383 }%
384 \def\XINT_cstf_prep
385 {%
386   \XINT_cstf_loop_a 1001%
387 }%
388 \def\XINT_cstf_loop_a #1#2#3#4#5,%
389 {%
390   \xint_gob_til_W #5\XINT_cstf_end\W
391   \expandafter\XINT_cstf_loop_b
392   \romannumeral0\xintraawithzeros {#5}#{#1}#{#2}#{#3}#{#4}%
393 }%
394 \def\XINT_cstf_loop_b #1/#2.#3#4#5#6%
395 {%
396   \expandafter\XINT_cstf_loop_c\expandafter
397   {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
398   {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
399   {\romannumeral0\xintiiadd {\XINT_Mul {#2}#{#6}}{\XINT_Mul {#1}#{#4}}}%
400   {\romannumeral0\xintiiadd {\XINT_Mul {#2}#{#5}}{\XINT_Mul {#1}#{#3}}}%
401 }%
402 \def\XINT_cstf_loop_c #1#2%
403 {%
404   \expandafter\XINT_cstf_loop_d\expandafter {\expandafter{#2}#{1}}%
405 }%
406 \def\XINT_cstf_loop_d #1#2%
407 {%
408   \expandafter\XINT_cstf_loop_e\expandafter {\expandafter{#2}#1}%
409 }%
410 \def\XINT_cstf_loop_e #1#2%
411 {%
412   \expandafter\XINT_cstf_loop_a\expandafter{#2}#1%
413 }%
414 \def\XINT_cstf_end #1.#2#3#4#5{\xintraawithzeros {#2/#3}}% 1.09b removes [0]

```

39.15 \xintiCstoF

```

415 \def\xintiCstoF {\romannumeral0\xinticstof }%
416 \def\xinticstof #1%
417 {%
418   \expandafter\XINT_icstf_prep \romannumeral-‘0#1,\W,%
419 }%
420 \def\XINT_icstf_prep
421 {%
422   \XINT_icstf_loop_a 1001%
423 }%
424 \def\XINT_icstf_loop_a #1#2#3#4#5,%
425 {%
426   \xint_gob_til_W #5\XINT_icstf_end\W
427   \expandafter
428   \XINT_icstf_loop_b \romannumeral-‘0#5.{#1}{#2}{#3}{#4}%
429 }%
430 \def\XINT_icstf_loop_b #1.#2#3#4#5%
431 {%
432   \expandafter\XINT_icstf_loop_c\expandafter
433   {\romannumeral0\xintiiadd {#5}{\XINT_Mul {#1}{#3}}}%
434   {\romannumeral0\xintiiadd {#4}{\XINT_Mul {#1}{#2}}}%
435   {#2}{#3}%
436 }%
437 \def\XINT_icstf_loop_c #1#2%
438 {%
439   \expandafter\XINT_icstf_loop_a\expandafter {#2}{#1}%
440 }%
441 \def\XINT_icstf_end#1.#2#3#4#5{\xintrawwithzeros {#2/#3}}% 1.09b removes [0]

```

39.16 \xintGctoF

```

442 \def\xintGctoF {\romannumeral0\xintgctof }%
443 \def\xintgctof #1%
444 {%
445   \expandafter\XINT_gctf_prep \romannumeral-‘0#1+\W/%
446 }%
447 \def\XINT_gctf_prep
448 {%
449   \XINT_gctf_loop_a 1001%
450 }%
451 \def\XINT_gctf_loop_a #1#2#3#4#5+%
452 {%
453   \expandafter\XINT_gctf_loop_b
454   \romannumeral0\xintrawwithzeros {#5}.{#1}{#2}{#3}{#4}%
455 }%
456 \def\XINT_gctf_loop_b #1/#2.#3#4#5#6%
457 {%
458   \expandafter\XINT_gctf_loop_c\expandafter
459   {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%

```

```

460 {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
461 {\romannumeral0\xintiiadd {\XINT_Mul {#2}{#6}}{\XINT_Mul {#1}{#4}}}%
462 {\romannumeral0\xintiiadd {\XINT_Mul {#2}{#5}}{\XINT_Mul {#1}{#3}}}%
463 }%
464 \def\XINT_gctf_loop_c #1#2%
465 {%
466 \expandafter\XINT_gctf_loop_d\expandafter {\expandafter{#2}{#1}}%
467 }%
468 \def\XINT_gctf_loop_d #1#2%
469 {%
470 \expandafter\XINT_gctf_loop_e\expandafter {\expandafter{#2}{#1}}%
471 }%
472 \def\XINT_gctf_loop_e #1#2%
473 {%
474 \expandafter\XINT_gctf_loop_f\expandafter {\expandafter{#2}{#1}}%
475 }%
476 \def\XINT_gctf_loop_f #1#2/%
477 {%
478 \xint_gob_til_W #2\XINT_gctf_end\W
479 \expandafter\XINT_gctf_loop_g
480 \romannumeral0\xintraewithzeros {#2}.#1%
481 }%
482 \def\XINT_gctf_loop_g #1/#2.#3#4#5#6%
483 {%
484 \expandafter\XINT_gctf_loop_h\expandafter
485 {\romannumeral0\XINT_mul_fork #1\Z #6\Z }%
486 {\romannumeral0\XINT_mul_fork #1\Z #5\Z }%
487 {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
488 {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
489 }%
490 \def\XINT_gctf_loop_h #1#2%
491 {%
492 \expandafter\XINT_gctf_loop_i\expandafter {\expandafter{#2}{#1}}%
493 }%
494 \def\XINT_gctf_loop_i #1#2%
495 {%
496 \expandafter\XINT_gctf_loop_j\expandafter {\expandafter{#2}{#1}}%
497 }%
498 \def\XINT_gctf_loop_j #1#2%
499 {%
500 \expandafter\XINT_gctf_loop_a\expandafter {#2}{#1}%
501 }%
502 \def\XINT_gctf_end #1.#2#3#4#5{\xintraewithzeros {#2/#3}}% 1.09b removes [0]

```

39.17 \xintiGctoF

```

503 \def\xintiGctoF {\romannumeral0\xintigctof }%
504 \def\xintigctof #1%
505 {%
506 \expandafter\XINT_igctf_prep \romannumeral-‘0#1+\W/%

```

```

507 }%
508 \def\XINT_igctf_prep
509 {%
510   \XINT_igctf_loop_a 1001%
511 }%
512 \def\XINT_igctf_loop_a #1#2#3#4#5+%
513 {%
514   \expandafter\XINT_igctf_loop_b
515   \romannumeral-'0#5.{#1}{#2}{#3}{#4}%
516 }%
517 \def\XINT_igctf_loop_b #1.#2#3#4#5%
518 {%
519   \expandafter\XINT_igctf_loop_c\expandafter
520   {\romannumeral0\xintiiadd {#5}{\XINT_Mul {#1}{#3}}}%
521   {\romannumeral0\xintiiadd {#4}{\XINT_Mul {#1}{#2}}}%
522   {#2}{#3}%
523 }%
524 \def\XINT_igctf_loop_c #1#2%
525 {%
526   \expandafter\XINT_igctf_loop_f\expandafter {\expandafter{#2}{#1}}%
527 }%
528 \def\XINT_igctf_loop_f #1#2#3#4/%
529 {%
530   \xint_gob_til_W #4\XINT_igctf_end\W
531   \expandafter\XINT_igctf_loop_g
532   \romannumeral-'0#4.{#2}{#3}#1%
533 }%
534 \def\XINT_igctf_loop_g #1.#2#3%
535 {%
536   \expandafter\XINT_igctf_loop_h\expandafter
537   {\romannumeral0\XINT_mul_fork #1\Z #3\Z }%
538   {\romannumeral0\XINT_mul_fork #1\Z #2\Z }%
539 }%
540 \def\XINT_igctf_loop_h #1#2%
541 {%
542   \expandafter\XINT_igctf_loop_i\expandafter {#2}{#1}%
543 }%
544 \def\XINT_igctf_loop_i #1#2#3#4%
545 {%
546   \XINT_igctf_loop_a {#3}{#4}{#1}{#2}%
547 }%
548 \def\XINT_igctf_end #1.#2#3#4#5{\xintrawithzeros {#4/#5}}% 1.09b removes [0]

```

39.18 \xintCstoCv

```

549 \def\xintCstoCv {\romannumeral0\xintcstocv }%
550 \def\xintcstocv #1%
551 {%
552   \expandafter\XINT_cstcv_prep \romannumeral-'0#1,\W,%
553 }%

```

```

554 \def\XINT_cstcv_prep
555 {%
556   \XINT_cstcv_loop_a {}1001%
557 }%
558 \def\XINT_cstcv_loop_a #1#2#3#4#5#6,%
559 {%
560   \xint_gob_til_W #6\XINT_cstcv_end\W
561   \expandafter\XINT_cstcv_loop_b
562   \romannumeral0\xintrawithzeros {#6}#{#2}#{#3}#{#4}#{#5}#{#1}%
563 }%
564 \def\XINT_cstcv_loop_b #1/#2.#3#4#5#6%
565 {%
566   \expandafter\XINT_cstcv_loop_c\expandafter
567   {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
568   {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
569   {\romannumeral0\xintiiadd {\XINT_Mul {#2}#{#6}}{\XINT_Mul {#1}#{#4}}}%
570   {\romannumeral0\xintiiadd {\XINT_Mul {#2}#{#5}}{\XINT_Mul {#1}#{#3}}}%
571 }%
572 \def\XINT_cstcv_loop_c #1#2%
573 {%
574   \expandafter\XINT_cstcv_loop_d\expandafter {\expandafter{#2}#{#1}}%
575 }%
576 \def\XINT_cstcv_loop_d #1#2%
577 {%
578   \expandafter\XINT_cstcv_loop_e\expandafter {\expandafter{#2}#1}%
579 }%
580 \def\XINT_cstcv_loop_e #1#2%
581 {%
582   \expandafter\XINT_cstcv_loop_f\expandafter{#2}#1%
583 }%
584 \def\XINT_cstcv_loop_f #1#2#3#4#5%
585 {%
586   \expandafter\XINT_cstcv_loop_g\expandafter
587   {\romannumeral0\xintrawithzeros {#1/#2}}{#5}#{#1}#{#2}#{#3}#{#4}%
588 }%
589 \def\XINT_cstcv_loop_g #1#2{\XINT_cstcv_loop_a {#2}#{#1}}% 1.09b removes [0]
590 \def\XINT_cstcv_end #1.#2#3#4#5#6{ #6}%

```

39.19 \xintiCstoCv

