The xint bundle: xint, xintgcd, xintfrac, xintseries and xintcfrac.

Jean-François Burnol
jfbu (at) free (dot) fr
Package version: 1.06b (2013/05/14)
Documentation generated from the source file
with timestamp "14-05-2013 at 22:00:09 CEST"

Abstract

The xint package implements with expandable TEX macros the basic arithmetic operations of addition, subtraction, multiplication and division, as applied to arbitrarily long numbers represented as chains of digits with an optional minus sign. The xintgcd package provides implementations of the Euclidean algorithm and of its typesetting.

The xintfrac package extends the scope of xint to fractional numbers of arbitrary sizes ; xintseries provides some basic functionality for computing in an expandable manner partial sums of series and power series with fractional coefficients. And xintcfrac deals with the computation of continued fractions.

The packages may be used with Plain and with LATEX. Most macros, and all of those doing computations, work purely by expansion without assignments, and may thus be used almost everywhere in TEX.

Contents

1 Raison d'être of these packages

1.1 Some examples

The main goal is to allow computations with integers and fractions of arbitrary sizes.[1]

Here are some examples:

123456^99:

\xintiPow{123456}{99}: 11473818116626655663327333000845458674702548042342610297588954543735908946970320276226470542663205834690270868221168133415250032403876277616895322211763429587203376221608860691585075716801971675

[1] Here and elsewhere, "arbitrarily big" means roughly numbers with numerators and denominators having strictly less than 2^{31}=2147483648 digits. Memory constraints from the etex or pdftex executables presumably limit even more the possible computations, not to mention the time taken by them.

1071208769703353650737748777873778498781606749999798366581251723275215497054165956673849115333267485410756076697189062351899583237782636999811095323939932351899922205645878127014958776791431677354372538584459487155941215197416398666125896983737258716757394949435520170950261865801665199030718414432231169678376 96

1234/56789 with 1500 digits after the decimal point:

```
\xintTrunc{1500}{1234/56789}\dots
```

: 0.021729560302171195125816619415731919914067865255595273732589057740055292398175703041081899663667259504481501699272746482593460001056542640300058109845216503196041486907675782281779922168025497895719241402384264558277131134550705242212400288788321682015883357692510873584673088098047157019845392593636091496592649985032312595749176777192766204722745602141259751008117769286305446477310746799556252091073975593865009068657662575498776171441652432689429290883797918610998608885523604923488703798270791878708904893553328989769145433094437302998820194051664935110672841571431087006286428709785345753579038194016446847100670904576590536899751712479529486344186374121748930250576696191163781718290514007994505978270439697828804874183380584268080085932134744404726267410942259944707601824296958918100336332740495518498300727253517406539998943457359699941890154783496803958513092324217718220077831974502104280758597615735441722868865449294757787599711211678317984116642307489126415326911901952842980154607406363908503407350014967687404250823222807233795277254397858740248991882230713694553522689253200443747908926024406134990931342337424501223828558347567310570709116202081389001391114476395076511296201729208121291095106446671010230854566905562697001179805948335064889327158428568912993713571290214654246420961805983553152899329095423409463100248287520470513655813625878251069749423303808836218281709485992005494021729560302171195125816619415731919914067865255595273732589057740055292398175703041081899663667...

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

```
\xintTrunc{200}{\xintPow{.99}{-100}}\dots
```

: 2.731999026429026003846671721257837435505351642938572070833430572508246455518705343044814301378480614036805562476501925307034269685489153194616612271015920671913840348851485747943086470963920731779793 03...