```

591 \def\xintiCstoCv {\romannumeral0\xinticstocv }%
592 \def\xinticstocv #1%
593 {%
594   \expandafter\XINT_icstcv_prep \romannumeral-‘0#1,\W,%
595 }%
596 \def\XINT_icstcv_prep
597 {%
598   \XINT_icstcv_loop_a {}1001%
599 }%
600 \def\XINT_icstcv_loop_a #1#2#3#4#5#6,%

```

```

601 {%
602   \xint_gob_til_W #6\XINT_icstcv_end\W
603   \expandafter
604   \XINT_icstcv_loop_b \romannumeral-‘0#6.{#2}{#3}{#4}{#5}{#1}%
605 }%
606 \def\XINT_icstcv_loop_b #1.#2#3#4#5%
607 {%
608   \expandafter\XINT_icstcv_loop_c\expandafter
609   {\romannumeral0\xintiiadd {#5}{\XINT_Mul {#1}{#3}}}%
610   {\romannumeral0\xintiiadd {#4}{\XINT_Mul {#1}{#2}}}%
611   {{#2}{#3}}}%
612 }%
613 \def\XINT_icstcv_loop_c #1#2%
614 {%
615   \expandafter\XINT_icstcv_loop_d\expandafter {#2}{#1}%
616 }%
617 \def\XINT_icstcv_loop_d #1#2%
618 {%
619   \expandafter\XINT_icstcv_loop_e\expandafter
620   {\romannumeral0\xintrawithzeros {#1/#2}}{#1}{#2}}%
621 }%
622 \def\XINT_icstcv_loop_e #1#2#3#4{\XINT_icstcv_loop_a {#4}{#1}}#2#3}%
623 \def\XINT_icstcv_end #1.#2#3#4#5#6{ #6}% 1.09b removes [0]

```

39.20 \xintGctoCv

```

624 \def\xintGctoCv {\romannumeral0\xintgctocv }%
625 \def\xintgctocv #1%
626 {%
627   \expandafter\XINT_gctcv_prep \romannumeral-‘0#1+\W/%
628 }%
629 \def\XINT_gctcv_prep
630 {%
631   \XINT_gctcv_loop_a {}1001%
632 }%
633 \def\XINT_gctcv_loop_a #1#2#3#4#5#6+%
634 {%
635   \expandafter\XINT_gctcv_loop_b
636   \romannumeral0\xintrawithzeros {#6}.{#2}{#3}{#4}{#5}{#1}%
637 }%
638 \def\XINT_gctcv_loop_b #1/#2.#3#4#5#6%
639 {%
640   \expandafter\XINT_gctcv_loop_c\expandafter
641   {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
642   {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
643   {\romannumeral0\xintiiadd {\XINT_Mul {#2}{#6}}{\XINT_Mul {#1}{#4}}}%
644   {\romannumeral0\xintiiadd {\XINT_Mul {#2}{#5}}{\XINT_Mul {#1}{#3}}}%
645 }%
646 \def\XINT_gctcv_loop_c #1#2%
647 {%

```



```

648 \expandafter\XINT_gctcv_loop_d\expandafter {\expandafter{#2}{#1}}%
649 }%
650 \def\XINT_gctcv_loop_d #1#2%
651 {%
652 \expandafter\XINT_gctcv_loop_e\expandafter {\expandafter{#2}{#1}}%
653 }%
654 \def\XINT_gctcv_loop_e #1#2%
655 {%
656 \expandafter\XINT_gctcv_loop_f\expandafter {#2}#1%
657 }%
658 \def\XINT_gctcv_loop_f #1#2%
659 {%
660 \expandafter\XINT_gctcv_loop_g\expandafter
661 {\romannumeral0\xintraewithzeros {#1/#2}}{#1}{#2}}%
662 }%
663 \def\XINT_gctcv_loop_g #1#2#3#4%
664 {%
665 \XINT_gctcv_loop_h {#4{#1}}{#2#3}% 1.09b removes [0]
666 }%
667 \def\XINT_gctcv_loop_h #1#2#3/%
668 {%
669 \xint_gob_til_W #3\XINT_gctcv_end\W
670 \expandafter\XINT_gctcv_loop_i
671 \romannumeral0\xintraewithzeros {#3}.#2{#1}%
672 }%
673 \def\XINT_gctcv_loop_i #1/#2.#3#4#5#6%
674 {%
675 \expandafter\XINT_gctcv_loop_j\expandafter
676 {\romannumeral0\XINT_mul_fork #1\Z #6\Z }%
677 {\romannumeral0\XINT_mul_fork #1\Z #5\Z }%
678 {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
679 {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
680 }%
681 \def\XINT_gctcv_loop_j #1#2%
682 {%
683 \expandafter\XINT_gctcv_loop_k\expandafter {\expandafter{#2}{#1}}%
684 }%
685 \def\XINT_gctcv_loop_k #1#2%
686 {%
687 \expandafter\XINT_gctcv_loop_l\expandafter {\expandafter{#2}#1}%
688 }%
689 \def\XINT_gctcv_loop_l #1#2%
690 {%
691 \expandafter\XINT_gctcv_loop_m\expandafter {\expandafter{#2}#1}%
692 }%
693 \def\XINT_gctcv_loop_m #1#2{\XINT_gctcv_loop_a {#2}#1}%
694 \def\XINT_gctcv_end #1.#2#3#4#5#6{ #6}%

```

39.21 \xintiGctoCv

```

695 \def\xintiGctoCv {\romannumeral0\xintigctocv }%
696 \def\xintigctocv #1%
697 {%
698   \expandafter\xINT_igctcv_prep \romannumeral-‘0#1+\W/%
699 }%
700 \def\xINT_igctcv_prep
701 {%
702   \XINT_igctcv_loop_a {}1001%
703 }%
704 \def\xINT_igctcv_loop_a #1#2#3#4#5#6+%
705 {%
706   \expandafter\xINT_igctcv_loop_b
707   \romannumeral-‘0#6.{#2}{#3}{#4}{#5}{#1}%
708 }%
709 \def\xINT_igctcv_loop_b #1.#2#3#4#5%
710 {%
711   \expandafter\xINT_igctcv_loop_c\expandafter
712   {\romannumeral0\xintiiadd {#5}{\XINT_Mul {#1}{#3}}}%
713   {\romannumeral0\xintiiadd {#4}{\XINT_Mul {#1}{#2}}}%
714   {{#2}{#3}}}%
715 }%
716 \def\xINT_igctcv_loop_c #1#2%
717 {%
718   \expandafter\xINT_igctcv_loop_f\expandafter {\expandafter{#2}{#1}}%
719 }%
720 \def\xINT_igctcv_loop_f #1#2#3#4/%
721 {%
722   \xint_gob_til_W #4\xINT_igctcv_end_a\W
723   \expandafter\xINT_igctcv_loop_g
724   \romannumeral-‘0#4.#1#2{#3}%
725 }%
726 \def\xINT_igctcv_loop_g #1.#2#3#4#5%
727 {%
728   \expandafter\xINT_igctcv_loop_h\expandafter
729   {\romannumeral0\xINT_mul_fork #1\Z #5\Z }%
730   {\romannumeral0\xINT_mul_fork #1\Z #4\Z }%
731   {{#2}{#3}}}%
732 }%
733 \def\xINT_igctcv_loop_h #1#2%
734 {%
735   \expandafter\xINT_igctcv_loop_i\expandafter {\expandafter{#2}{#1}}%
736 }%
737 \def\xINT_igctcv_loop_i #1#2{\xINT_igctcv_loop_k #2{#2#1}}%
738 \def\xINT_igctcv_loop_k #1#2%
739 {%
740   \expandafter\xINT_igctcv_loop_l\expandafter
741   {\romannumeral0\xintrawwithzeros {#1/#2}}}%
742 }%
743 \def\xINT_igctcv_loop_l #1#2#3{\xINT_igctcv_loop_a {#3{#1[0]}}#2}%

```

```

744 \def\XINT_igctcv_end_a #1.#2#3#4#5%
745 {%
746   \expandafter\XINT_igctcv_end_b\expandafter
747   {\romannumeral0\xintraewithzeros {#2/#3}}%
748 }%
749 \def\XINT_igctcv_end_b #1#2{ #2{#1}}% 1.09b removes [0]

```

39.22 \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.

```

750 \def\xintCntoF {\romannumeral0\xintcntof }%
751 \def\xintcntof #1%
752 {%
753   \expandafter\XINT_cntf\expandafter {\the\numexpr #1}%
754 }%
755 \def\XINT_cntf #1#2%
756 {%
757   \ifnum #1>\xint_c_
758     \xint_afterfi {\expandafter\XINT_cntf_loop\expandafter
759                   {\the\numexpr #1-1\expandafter}\expandafter
760                   {\romannumeral-‘0#2{#1}}{#2}}%
761   \else
762     \xint_afterfi
763       {\ifnum #1=\xint_c_
764         \xint_afterfi {\expandafter\space \romannumeral-‘0#2{0}}%
765         \else \xint_afterfi { 0/1[0]}%
766         \fi}%
767   \fi
768 }%
769 \def\XINT_cntf_loop #1#2#3%
770 {%
771   \ifnum #1>\xint_c_ \else \XINT_cntf_exit \fi
772   \expandafter\XINT_cntf_loop\expandafter
773   {\the\numexpr #1-1\expandafter }\expandafter
774   {\romannumeral0\xintadd {\xintDiv {1[0]}{#2}}{#3{#1}}}%
775   {#3}%
776 }%
777 \def\XINT_cntf_exit \fi
778   \expandafter\XINT_cntf_loop\expandafter
779   #1\expandafter #2#3%
780 {%
781   \fi\xint_gobble_ii #2%
782 }%

```

39.23 \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.

```

783 \def\xintGCntoF {\romannumeral0\xintgcntof }%
784 \def\xintgcntof #1%
785 {%
786   \expandafter\XINT_gcntf\expandafter {\the\numexpr #1}%
787 }%
788 \def\XINT_gcntf #1#2#3%
789 {%
790   \ifnum #1>\xint_c_
791     \xint_afterfi {\expandafter\XINT_gcntf_loop\expandafter
792                   {\the\numexpr #1-1\expandafter}\expandafter
793                   {\romannumeral-‘0#2{#1}}{#2}{#3}}%
794   \else
795     \xint_afterfi
796       {\ifnum #1=\xint_c_
797         \xint_afterfi {\expandafter\space\romannumeral-‘0#2{0}}%
798         \else \xint_afterfi { 0/1[0]}%
799         \fi}%
800   \fi
801 }%
802 \def\XINT_gcntf_loop #1#2#3#4%
803 {%
804   \ifnum #1>\xint_c_ \else \XINT_gcntf_exit \fi
805   \expandafter\XINT_gcntf_loop\expandafter
806   {\the\numexpr #1-1\expandafter }\expandafter
807   {\romannumeral0\xintadd {\xintDiv {#4{#1}}{#2}}{#3{#1}}}%
808   {#3}{#4}%
809 }%
810 \def\XINT_gcntf_exit \fi
811   \expandafter\XINT_gcntf_loop\expandafter
812   #1\expandafter #2#3#4%
813 {%
814   \fi\xint_gobble_ii #2%
815 }%

```

39.24 \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.

```

816 \def\xintCntoCs {\romannumeral0\xintcntocs }%
817 \def\xintcntocs #1%
818 {%
819   \expandafter\XINT_cntcs\expandafter {\the\numexpr #1}%
820 }%
821 \def\XINT_cntcs #1#2%
822 {%
823   \ifnum #1<0
824     \xint_afterfi { 0/1[0]}%
825   \else

```

```

826     \xint_afterfi {\expandafter\XINT_cntcs_loop\expandafter
827                   {\the\numexpr #1-1\expandafter}\expandafter
828                   {\expandafter{\romannumeral-'\0#2{#1}}{#2}}}%
829   \fi
830 }%
831 \def\XINT_cntcs_loop #1#2#3%
832 {%
833   \ifnum #1>-1 \else \XINT_cntcs_exit \fi
834   \expandafter\XINT_cntcs_loop\expandafter
835   {\the\numexpr #1-1\expandafter }\expandafter
836   {\expandafter{\romannumeral-'\0#3{#1}},#2}{#3}%
837 }%
838 \def\XINT_cntcs_exit \fi
839   \expandafter\XINT_cntcs_loop\expandafter
840   #1\expandafter #2#3%
841 {%
842   \fi\XINT_cntcs_exit_b #2%
843 }%
844 \def\XINT_cntcs_exit_b #1,{ }%

```

39.25 \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.

```

845 \def\xintCntoGC {\romannumeral0\xintcntogc }%
846 \def\xintcntogc #1%
847 {%
848   \expandafter\XINT_cntgc\expandafter {\the\numexpr #1}%
849 }%
850 \def\XINT_cntgc #1#2%
851 {%
852   \ifnum #1<0
853     \xint_afterfi { 0/1[0]}%
854   \else
855     \xint_afterfi {\expandafter\XINT_cntgc_loop\expandafter
856                   {\the\numexpr #1-1\expandafter}\expandafter
857                   {\expandafter{\romannumeral-'\0#2{#1}}{#2}}}%
858   \fi
859 }%
860 \def\XINT_cntgc_loop #1#2#3%
861 {%
862   \ifnum #1>-1 \else \XINT_cntgc_exit \fi
863   \expandafter\XINT_cntgc_loop\expandafter
864   {\the\numexpr #1-1\expandafter }\expandafter
865   {\expandafter{\romannumeral-'\0#3{#1}}+1/#2}{#3}%
866 }%
867 \def\XINT_cntgc_exit \fi
868   \expandafter\XINT_cntgc_loop\expandafter

```

```

869 #1\expandafter #2#3%
870 {%
871   \fi\XINT_cntgc_exit_b #2%
872 }%
873 \def\XINT_cntgc_exit_b #1+1/{ }%

```

39.26 \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.

```

874 \def\xintGCntoGC {\romannumeral0\xintgcntogc }%
875 \def\xintgcntogc #1%
876 {%
877   \expandafter\XINT_gcntgc\expandafter {\the\numexpr #1}%
878 }%
879 \def\XINT_gcntgc #1#2#3%
880 {%
881   \ifnum #1<0
882     \xint_afterfi { {0/1[0]}}%
883   \else
884     \xint_afterfi {\expandafter\XINT_gcntgc_loop\expandafter
885                   {\the\numexpr #1-1\expandafter}\expandafter
886                   {\expandafter{\romannumeral-'0#2{#1}}{#2}{#3}}}%
887   \fi
888 }%
889 \def\XINT_gcntgc_loop #1#2#3#4%
890 {%
891   \ifnum #1>-1 \else \XINT_gcntgc_exit \fi
892   \expandafter\XINT_gcntgc_loop_b\expandafter
893   {\expandafter{\romannumeral-'0#4{#1}}/#2}{#3{#1}}{#1}{#3}{#4}%
894 }%
895 \def\XINT_gcntgc_loop_b #1#2#3%
896 {%
897   \expandafter\XINT_gcntgc_loop\expandafter
898   {\the\numexpr #3-1\expandafter}\expandafter
899   {\expandafter{\romannumeral-'0#2}+#1}%
900 }%
901 \def\XINT_gcntgc_exit \fi
902   \expandafter\XINT_gcntgc_loop_b\expandafter #1#2#3#4#5%
903 }%
904   \fi\XINT_gcntgc_exit_b #1%
905 }%
906 \def\XINT_gcntgc_exit_b #1/{ }%

```

39.27 \xintCstoGC

```

907 \def\xintCstoGC {\romannumeral0\xintcstogc }%
908 \def\xintcstogc #1%

```

```

909 {%
910   \expandafter\XINT_cstc_prep \romannumeral-‘0#1,\W,%
911 }%
912 \def\XINT_cstc_prep #1,{\XINT_cstc_loop_a {{#1}}}%
913 \def\XINT_cstc_loop_a #1#2,%
914 {%
915   \xint_gob_til_W #2\XINT_cstc_end\W
916   \XINT_cstc_loop_b {#1}{#2}%
917 }%
918 \def\XINT_cstc_loop_b #1#2{\XINT_cstc_loop_a {#1+1/{#2}}}%
919 \def\XINT_cstc_end\W\XINT_cstc_loop_b #1#2{ #1}%

```

39.28 \xintGctoGC

```

920 \def\xintGctoGC {\romannumeral0\xintgctogc }%
921 \def\xintgctogc #1%
922 {%
923   \expandafter\XINT_gctgc_start \romannumeral-‘0#1+\W/%
924 }%
925 \def\XINT_gctgc_start {\XINT_gctgc_loop_a {}}%
926 \def\XINT_gctgc_loop_a #1#2+#3/%
927 {%
928   \xint_gob_til_W #3\XINT_gctgc_end\W
929   \expandafter\XINT_gctgc_loop_b\expandafter
930   {\romannumeral-‘0#2}{#3}{#1}%
931 }%
932 \def\XINT_gctgc_loop_b #1#2%
933 {%
934   \expandafter\XINT_gctgc_loop_c\expandafter
935   {\romannumeral-‘0#2}{#1}%
936 }%
937 \def\XINT_gctgc_loop_c #1#2#3%
938 {%
939   \XINT_gctgc_loop_a {#3{#2}+{#1}}/%
940 }%
941 \def\XINT_gctgc_end\W\expandafter\XINT_gctgc_loop_b
942 {%
943   \expandafter\XINT_gctgc_end_b
944 }%
945 \def\XINT_gctgc_end_b #1#2#3{ #3{#1}}%
946 \XINT_restorecatcodes_endinput%

```

40 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

.1	Catcodes, ε -TeX and reload detection ..	359		
.2	Confirmation of <i>xintfrac</i> loading ...	360	.12	Parentheses
.3	Catcodes	361	.13	The \XINT_expr_until_<op> macros
.4	Package identification	361		for boolean operators, comparison op-
.5	Helper macros	361		erators, arithmetic operators, scientific
.6	Encapsulation in pseudo names	361		notation.
.7	\xintifboolexpr, \xintifboolfloatexpr	361	.14	The comma as binary operator
.8	\xintexpr, \xinttheexpr, \xintthe ..	361	.15	\XINT_expr_op_-<level>: minus as
.9	\XINT_get_next: looking for a number	362		prefix inherits its precedence level
.10	\XINT_expr_scan_dec_or_func: col-		.16	? as two-way conditional
	lecting an integer or decimal number		.17	: as three-way conditional
	or function name	364	.18	! as postfix factorial operator
.11	\XINT_expr_getop: looking for an		.19	Functions
			.20	\xintNewExpr, \xintNewFloatExpr...
				378

40.1 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.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2 \catcode13=5      % ^^M
3 \endlinechar=13 %
4 \catcode123=1     % {
5 \catcode125=2     % }
6 \catcode64=11     % @
7 \catcode35=6      % #
8 \catcode44=12     % ,
9 \catcode45=12     % -
10 \catcode46=12    % .
11 \catcode58=12    % :
12 \def\space { }%
13 \let\z\endgroup
14 \expandafter\let\expandafter\x\csname ver@xintexpr.sty\endcsname
15 \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
16 \expandafter
17 \ifx\csname PackageInfo\endcsname\relax
18 \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
19 \else
20 \def\y#1#2{\PackageInfo{#1}{#2}}%
21 \fi