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

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

-220045702773594816771390169652074193009609478853×(7^200-3^200)+14325894936276369318591306832683204654744168633877140891583816724789919211328201191274624371580391777549768571912876931442406050669914563361432056776967 74891×(2^200-1)=1803403947125

The Euclide algorithm applied to 179, 876, 541, 573 and 66, 172, 838, 904:[2]

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

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

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

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

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

The complete series, extended to infinity, has value $\frac{\pi^2}{144}-\frac{1}{162} = 0.062,366,079,945,836,595,346,844,45\ldots$ [3] 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]}}
```

The first example uses only the base module **xint**, the next two require loading also the **xintfrac** package, which deals with fractions. Then two examples with the **xintgcd** package, and finally one with the **xintseries** package, for partial sums of series with fractional coefficients. There is also **xintcfrac** for continued fractions computations.

To see more of **xint** in action, jump to the section 17 describing the commands of the **xintseries** package, especially as illustrated with the traditional computations of π and $\log 2$, or also see the computation of the convergents of e made with the **xintcfrac** package.

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

1.2 Expandability, (in)-efficiency

For some initially circumstantial reasons (related to the origins of the package) all macros performing computations are compatible with an expansion-only context. This programming constraint of expandability weighs in a lot on the computation time as the macros may have to shuffle around data containing hundreds of tokens: our current implementation of addition doesn't even achieve linear computation time!

For addition, I try to optimize things for the 50-500 digits range. I have a variant of addition which is twice faster on numbers with 1000 digits, but it is slower than the original

[3] This number is typeset using the numprint package, with `\npthousandsep {,\hskip .16667em plus .01em minus .01em}`. But the breaking accross lines works only in text mode. The number itself was (of course...) computed initially with **xint**, with 30 digits of π as input. See how **xint** may compute π from scratch.

for numbers with less than 200 digits, and adding to the code a fork to choose what to do would mean overhead; besides it wouldn't be that easy to use this variant of addition in the other routines such as multiplication and division. And multiplication is anyhow too slow on numbers with 1000 digits, even dividing the time by two would not be enough.

Analogously to the not even linear addition, multiplication is worse than quadratic. Same causes, same effects. It is about cubic in the 100-1000 digits range: on my laptop, with release `1.04` of the bundle, squaring a randomly chosen number with 200 digits takes about 4 hundredths of a second, and squaring a 400 digits number about a quarter of a second. But squaring a 500 digits number is about 1.9 times as costly as one with 400 digits, and squaring a 1000 digits number is 8 times more expensive than for a 500 digits number (about 3.5 seconds). Implementation of a Gauss-Karatsuba scheme for intelligent multiplication has not been attempted so far. This kind of thing is motivating when one has instant memory access!

As clearly demonstrated long ago by the pi computing file by D. Roegel one can program TEX to compute with many digits at a much higher speed than what **xint** achieves: but, direct access to memory storage in one form or another seems a necessity for this kind of speed and one has to renounce at the complete expandability.[4] [5]

1.3 Missing things

Currently **xint** does not provide 'floating-point' operations. The LATEX3 project has implemented expandably floating-point computations with 16 significant figures (l3fp), including special functions such as exp, log, sine and cosine.

The most blatantly lacking thing in the **xint** project is a decent input parser, allowing to type in computations in a usual infix form such as, for example `3*14+2.7^-2*5`. At this time, one has to type `\xintAdd {\xintMul {3}{14}}{\xintMul{\xintPow{2.7}{-2}}{5}}`. Previous computation results can be stored in macros and given as arguments to the package macros (see further on for important aspects of this).

1.4 Origins of the package

Package `bigintcalc` by Heiko Oberdiek already provides expandable arithmetic operations on "big integers", exceeding the TEX limits (of 2^{31}-1), so why another[6] 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.[7] What I had learned in this other thread thanks to interaction with Ulrich Diez and GL on expandable manipulations of tokens motivated me to try my hands at addition and multiplication.

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

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

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

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

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

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.

xint requires the ε-TEX \numexpr primitive.

2 Expansions

Except for some specific macros dealing with assignments or typesetting, the bundle macros all work in expansion-only context. For example, with the following code snippet within myfile.tex:

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

the tex run creates a file myfile-out.tex containing the decimal representation of the integer quotient 2^{1000}/100!. Such macros can also be used inside a \csname...\endcsname, and of course in an \edef.

Furthermore the package macros give their final results in two expansion steps. They expand 'fully' (the first token of) their arguments so that they can be arbitrarily chained. Hence

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

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

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

```
\def\allowsplits #1{\ifx #1\relax \else #1\hskip 0pt plus 1pt \relax
                    \expandafter\allowsplits\fi}%
\def\printnumber #1{\expandafter\expandafter\expandafter
                    \allowsplits #1\relax }%
% Expands twice before printing.
```

The \printnumber macro is not part of the package and would need additional thinking for more general use.[8] 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}}}
```

[8] as explained in a previous footnote, the numprint package may also be used, in text mode only (as the thousand separator seemingly ends up typeset in a \hbox when in math mode).