```

```

22 \expandafter
23 \ifx\csname numexpr\endcsname\relax
24   \y{xintexpr}{\numexpr not available, aborting input}%
25   \aftergroup\endinput
26 \else
27   \ifx\x\relax % plain-TeX, first loading of xintexpr.sty
28     \ifx\w\relax % but xintfrac.sty not yet loaded.
29       \y{xintexpr}{Package xintfrac is required}%
30       \y{xintexpr}{Will try \string\input\space xintfrac.sty}%
31       \def\z{\endgroup\input xintfrac.sty\relax}%
32     \fi
33   \else
34     \def\empty {}%
35     \ifx\x\empty % LaTeX, first loading,
36     % variable is initialized, but \ProvidesPackage not yet seen
37     \ifx\w\relax % xintfrac.sty not yet loaded.
38       \y{xintexpr}{Package xintfrac is required}%
39       \y{xintexpr}{Will try \string\RequirePackage{xintfrac}}%
40       \def\z{\endgroup\RequirePackage{xintfrac}}%
41     \fi
42   \else
43     \y{xintexpr}{I was already loaded, aborting input}%
44     \aftergroup\endinput
45   \fi
46 \fi
47 \fi
48 \z%

```

40.2 Confirmation of *xintfrac* loading

```

49 \begingroup\catcode61\catcode48\catcode32=10\relax%
50 \catcode13=5 % ^^M
51 \endlinechar=13 %
52 \catcode123=1 % {
53 \catcode125=2 % }
54 \catcode64=11 % @
55 \catcode35=6 % #
56 \catcode44=12 % ,
57 \catcode45=12 % -
58 \catcode46=12 % .
59 \catcode58=12 % :
60 \ifdefined\PackageInfo
61   \def\y#1#2{\PackageInfo{#1}{#2}}%
62   \else
63     \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
64   \fi
65 \def\empty {}%
66 \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
67 \ifx\w\relax % Plain TeX, user gave a file name at the prompt

```

```

68     \y{xintexpr}{Loading of package xintfrac failed, aborting input}%
69     \aftergroup\endinput
70 \fi
71 \ifx\w\empty % LaTeX, user gave a file name at the prompt
72     \y{xintexpr}{Loading of package xintfrac failed, aborting input}%
73     \aftergroup\endinput
74 \fi
75 \endgroup%

```

40.3 Catcodes

```
76 \XINTsetupcatcodes%
```

40.4 Package identification

```

77 \XINT_providespackage
78 \ProvidesPackage{xintexpr}%
79 [2013/11/28 v1.09h Expandable expression parser (jfb)]%

```

40.5 Helper macros

```

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

```

40.6 Encapsulation in pseudo names

```

85 \def\XINT_expr_lock #1!{\expandafter\space\csname .#1\endcsname }%
86 \def\XINT_expr_unlock {\expandafter\xint_gob_til_dot\string }%
87 \def\XINT_expr_usethe {use_xintthe!\xintError:use_xintthe! }%
88 \def\XINT_expr_done {\XINT_expr_usethe\XINT_expr_print }%
89 \def\XINT_expr_print #1{\XINT_expr_unlock #1}%
90 \def\XINT_flexpr_done {\XINT_expr_usethe\XINT_flexpr_print }%
91 \def\XINT_flexpr_print #1{\xintFloat:csv{\XINT_expr_unlock #1}}%
92 \def\XINT_numexpr_print #1{\xintRound:csv{\XINT_expr_unlock #1}}%
93 \def\XINT_boolexpr_print #1{\xintIsTrue:csv{\XINT_expr_unlock #1}}%

```

40.7 \xintifboolexpr, \xintifboolfloatexpr

1.09c. Not to be used on comma separated expressions. I could perhaps use \xint0-Rof:csv (or AND, or XOR) to allow it?

```

94 \def\xintifboolexpr #1{\romannumeral0\xintifnotzero {\xinttheexpr #1\relax}}%
95 \def\xintifboolfloatexpr #1{\romannumeral0\xintifnotzero
96     {\xintthefloatexpr #1\relax}}%

```

40.8 \xintexpr, \xinttheexpr, \xintthe

```

97 \def\xintexpr {\romannumeral0\xinteval }%
98 \def\xinteval

```

```

99 {%
100   \expandafter\XINT_expr_until_end_a \romannumeral-‘0\XINT_expr_getnext
101 }%
102 \def\xinttheeval {\expandafter\xint_gobble_ii\romannumeral0\xinteval }%
103 \def\xinttheexpr {\romannumeral-‘0\xinttheeval }%
104 \def\XINT_numexpr_post !\XINT_expr_usethe\XINT_expr_print%
105       { !\XINT_expr_usethe\XINT_numexpr_print }%
106 \def\xintnumexpr {\romannumeral0\expandafter\XINT_numexpr_post
107       \romannumeral0\xinteval }%
108 \def\xintthenumexpr {\romannumeral-‘0\xintthe\xintnumexpr }%
109 \def\XINT_boolexpr_post !\XINT_expr_usethe\XINT_expr_print%
110       { !\XINT_expr_usethe\XINT_boolexpr_print }%
111 \def\xintboolexpr {\romannumeral0\expandafter\XINT_boolexpr_post
112       \romannumeral0\xinteval }%
113 \def\xinttheboolexpr {\romannumeral-‘0\xintthe\xintboolexpr }%
114 \def\xintfloatexpr {\romannumeral0\xintfloateval }%
115 \def\xintfloateval
116 {%
117   \expandafter\XINT_flexpr_until_end_a \romannumeral-‘0\XINT_expr_getnext
118 }%
119 \def\xintthefloatexpr {\romannumeral-‘0\xintthe\xintfloatexpr }%
120 \def\xintthe #1{\romannumeral-‘0\expandafter\xint_gobble_ii\romannumeral-‘0#1}%

```

40.9 \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.

```

121 \def\xINT_expr_getnext
122 {%
123   \expandafter\xINT_expr_getnext_checkforbraced_a
124   \romannumeral-'0\romannumeral-'0%
125 }%
126 \def\xINT_expr_getnext_checkforbraced_a #1%
127 {%
128   \XINT_expr_getnext_checkforbraced_b #1\W\Z {#1}%
129 }%
130 \def\xINT_expr_getnext_checkforbraced_b #1#2%
131 {%
132   \xint_UDwfork
133     #1\dummy \XINT_expr_getnext_emptybracepair
134     #2\dummy \XINT_expr_getnext_onetoken_perhaps
135     \W\dummy \XINT_expr_getnext_gotbracedstuff
136   \krof
137 }%
138 \def\xINT_expr_getnext_onetoken_perhaps\Z #1%
139 {%
140   \expandafter\xINT_expr_getnext_checkforbraced_c\expandafter
141   {\romannumeral-'0#1}%
142 }%
143 \def\xINT_expr_getnext_checkforbraced_c #1%
144 {%
145   \XINT_expr_getnext_checkforbraced_d #1\W\Z {#1}%
146 }%
147 \def\xINT_expr_getnext_checkforbraced_d #1#2%
148 {%
149   \xint_UDwfork
150     #1\dummy \XINT_expr_getnext_emptybracepair
151     #2\dummy \XINT_expr_getnext_onetoken_wehope
152     \W\dummy \XINT_expr_getnext_gotbracedstuff
153   \krof
154 }% doubly braced things are not acceptable, will cause errors.
155 \def\xINT_expr_getnext_emptybracepair #1{\XINT_expr_getnext }%
156 \def\xINT_expr_getnext_gotbracedstuff #1\W\Z #2% {...} -> number/fraction
157 {%
158   \expandafter\xINT_expr_getop\csname .#2\endcsname
159 }%
160 \def\xINT_expr_getnext_onetoken_wehope\Z #1% #1 isn't a control sequence!
161 {%
162   \xint_gob_til! #1\XINT_expr_subexpr !%
163   \expandafter\xINT_expr_getnext_onetoken_fork\string #1%
164 }% after this #1 should be now a catcode 12 token.
165 \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 responsibility 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 `#`.

```

166 \begingroup
167 \lccode'*= '#
168 \lowercase{\endgroup
169 \def\xINT_expr_sixwayfork #1(-.**\dummy #2#3\krof {#2}%
170 \def\xINT_expr_getnext_onetoken_fork #1%
171 {% The * is in truth catcode 12 #. For (clever!) use with \xintNewExpr.
172   \xINT_expr_sixwayfork
173     #1-.**\dummy {\xint_c_xviii ({}}% back to until to trigger oparen
174     (#1-.**\dummy   -%
175     (-#1+*\dummy {\xINT_expr_scandec_II.}%
176     (-.#1*\dummy  \xINT_expr_getnext%
177     (-.+*#1\dummy {\xINT_expr_scandec_II}%
178     (-.**\dummy {\xINT_expr_scan_dec_or_func #1}%
179   \krof
180 }}}%

```

40.10 `\XINT_expr_scan_dec_or_func`: collecting an integer or decimal number or function name

```

181 \def\xINT_expr_scan_dec_or_func #1% this #1 of catcode 12
182 {%
183   \ifnum \xint_c_ix<1#1
184     \expandafter\xINT_expr_scandec_I
185   \else % We assume we are dealing with a function name!!
186     \expandafter\xINT_expr_scanfunc
187   \fi #1%
188 }%
189 \def\xINT_expr_scanfunc
190 {%
191   \expandafter\xINT_expr_func\romannumeral-'0\xINT_expr_scanfunc_c
192 }%
193 \def\xINT_expr_scanfunc_c #1%
194 {%
195   \expandafter #1\romannumeral-'0\expandafter
196   \XINT_expr_scanfunc_a\romannumeral-'0\romannumeral-'0%
197 }%
198 \def\xINT_expr_scanfunc_a #1% please no braced things here!
199 {%
200   \ifcat #1\relax % missing opening parenthesis, probably
201     \expandafter\xINT_expr_scanfunc_panic
202   \else
203     \xint_afterfi{\expandafter\xINT_expr_scanfunc_b \string #1}%
204   \fi

```

```

205 }%
206 \def\XINT_expr_scanfunc_b #1%
207 {%
208   \if #1(\expandafter \xint_gobble_iii\fi
209   \xint_firstofone
210   {% added in 1.09c for bool and togl
211   \if #1)\expandafter \xint_gobble_i
212   \else \expandafter \xint_firstoftwo
213   \fi }%
214   {\XINT_expr_scanfunc_c #1}{%
215 }%
216 \def\XINT_expr_scanfunc_panic {\xintError:bigtroubleahead(0\relax }%
217 \def\XINT_expr_func #1(% common to expr and flexpr
218 {%
219   \xint_c_xviii @{#1}% functions have the highest priority.
220 }%

```

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

```

221 \def\XINT_expr_scandec_I
222 {%
223   \expandafter\XINT_expr_getop\romannumeral-'0\expandafter
224   \XINT_expr_lock\romannumeral-'0\XINT_expr_scanintpart_b
225 }%
226 \def\XINT_expr_scandec_II
227 {%
228   \expandafter\XINT_expr_getop\romannumeral-'0\expandafter
229   \XINT_expr_lock\romannumeral-'0\XINT_expr_scanfracpart_b
230 }%
231 \def\XINT_expr_scanintpart_a #1%
232 {%
233   \ifnum \xint_c_ix<1\string#1
234     \expandafter\expandafter\expandafter\XINT_expr_scanintpart_b
235     \expandafter\string
236   \else
237     \if #1.%
238       \expandafter\expandafter\expandafter
239       \XINT_expr_scandec_transition
240     \else
241       \expandafter\expandafter\expandafter !% ! of catcode 11...
242     \fi
243   \fi
244   #1%
245 }%
246 \def\XINT_expr_scanintpart_b #1%
247 {%
248   \expandafter #1\romannumeral-'0\expandafter
249   \XINT_expr_scanintpart_a\romannumeral-'0\romannumeral-'0%
250 }%
251 \def\XINT_expr_scandec_transition #1%

```

```

252 {%
253   \expandafter.\romannumeral-'0\expandafter
254   \XINT_expr_scanfracpart_a\romannumeral-'0\romannumeral-'0%
255 }%
256 \def\XINT_expr_scanfracpart_a #1%
257 {%
258   \ifnum \xint_c_ix<1\string#1
259     \expandafter\expandafter\expandafter\XINT_expr_scanfracpart_b
260     \expandafter\string
261   \else
262     \expandafter !%
263   \fi
264   #1%
265 }%
266 \def\XINT_expr_scanfracpart_b #1%
267 {%
268   \expandafter #1\romannumeral-'0\expandafter
269   \XINT_expr_scanfracpart_a\romannumeral-'0\romannumeral-'0%
270 }%

```

40.11 \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.

```

271 \def\XINT_expr_getop #1% this #1 is the current locked computed value
272 {% full expansion of next token, first swallowing a possible space
273   \expandafter\XINT_expr_getop_a\expandafter #1%
274   \romannumeral-'0\romannumeral-'0%
275 }%
276 \def\XINT_expr_getop_a #1#2%
277 {% if an un-expandable control sequence is found, must be the ending \relax
278   \ifcat #2\relax
279     \ifx #2\relax
280       \expandafter\expandafter\expandafter
281       \XINT_expr_founded
282     \else
283       \XINT_expr_unexpectedtoken
284       \expandafter\expandafter\expandafter

```



```

285         \XINT_expr_getop
286     \fi
287 \else
288     \expandafter\XINT_expr_foundop\expandafter #2%
289 \fi
290 #1%
291 }%
292 \def\XINT_expr_foundend {\xint_c_ \relax }% \relax is a place holder here.
293 \def\XINT_expr_foundop #1% then becomes <prec> <op> and is followed by <\.f>
294 {% 1.09a: no control sequence \XINT_expr_op_#1, code common to expr/flexpr
295     \ifcsname XINT_expr_precedence_#1\endcsname
296         \expandafter\xint_afterfi\expandafter
297         {\csname XINT_expr_precedence_#1\endcsname #1}%
298     \else
299         \XINT_expr_unexpectedtoken
300         \expandafter\XINT_expr_getop
301     \fi
302 }%

```

40.12 Parentheses

1.09a removes some doubling of \romannumeral-'\0 from 1.08b which served no useful purpose here (I think...).

```

303 \def\XINT_tmpa #1#2#3#4#5%
304 {%
305     \def#1##1%
306     {%
307         \xint_UDsignfork
308             ##1\dummy {\expandafter#1\romannumeral-'\0#3}%
309             -\dummy  {#2##1}%
310     \krof
311     }%
312 \def#2##1##2%
313 {%
314     \ifcase ##1\expandafter #4%
315     \or \xint_afterfi{%
316         \XINT_expr_extra_closing_paren
317         \expandafter #1\romannumeral-'\0\XINT_expr_getop
318         }%
319     \else \xint_afterfi{%
320         \expandafter#1\romannumeral-'\0\csname XINT_#5_op_##2\endcsname
321         }%
322     \fi
323     }%
324 }%
325 \expandafter\XINT_tmpa
326 \csname XINT_expr_until_end_a\expandafter\endcsname
327 \csname XINT_expr_until_end_b\expandafter\endcsname

```

```

328 \csname XINT_expr_op_-vi\expandafter\endcsname
329 \csname XINT_expr_done\endcsname
330 {expr}%
331 \expandafter\XINT_tmpa
332 \csname XINT_flexpr_until_end_a\expandafter\endcsname
333 \csname XINT_flexpr_until_end_b\expandafter\endcsname
334 \csname XINT_flexpr_op_-vi\expandafter\endcsname
335 \csname XINT_flexpr_done\endcsname
336 {flexpr}%
337 \def\XINT_expr_extra_closing_paren {\xintError:removed }%
338 \def\XINT_tmpa #1#2#3#4#5#6%
339 {%
340 \def #1{\expandafter #3\romannumeral-'0\XINT_expr_getnext }%
341 \let #2#1%
342 \def #3##1{\xint_UDsignfork
343     ##1\dummy {\expandafter #3\romannumeral-'0#5}%
344     -\dummy {#4##1}%
345     \krof }%
346 \def #4##1##2%
347 {%
348 \ifcase ##1\expandafter \XINT_expr_missing_cparen
349 \or \expandafter \XINT_expr_getop
350 \else \xint_afterfi
351 {\expandafter #3\romannumeral-'0\csname XINT_#6_op_##2\endcsname }%
352 \fi
353 }%
354 }%
355 \expandafter\XINT_tmpa
356 \csname XINT_expr_op_(\expandafter\endcsname
357 \csname XINT_expr_oparen\expandafter\endcsname
358 \csname XINT_expr_until_)_a\expandafter\endcsname
359 \csname XINT_expr_until_)_b\expandafter\endcsname
360 \csname XINT_expr_op_-vi\endcsname
361 {expr}%
362 \expandafter\XINT_tmpa
363 \csname XINT_flexpr_op_(\expandafter\endcsname
364 \csname XINT_flexpr_oparen\expandafter\endcsname
365 \csname XINT_flexpr_until_)_a\expandafter\endcsname
366 \csname XINT_flexpr_until_)_b\expandafter\endcsname
367 \csname XINT_flexpr_op_-vi\endcsname
368 {flexpr}%
369 \def\XINT_expr_missing_cparen {\xintError:inserted \xint_c_ \XINT_expr_done }%
370 \expandafter\let\csname XINT_expr_precedence_\endcsname \xint_c_i
371 \expandafter\let\csname XINT_expr_op_\endcsname\XINT_expr_getop
372 \expandafter\let\csname XINT_flexpr_precedence_\endcsname \xint_c_i
373 \expandafter\let\csname XINT_flexpr_op_\endcsname\XINT_expr_getop

```

40.13 The `\XINT_expr_until_<op>` macros for boolean operators, comparison operators, arithmetic operators, scientific notation.

Extended in 1.09a with comparison and boolean operators.

```

374 \def\XINT_tmpb #1#2#3#4#5#6%
375 {%
376   \expandafter\XINT_tmpc
377   \csname XINT_#1_op_#3\expandafter\endcsname
378   \csname XINT_#1_until_#3_a\expandafter\endcsname
379   \csname XINT_#1_until_#3_b\expandafter\endcsname
380   \csname XINT_#1_op_#5\expandafter\endcsname
381   \csname xint_c_#4\expandafter\endcsname
382   \csname #2#6\expandafter\endcsname
383   \csname XINT_expr_precedence_#3\endcsname {#1}%
384 }%
385 \def\XINT_tmpc #1#2#3#4#5#6#7#8%
386 {%
387   \def #1##1% \XINT_expr_op_<op>
388   {% keep value, get next number and operator, then do until
389     \expandafter #2\expandafter ##1%
390     \romannumeral-'0\expandafter\XINT_expr_getnext
391   }%
392   \def #2##1##2% \XINT_expr_until_<op>_a
393   {\xint_UDsignfork
394     ##2\dummy {\expandafter #2\expandafter ##1\romannumeral-'0#4}%
395     -\dummy {#3##1##2}%
396   \krof }%
397   \def #3##1##2##3##4% \XINT_expr_until_<op>_b
398   {% either execute next operation now, or first do next (possibly unary)
399     \ifnum ##2>#5%
400       \xint_afterfi {\expandafter #2\expandafter ##1\romannumeral-'0%
401         \csname XINT_#8_op_#3\endcsname {##4}}%
402     \else
403       \xint_afterfi
404       {\expandafter ##2\expandafter ##3%
405        \csname .#6{\XINT_expr_unlock ##1}{\XINT_expr_unlock ##4}\endcsname }%
406     \fi
407   }%
408   \let #7#5%
409 }%
410 \def\XINT_tmpa #1{\XINT_tmpb {expr}{xint}#1}%
411 \xintApplyInline {\XINT_tmpa }{%
412   {|{iii}{vi}{OR}}%
413   {&{iv}{vi}{AND}}%
414   {<{v}{vi}{Lt}}%
415   {>{v}{vi}{Gt}}%
416   {={v}{vi}{Eq}}%
417   {+{vi}{vi}{Add}}%

```

```

418 {-{vi}{vi}{Sub}}%
419 {*{vii}{vii}{Mul}}%
420 {/{vii}{vii}{Div}}%
421 {^{viii}{viii}{Pow}}%
422 {e{ix}{ix}{fE}}%
423 {E{ix}{ix}{fE}}%
424 }%
425 \def\XINT_tmpa #1{\XINT_tmpb {flexpr}{xint}#1}%
426 \xintApplyInline {\XINT_tmpa }{%
427   |{iii}{vi}{OR}}%
428   &{iv}{vi}{AND}}%
429   <{v}{vi}{Lt}}%
430   >{v}{vi}{Gt}}%
431   ={v}{vi}{Eq}}%
432 }%
433 \def\XINT_tmpa #1{\XINT_tmpb {flexpr}{XINTinFloat}#1}%
434 \xintApplyInline {\XINT_tmpa }{%
435   +{vi}{vi}{Add}}%
436   -{vi}{vi}{Sub}}%
437   *{vii}{vii}{Mul}}%
438   /{vii}{vii}{Div}}%
439   ^{viii}{viii}{Power}}%
440   e{ix}{ix}{fE}}%
441   E{ix}{ix}{fE}}%
442 }%

```

40.14 The comma as binary operator

New with 1.09a.

```

443 \def\XINT_tmpa #1#2#3#4#5#6%
444 {%
445   \def #1##1% \XINT_expr_op_,_a
446   {%
447     \expandafter #2\expandafter ##1\romannumeral-'0\XINT_expr_getnext
448   }%
449   \def #2##1##2% \XINT_expr_until_,_a
450   {\xint_UDsignfork
451     ##2\dummy {\expandafter #2\expandafter ##1\romannumeral-'0#4}%
452     -\dummy {#3##1##2}%
453   \krof }%
454   \def #3##1##2##3##4% \XINT_expr_until_,_b
455   {%
456     \ifnum ##2>\xint_c_ii
457       \xint_afterfi {\expandafter #2\expandafter ##1\romannumeral-'0%
458         \csname XINT_#6_op_##3\endcsname {##4}}%
459     \else
460       \xint_afterfi
461       {\expandafter ##2\expandafter ##3%

```

```

462      \csname .\XINT_expr_unlock ##1,\XINT_expr_unlock ##4\endcsname }%
463      \fi
464    }%
465    \let #5\xint_c_ii
466  }%
467 \expandafter\XINT_tmpa
468   \csname XINT_expr_op_,\expandafter\endcsname
469   \csname XINT_expr_until_,_a\expandafter\endcsname
470   \csname XINT_expr_until_,_b\expandafter\endcsname
471   \csname XINT_expr_op_-vi\expandafter\endcsname
472   \csname XINT_expr_precedence_,\endcsname {expr}%
473 \expandafter\XINT_tmpa
474   \csname XINT_flexpr_op_,\expandafter\endcsname
475   \csname XINT_flexpr_until_,_a\expandafter\endcsname
476   \csname XINT_flexpr_until_,_b\expandafter\endcsname
477   \csname XINT_flexpr_op_-vi\expandafter\endcsname
478   \csname XINT_expr_precedence_,\endcsname {flexpr}%

```

40.15 \XINT_expr_op_-<level>: minus as prefix inherits its precedence level

```

479 \def\XINT_tmpa #1#2%
480 {%
481   \expandafter\XINT_tmppb
482   \csname XINT_#1_op_-#2\expandafter\endcsname
483   \csname XINT_#1_until_-#2_a\expandafter\endcsname
484   \csname XINT_#1_until_-#2_b\expandafter\endcsname
485   \csname xint_c_#2\endcsname {#1}%
486 }%
487 \def\XINT_tmppb #1#2#3#4#5%
488 {%
489   \def #1% \XINT_expr_op_-<level>
490   {% get next number+operator then switch to _until macro
491     \expandafter #2\romannumeral-'0\XINT_expr_getnext
492   }%
493   \def #2##1% \XINT_expr_until_-<l>_a
494   {\xint_UDsignfork
495     ##1\dummy {\expandafter #2\romannumeral-'0#1}%
496     -\dummy {#3##1}%
497   \krof }%
498   \def #3##1##2##3% \XINT_expr_until_-<l>_b
499   {% _until tests precedence level with next op, executes now or postpones
500     \ifnum ##1>#4%
501       \xint_afterfi {\expandafter #2\romannumeral-'0%
502         \csname XINT_#5_op_##2\endcsname {##3}}%
503     \else
504       \xint_afterfi {\expandafter ##1\expandafter ##2%
505         \csname .\xintOpp{\XINT_expr_unlock ##3}\endcsname }%
506     \fi
507   }%

```

```

508 }%
509 \xintApplyInline{\XINT_tmpa {expr}}{\vi}{vii}{viii}{ix}}%
510 \xintApplyInline{\XINT_tmpa {flexpr}}{\vi}{vii}{viii}{ix}}%

```

40.16 ? 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).

```

511 \let\XINT_expr_precedence_? \xint_c_x
512 \def \XINT_expr_op_? #1#2#3%
513 {%
514     \xintifZero{\XINT_expr_unlock #1}%
515                 {\XINT_expr_getnext #3}%
516                 {\XINT_expr_getnext #2}%
517 }%
518 \let\XINT_flexpr_op_?\XINT_expr_op_?

```

40.17 : as three-way conditional

New with 1.09a. Modified in 1.09c to have less precedence than functions.

```

519 \let\XINT_expr_precedence_: \xint_c_x
520 \def \XINT_expr_op_: #1#2#3#4%
521 {%
522     \xintifSgn {\XINT_expr_unlock #1}%
523                 {\XINT_expr_getnext #2}%
524                 {\XINT_expr_getnext #3}%
525                 {\XINT_expr_getnext #4}%
526 }%
527 \let\XINT_flexpr_op_:\XINT_expr_op_:

```

40.18 ! 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.

```

528 \let\XINT_expr_precedence_! \xint_c_x
529 \def\XINT_expr_op_! #1{\expandafter\XINT_expr_getop
530     \csname .\xintFac{\XINT_expr_unlock #1}\endcsname }% [0] removed in 1.09c
531 \let\XINT_flexpr_op_!\XINT_expr_op_!

```

40.19 Functions

New with 1.09a. Names of `..Float...` csv macros have been changed in 1.09h

```

532 \def\XINT_expr_op_@ #1%
533 {%
534   \ifcsname XINT_expr_onlitteral_#1\endcsname
535     \expandafter\XINT_expr_funcoflitteral
536   \else
537     \expandafter\XINT_expr_op_@@
538   \fi {#1}%
539 }%
540 \def\XINT_flexpr_op_@ #1%
541 {%
542   \ifcsname XINT_expr_onlitteral_#1\endcsname
543     \expandafter\XINT_expr_funcoflitteral
544   \else
545     \expandafter\XINT_flexpr_op_@@
546   \fi {#1}%
547 }%
548 \def\XINT_expr_funcoflitteral #1%
549 {%
550   \expandafter\expandafter\csname XINT_expr_onlitteral_#1\endcsname
551   \romannumeral-'0\XINT_expr_scanfunc
552 }%
553 \def\XINT_expr_op_@@ #1%
554 {%
555   \ifcsname XINT_expr_func_#1\endcsname
556     \xint_afterfi{\expandafter\expandafter\csname XINT_expr_func_#1\endcsname}%
557   \else \xintError:unknownfunction
558     \xint_afterfi{\expandafter\XINT_expr_func_unknown}%
559   \fi
560   \romannumeral-'0\XINT_expr_oparen
561 }%
562 \def\XINT_flexpr_op_@@ #1%
563 {%
564   \ifcsname XINT_flexpr_func_#1\endcsname
565     \xint_afterfi{\expandafter\expandafter\csname XINT_flexpr_func_#1\endcsname}%
566   \else \xintError:unknownfunction
567     \xint_afterfi{\expandafter\XINT_expr_func_unknown}%
568   \fi
569   \romannumeral-'0\XINT_flexpr_oparen
570 }%
571 \def\XINT_expr_onlitteral_bool #1#2#3{\expandafter\XINT_expr_getop
572   \csname .\xintBool{#3}\endcsname }%
573 \def\XINT_expr_onlitteral_togl #1#2#3{\expandafter\XINT_expr_getop
574   \csname .\xintToggle{#3}\endcsname }%
575 \def\XINT_expr_func_unknown #1#2#3%
576 {%

```

```

577 \expandafter #1\expandafter #2\csname .0[0]\endcsname
578 }%
579 \def\XINT_expr_func_reduce #1#2#3%
580 {%
581   \expandafter #1\expandafter #2\csname
582     .\xintIrr {\XINT_expr_unlock #3}\endcsname
583 }%
584 \let\XINT_flexpr_func_reduce\XINT_expr_func_reduce
585 \def\XINT_expr_func_sqr #1#2#3%
586 {%
587   \expandafter #1\expandafter #2\csname
588     .\xintSqr {\XINT_expr_unlock #3}\endcsname
589 }%
590 \def\XINT_flexpr_func_sqr #1#2#3%
591 {%
592   \expandafter #1\expandafter #2\csname
593     .\XINTinFloatMul {\XINT_expr_unlock #3}{\XINT_expr_unlock #3}\endcsname
594 }%
595 \def\XINT_expr_func_abs #1#2#3%
596 {%
597   \expandafter #1\expandafter #2\csname
598     .\xintAbs {\XINT_expr_unlock #3}\endcsname
599 }%
600 \let\XINT_flexpr_func_abs\XINT_expr_func_abs
601 \def\XINT_expr_func_sgn #1#2#3%
602 {%
603   \expandafter #1\expandafter #2\csname
604     .\xintSgn {\XINT_expr_unlock #3}\endcsname
605 }%
606 \let\XINT_flexpr_func_sgn\XINT_expr_func_sgn
607 \def\XINT_expr_func_floor #1#2#3%
608 {%
609   \expandafter #1\expandafter #2\csname
610     .\xintFloor {\XINT_expr_unlock #3}\endcsname
611 }%
612 \let\XINT_flexpr_func_floor\XINT_expr_func_floor
613 \def\XINT_expr_func_ceil #1#2#3%
614 {%
615   \expandafter #1\expandafter #2\csname
616     .\xintCeil {\XINT_expr_unlock #3}\endcsname
617 }%
618 \let\XINT_flexpr_func_ceil\XINT_expr_func_ceil
619 \def\XINT_expr_twoargs #1,#2,{\{#1}{#2}}%
620 \def\XINT_expr_func_quo #1#2#3%
621 {%
622   \expandafter #1\expandafter #2\csname .%
623   \expandafter\expandafter\expandafter\xintQuo
624   \expandafter\XINT_expr_twoargs
625   \romannumeral-'0\XINT_expr_unlock #3,\endcsname

```



```

626 }%
627 \let\XINT_flexpr_func_quo\XINT_expr_func_quo
628 \def\XINT_expr_func_rem #1#2#3%
629 {%
630   \expandafter #1\expandafter #2\csname .%
631   \expandafter\expandafter\expandafter\xintRem
632   \expandafter\XINT_expr_twoargs
633   \romannumeral-'0\XINT_expr_unlock #3,\endcsname
634 }%
635 \let\XINT_flexpr_func_rem\XINT_expr_func_rem
636 \def\XINT_expr_oneortwo #1#2#3,#4,#5.%
637 {%
638   \if\relax#5\relax\expandafter\xint_firstoftwo\else
639   \expandafter\xint_secondoftwo\fi
640   {#1{0}{#3}}{#2{\xintNum {#4}}{#3}}}%
641 }%
642 \def\XINT_expr_func_round #1#2#3%
643 {%
644   \expandafter #1\expandafter #2\csname .%
645   \expandafter\XINT_expr_oneortwo
646   \expandafter\xintiRound\expandafter\xintRound
647   \romannumeral-'0\XINT_expr_unlock #3,,.\endcsname
648 }%
649 \let\XINT_flexpr_func_round\XINT_expr_func_round
650 \def\XINT_expr_func_trunc #1#2#3%
651 {%
652   \expandafter #1\expandafter #2\csname .%
653   \expandafter\XINT_expr_oneortwo
654   \expandafter\xintiTrunc\expandafter\xintTrunc
655   \romannumeral-'0\XINT_expr_unlock #3,,.\endcsname
656 }%
657 \let\XINT_flexpr_func_trunc\XINT_expr_func_trunc
658 \def\XINT_expr_argandopt #1,#2,#3.%
659 {%
660   \if\relax#3\relax\expandafter\xint_firstoftwo\else
661   \expandafter\xint_secondoftwo\fi
662   {[ \XINTdigits]{#1}}{[ \xintNum {#2}]{#1}}}%
663 }%
664 \def\XINT_expr_func_float #1#2#3%
665 {%
666   \expandafter #1\expandafter #2\csname .%
667   \expandafter\XINTinFloat
668   \romannumeral-'0\expandafter\XINT_expr_argandopt
669   \romannumeral-'0\XINT_expr_unlock #3,,.\endcsname
670 }%
671 \let\XINT_flexpr_func_float\XINT_expr_func_float
672 \def\XINT_expr_func_sqrt #1#2#3%
673 {%
674   \expandafter #1\expandafter #2\csname .%

```

```

675 \expandafter\XINTinFloatSqrt
676 \romannumeral-'0\expandafter\XINT_expr_argadopt
677 \romannumeral-'0\XINT_expr_unlock #3,,.\endcsname
678 }%
679 \let\XINT_flexpr_func_sqrt\XINT_expr_func_sqrt
680 \def\XINT_expr_func_gcd #1#2#3%
681 {%
682   \expandafter #1\expandafter #2\csname
683     .\xintGCDof:csv{\XINT_expr_unlock #3}\endcsname
684 }%
685 \let\XINT_flexpr_func_gcd\XINT_expr_func_gcd
686 \def\XINT_expr_func_lcm #1#2#3%
687 {%
688   \expandafter #1\expandafter #2\csname
689     .\xintLCMof:csv{\XINT_expr_unlock #3}\endcsname
690 }%
691 \let\XINT_flexpr_func_lcm\XINT_expr_func_lcm
692 \def\XINT_expr_func_max #1#2#3%
693 {%
694   \expandafter #1\expandafter #2\csname
695     .\xintMaxof:csv{\XINT_expr_unlock #3}\endcsname
696 }%
697 \def\XINT_flexpr_func_max #1#2#3%
698 {%
699   \expandafter #1\expandafter #2\csname
700     .\XINTinFloatMaxof:csv{\XINT_expr_unlock #3}\endcsname
701 }%
702 \def\XINT_expr_func_min #1#2#3%
703 {%
704   \expandafter #1\expandafter #2\csname
705     .\xintMinof:csv{\XINT_expr_unlock #3}\endcsname
706 }%
707 \def\XINT_flexpr_func_min #1#2#3%
708 {%
709   \expandafter #1\expandafter #2\csname
710     .\XINTinFloatMinof:csv{\XINT_expr_unlock #3}\endcsname
711 }%
712 \def\XINT_expr_func_sum #1#2#3%
713 {%
714   \expandafter #1\expandafter #2\csname
715     .\xintSum:csv{\XINT_expr_unlock #3}\endcsname
716 }%
717 \def\XINT_flexpr_func_sum #1#2#3%
718 {%
719   \expandafter #1\expandafter #2\csname
720     .\XINTinFloatSum:csv{\XINT_expr_unlock #3}\endcsname
721 }%
722 \def\XINT_expr_func_prd #1#2#3%
723 {%

```

```

724 \expandafter #1\expandafter #2\csname
725     .\xintPrd:csv{\XINT_expr_unlock #3}\endcsname
726 }%
727 \def\XINT_flexpr_func_prd #1#2#3%
728 {%
729     \expandafter #1\expandafter #2\csname
730         .\XINTinFloatPrd:csv{\XINT_expr_unlock #3}\endcsname
731 }%
732 \let\XINT_expr_func_add\XINT_expr_func_sum
733 \let\XINT_expr_func_mul\XINT_expr_func_prd
734 \let\XINT_flexpr_func_add\XINT_flexpr_func_sum
735 \let\XINT_flexpr_func_mul\XINT_flexpr_func_prd
736 \def\XINT_expr_func_? #1#2#3%
737 {%
738     \expandafter #1\expandafter #2\csname
739         .\xintIsNotZero {\XINT_expr_unlock #3}\endcsname
740 }%
741 \let\XINT_flexpr_func_? \XINT_expr_func_?
742 \def\XINT_expr_func_! #1#2#3%
743 {%
744     \expandafter #1\expandafter #2\csname
745         .\xintIsZero {\XINT_expr_unlock #3}\endcsname
746 }%
747 \let\XINT_flexpr_func_! \XINT_expr_func_!
748 \def\XINT_expr_func_not #1#2#3%
749 {%
750     \expandafter #1\expandafter #2\csname
751         .\xintIsZero {\XINT_expr_unlock #3}\endcsname
752 }%
753 \let\XINT_flexpr_func_not \XINT_expr_func_not
754 \def\XINT_expr_func_all #1#2#3%
755 {%
756     \expandafter #1\expandafter #2\csname
757         .\xintANDof:csv{\XINT_expr_unlock #3}\endcsname
758 }%
759 \let\XINT_flexpr_func_all\XINT_expr_func_all
760 \def\XINT_expr_func_any #1#2#3%
761 {%
762     \expandafter #1\expandafter #2\csname
763         .\xintORof:csv{\XINT_expr_unlock #3}\endcsname
764 }%
765 \let\XINT_flexpr_func_any\XINT_expr_func_any
766 \def\XINT_expr_func_xor #1#2#3%
767 {%
768     \expandafter #1\expandafter #2\csname
769         .\xintXORof:csv{\XINT_expr_unlock #3}\endcsname
770 }%
771 \let\XINT_flexpr_func_xor\XINT_expr_func_xor
772 \def\xintifNotZero:: #1,#2,#3,{\xintifNotZero{#1}{#2}{#3}}%

```

```

773 \def\XINT_expr_func_if #1#2#3%
774 {%
775   \expandafter #1\expandafter #2\csname
776     .\expandafter\xintifNotZero::
777     \romannumeral-‘0\XINT_expr_unlock #3,\endcsname
778 }%
779 \let\XINT_flexpr_func_if\XINT_expr_func_if
780 \def\xintifSgn:: #1,#2,#3,#4,{\xintifSgn{#1}{#2}{#3}{#4}}%
781 \def\XINT_expr_func_ifsgn #1#2#3%
782 {%
783   \expandafter #1\expandafter #2\csname
784     .\expandafter\xintifSgn::
785     \romannumeral-‘0\XINT_expr_unlock #3,\endcsname
786 }%
787 \let\XINT_flexpr_func_ifsgn\XINT_expr_func_ifsgn

```

40.20 \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.

```

788 \def\XINT_newexpr_print #1{\ifnum\xintNthElt{0}{#1}>1
789   \expandafter\xint_firstoftwo
790   \else
791   \expandafter\xint_secondoftwo
792   \fi
793   {_xintListWithSep,{#1}}{\xint_firstofone#1}}%
794 \xintForpair #1#2 in {(fl,Float),(num,iRound0),(bool,IsTrue)}\do {%
795   \expandafter\def\csname XINT_new#1expr_print\endcsname
796     ##1{\ifnum\xintNthElt{0}{##1}>1
797       \expandafter\xint_firstoftwo
798       \else
799       \expandafter\xint_secondoftwo
800       \fi
801       {_xintListWithSep,{\xintApply{_xint#2}{##1}}}
802       {_xint#2##1}}}%
803 \toks0 {}%
804 \xintFor #1 in {Bool,Toggle,Floor,Ceil,iRound,Round,iTrunc,Trunc,%
805   Lt,Gt,Eq,AND,OR,IsNotZero,IsZero,ifNotZero,ifSgn,%
806   Irr,Num,Abs,Sgn,Opp,Quo,Rem,Add,Sub,Mul,Sqr,Div,Pow,Fac,fE}\do
807 {\toks0
808   \expandafter{\the\toks0\expandafter\def\csname xint#1\endcsname {_xint#1}}}%
809 \xintFor #1 in {GCDof,LCMof,Maxof,Minof,ANDof,ORof,XORof,Sum,Prd}\do
810 {\toks0
811   \expandafter{\the\toks0\expandafter\def\csname xint#1:csv\endcsname
812     #####1{_xint#1{\xintCSVtoListNonStripped {#####1}}}}}%

```

```

813 \xintFor #1 in {Maxof,Minof,Sum,Prd}\do
814 {\toks0
815   \expandafter{\the\toks0\expandafter\def\csname XINTinFloat#1:csv\endcsname
816     #####1{_XINTinFloat#1{\xintCSVtoListNonStripped {#####1}}}}}%
817 \xintFor #1 in {,Sqrt,Add,Sub,Mul,Div,Power,fE}\do
818 {\toks0
819   \expandafter{\the\toks0\expandafter\def\csname XINTinFloat#1\endcsname
820     {_XINTinFloat#1}}}%
821 \expandafter\def\expandafter\XINT_expr_protect\expandafter{\the\toks0
822   \def\XINTdigits {_XINTdigits}%
823   \def\XINT_expr_print ##1{\expandafter\XINT_newexpr_print\expandafter
824     {\romannumeral0\xintcsvtoListNonStripped{\XINT_expr_unlock ##1}}}%
825   \def\XINT_flexpr_print ##1{\expandafter\XINT_newflexpr_print\expandafter
826     {\romannumeral0\xintcsvtoListNonStripped{\XINT_expr_unlock ##1}}}%
827   \def\XINT_numexpr_print ##1{\expandafter\XINT_newnumexpr_print\expandafter
828     {\romannumeral0\xintcsvtoListNonStripped{\XINT_expr_unlock ##1}}}%
829   \def\XINT_boolexpr_print ##1{\expandafter\XINT_newboolexpr_print\expandafter
830     {\romannumeral0\xintcsvtoListNonStripped{\XINT_expr_unlock ##1}}}%
831 }%
832 \toks0 {}%
833 \def\xintNewExpr {\xint_NewExpr\xinttheexpr }%
834 \def\xintNewFloatExpr {\xint_NewExpr\xintthefloatexpr }%
835 \def\xintNewNumExpr {\xint_NewExpr\xintthenumexpr }%
836 \def\xintNewBoolExpr {\xint_NewExpr\xinttheboolexpr }%
837 \def\xint_NewExpr #1#2[#3]%
838 {%
839   \begingroup
840     \ifcase #3\relax
841       \toks0 {\xdef #2}%
842     \or \toks0 {\xdef #2##1}%
843     \or \toks0 {\xdef #2##1##2}%
844     \or \toks0 {\xdef #2##1##2##3}%
845     \or \toks0 {\xdef #2##1##2##3##4}%
846     \or \toks0 {\xdef #2##1##2##3##4##5}%
847     \or \toks0 {\xdef #2##1##2##3##4##5##6}%
848     \or \toks0 {\xdef #2##1##2##3##4##5##6##7}%
849     \or \toks0 {\xdef #2##1##2##3##4##5##6##7##8}%
850     \or \toks0 {\xdef #2##1##2##3##4##5##6##7##8##9}%
851   \fi
852   \xintexprSafeCatcodes
853   \XINT_NewExpr #1%
854 }%
855 \catcode'* 13
856 \def\XINT_NewExpr #1#2%
857 {%
858   \def\xintTmp ##1##2##3##4##5##6##7##8##9{#2}%
859   \XINT_expr_protect
860   \lccode'\*='_ \lowercase {\def*}{!noexpand!}%
861   \catcode'_ 13 \catcode': 11 \endlinechar -1

```

```

862 \everyeof {\noexpand }%
863 \edef\XINTtmp ##1##2##3##4##5##6##7##8##9%
864     {\scantokens
865       \expandafter{\romannumeral-‘0#1%
866         \xintTmp {####1}{####2}{####3}%
867         {####4}{####5}{####6}%
868         {####7}{####8}{####9}%
869         \relax}}%
870 \lccode‘\*='‘\$ \lowercase {\def*}{####}%
871 \catcode‘\$ 13 \catcode‘! 0 \catcode‘_ 11 %
872 \the\toks0
873 {\scantokens\expandafter{\expandafter
874   \XINT_newexpr_stripprefix\meaning\XINTtmp}}%
875 \endgroup
876 }%
877 \let\xintexprRestoreCatcodes\relax
878 \def\xintexprSafeCatcodes
879 {% for end user.
880   \edef\xintexprRestoreCatcodes {%
881     \catcode63=\the\catcode63 % ?
882     \catcode124=\the\catcode124 % |
883     \catcode38=\the\catcode38 % &
884     \catcode33=\the\catcode33 % !
885     \catcode93=\the\catcode93 % ]
886     \catcode91=\the\catcode91 % [
887     \catcode94=\the\catcode94 % ^
888     \catcode95=\the\catcode95 % _
889     \catcode47=\the\catcode47 % /
890     \catcode41=\the\catcode41 % )
891     \catcode40=\the\catcode40 % (
892     \catcode42=\the\catcode42 % *
893     \catcode43=\the\catcode43 % +
894     \catcode62=\the\catcode62 % >
895     \catcode60=\the\catcode60 % <
896     \catcode58=\the\catcode58 % :
897     \catcode46=\the\catcode46 % .
898     \catcode45=\the\catcode45 % -
899     \catcode44=\the\catcode44 % ,
900     \catcode61=\the\catcode61\relax % =
901   }% this is just for some standard situation with a few made active by Babel
902     \catcode63=12 % ?
903     \catcode124=12 % |
904     \catcode38=4 % &
905     \catcode33=12 % !
906     \catcode93=12 % ]
907     \catcode91=12 % [
908     \catcode94=7 % ^
909     \catcode95=8 % _
910     \catcode47=12 % /

```

```

911      \catcode41=12 % )
912      \catcode40=12 % (
913      \catcode42=12 % *
914      \catcode43=12 % +
915      \catcode62=12 % >
916      \catcode60=12 % <
917      \catcode58=12 % :
918      \catcode46=12 % .
919      \catcode45=12 % -
920      \catcode44=12 % ,
921      \catcode61=12 % =
922 }%
923 \let\XINT_tmpa\relax \let\XINT_tmpb\relax \let\XINT_tmpc\relax
924 \XINT_restorecatcodes_endinput%

```

```

xinttools: 999. Total number of code lines: 10179. Each package starts with
xint: 3490. circa 80 lines dealing with catcodes, package identification and
xintbinhex: 642. reloading management, also for Plain TEX. Version 1.09h of
xintgcd: 468. 2013/11/28.
xintfrac: 2291.
xintseries: 419.
xintcfrac: 946.
xintexpr: 924.

```