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

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

```
\printnumber {\xintTrunc {300}{\xintPow{.7}{-25}}}\dots
```

7456.739985837358837609119727341853488853339101579533584812792108394305337246328231852818407506767353741490769900570763145015081436139227188742972826645967904896381378616815228254509149848168782309405985245368923678816256779083136938645362240130036489416562067450212897407646036464074648484309937461948589...

This computation uses **xintfrac** wich extends to fractions the basic arithmetic operations defined for integers by **xint**.

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

CHANGED! (1.06) →

1. the macros expand 'fully' their arguments, this means that they expand the first token seen (for each argument), then expand again, etc..., until something un-expandable such as a digit or a brace is hit against.[9] This example

   ```
   \def\x{12}\def\y{34}\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-routiunes of \xintAdd. It works here by sheer luck as the \y gets expanded inside a \numexpr. But this would fail in general: if you need a more complete (expandable...) expansion of your initial input, you should use the \bigintcalcNum macro from the bigintcalc package. Or, outside of an expandable-only context, just massage your inputs through \edef's.

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

3. With the definition

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

   one obtains an expandable macro producing the expected result, not in two, but rather in three steps: a first expansion is consumed by the macro expanding to its definition. The new expansion policy starting with the package release 1.06 allows to use this inside other package 'primitives' or also similar macros: \xintAdd {\AplusBC {1}{2}{3}}{4} does work and returns 11/1[0].[10]

   If, for some reason, it is important to create a macro expanding in two steps to its final value, the solution is to 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.

[9] the knowledgeable people will have recognized \romannumeral-`0

[10] this strange thing is because this document uses **xintfrac**, and we have printed the raw output of addition which is automatically a fraction.

The lowercase form is *only* for the external highest level of chained commands. All **xint** provided public macros have such a lowercase form. To more fully imitate the **xint** standard habits, the example above should thus be treated via the creation of two macros:

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

Or, for people using the LaTeX vocabulary:

```
\newcommand{\aplusbc}[3]{\xintadd {#1}{\xintMul {#2}{#3}}}
\newcommand{\AplusBC}{\romannumeral0\aplusbc}
```

This then allows further definitions of macros expanding in two steps only, such as:

```
\def\aplusbcsquared #1#2#3{\aplusbc {#1}{#2}{\xintSqr{#3}}}
\def\AplusBCSquared {\romannumeral0\aplusbcsquared}
\newcommand\myalgebra [6]{\xintmul {\AplusBC {#1}{#2}{#3}}{\AplusBC
{#4}{#5}{#6}}}
\newcommand\MyAlgebra {\romannumeral0\myalgebra}
```

The `\romannumeral0` 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.

New with `1.06`: those macro arguments which are intrinsically constrained to obey the TeX bounds on integers (see the next section) are now systematically fed to a `\numexpr`, hence they will be subjected to a complete expansion, registers are allowed, and things such as `\mycount+\myothercount*17` become admissible arguments.

3 Inputs and outputs

The arguments to most of the **xint** macros are of three types:

1. 'short' integers, *i.e.* less than (or equal to) in absolute value 2, 147, 483, 647. I will refer to this as the 'TeX' or '`\numexpr`' limit. This is the case for arguments which serve to count or index something. It is also the case for the exponent in the power function and for the argument to the factorial function. The bounds have been (arbitrarily) lowered to 999, 999, 999 and 999, 999 respectively for the latter cases. 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 macros.

2. 'long' integers, which are the bread and butter of the package commands. They are signed integers with a practically illimited number of digits. Theoretically though, most of the macros require that the number of digits itself be less than the TeX-`\numexpr` bound (more precisely 2^31-9). Some macros, such as addition when **xintfrac** has not been loaded, do not measure first the length of their arguments and could theoretically be used with 'gigantic' integers with a larger number of digits. However memory constraints from the TeX implementation probably exclude

such inputs. Concretely though, multiplying out two 1000 digits numbers is already a longish operation.

3. 'fractions': they become available after having loaded the **xintfrac** package. Their format on input will be described next, a fraction has a numerator, a forward slash and then a denominator.

NEW WITH 1.06 →

TEX's count registers cannot serve directly as arguments to the package macros accepting 'long numbers' or fractions on input: they must be prefixed by `\the` or `\number`. The same for `\numexpr` expressions. However, count registers and `\numexpr` expressions are allowed in arguments intrinsically constrained to obey the TEX bounds.

NEW WITH 1.06 → The package macros first operate a 'full' expansion of their arguments, as explained above: only the first token is repeatedly expanded until no more is possible.

NEW WITH 1.06 → On the other hand, this expansion is a *complete one* for those arguments which are constrained to obey the TEX bounds on numbers, as they are systematically inserted inside a `\numexpr...\relax` expression.

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

1. the strict format is when **xintfrac** is not loaded. The number should be a string of digits, optionally preceded by a unique minus sign. The first digit can be zero only if the number is zero. A plus sign is not accepted. There is a macro `\xintNum` which normalizes to this form an input having arbitrarily many minus and plus signs, followed by a string of zeros, then digits:
   `\xintNum {+-+-+----++-++----0000000009876543210}=-9876543210`
   Note that `-0` is not legal input and will confuse **xint** (but not `\xintNum` which even accepts an empty input).

2. the relaxed format is when **xintfrac** is loaded. Most macros are then modified to accept inputs of the form `A/B` (or just `A`), where `A` and `B` will be automatically given to the normalizing `\xintNum` macro. Additionally, each of `A` and `B` may have an optional decimal point with digits following it. Here is an example:
   `\xintAdd {+--0367.8920280/-++278.289287}{-109.2882/+270.12898}`
   Incidentally this evaluates to
   =-129792033529284840/7517400124223726[-1]
   =-6489601676464242/3758700062111863 (irreducible)
   =-1.726554811297710936942487048986778815563600552428 06...
   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.

Of course, even when **xintfrac** is loaded, some macros can not treat fractions on input. With release `1.05` they have, for the most part, been also extended to accept the relaxed format as long as the denominator turns out to be a divisor of the numerator (once the decimal points are suitably transformed into powers of ten). For example it used to be the case with the earlier releases that `\xintQuo {100/2}{12/3}` would not work (the macro `\xintQuo` computes a euclidean quotient). It now does, because its arguments are in truth integers.

A number can start directly with a decimal point:

```
\xintPow{-.3/.7}{11}=-177147/1977326743[0]
```

It is also licit to use `\A/\B` as input if each of `\A` and `\B` expands (in the sense previously described) to a "decimal number" as examplified above by the numerators and denominators. Or one may have just one macro `\C` which expands to such a "fraction with optional decimal points", or mixed things such as `\A 245/7.77`, where the numerator will be the concatenation of the expansion of `\A` and `245`. But, as explained already `123\A` is a no-go.

Loading **xintfrac** not only relaxes the format of the inputs; it also modifies the format of the outputs: except when filtered through the `\xintIrr` (and `\xintJrr`) or `\xintRaw` macros, a fraction is always output in the `A/B[n]` form (which stands for (A/B)10^n). The `A` and `B` may end in zeros (*i.e*, `n` does not represent all powers of ten), and will generally have a common factor. The denominator `B` is always strictly positive.

A macro `\xintFrac` is provided for the typesetting (math-mode only) of such a 'raw' output. Of course, the `\xintFrac` itself is not accepted as input to the package macros.

Direct user input of things such as `16000/289072[17]` or `3[-4]` is authorized. It is even possible to use `\A/\B[17]` if `\A` expands to `16000` and `\B` to `289072`, or `\A` if `\A` expands to `3[-4]`. However, NEITHER the numerator NOR the denominator may then have a decimal point. And, for this format, ONLY the numerator may carry a UNIQUE minus sign (and no superfluous leading zeros; and NO plus sign).

IMPORTANT! →

The, more demanding, 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 very important 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 do not make the package switch to a (currently non-existent) 'floating-point' mode. The inputs, however long, are always converted into an exact internal representation.

Generally speaking, there should be no spaces among the digits in the inputs. 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.

It would certainly be nice to be able to input directly expressions such as `2.3*5.6^3-17728/189.5`, but this is not possible. One must use, for example:

```
\xintSub {\xintMul {2.3}{\xintPow {5.6}{3}}} {17728/189.5}
```

or, an option in this case is:

```
\xintAdd {\xintPrd {{2.3}{5.6}{5.6}{5.6}}}{-17728/189.5}
```

Syntax such as `\xintMul\A\B` is accepted and equivalent[11] to `\xintMul {\A}{\B}`. Or course `\xintAdd\xintMul\A\B\C` does not work, the product operation must be put within braces: `\xintAdd{\xintMul\A\B}\C`. It would be nice to have a functional form `\add(x,\mul(y,z))` but this is not provided by the package. Arguments must be either within braces or a single control sequence.

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

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

4 More on fractions

With package **xintfrac** loaded, the routines \xintAdd, \xintSub, \xintMul, \xintPow, \xintSum, \xintPrd are modified to allow fractions on input,[12] [13] [14] [15] and produce on output a fractional number f=A/B[n] where A and B are integers, with B positive, and n is a signed "small" integer (*i.e* less in absolute value than 2^{31}-9). This represents (A/B) times 10^n. The fraction f may be, and generally is, reducible, and A and B may well end up with zeros (*i.e.* n does not contain all powers of 10). Conversely, this format is accepted on input (and is parsed more quickly than fractions containing decimal points; the input may be a number without denominator).[16]

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

The macro \xintRaw prints the fraction in A/B form, the former trailing [n] having been converted into explicit zeros either at the numerator or the denominator. 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 only the C if D=1. The macro \xintNum from **xint** is extended to act like \xintIrr but additionally raises an error when the fraction doesn't simplify to an integer. When one knows that necessarily the result of a computation is an integer, and one wants to get rid of the denominator and trailing [n], one can thus use \xintIrr or \xintNum (if the fraction has internally a denominator equal to 1, this is quickly identified, there is little overhead; else, the denominator will be discovered in the next step to be a divisor of the numerator).

The macro \xintTrunc{N}{f} prints[17] the decimal expansion of f with N digits after the decimal point.[18] Currently, it does not verify that N is non-negative and strange things

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

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

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

[15] and \xintQuo, \xintRem, \xintDivision, \xintGeq, \xintFDg, \xintLDg, \xintOdd, \xintMON, \xintMMON all accept a fractional input as long as it reduces to an integer. Note that \xintGeq still only works on (non-negative) integers, to compare fractions one must use \xintCmp.

[16] at each stage of the computations, the sum of n and the length of A, or of the absolute value of n and the length of B, must be kept less than 2^{31}-9.

[17] '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.

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

could happen with a negative N. Of course a negative f is no problem, needless to say. When the original fraction is negative and its truncation has only zeros, it is printed as -0.0...0, with N zeros following the decimal point:

```
\xintTrunc {5}{\xintPow {-13}{-9}}=-0.00000
\xintTrunc {20}{\xintPow {-13}{-9}}=-0.00000000009429959537
```

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

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

The output of \xintTrunc may of course serve as input to the other macros. And this is almost necessary when summing hundreds of terms of a series with fractional coefficients, as the exact rational number quickly becomes quite big (when doing the sum from n=1 to n=1000 of 1/n, the raw denominator is 1000!, which has 2568 digits) ; but for less than fifty terms with small denominators it is often possible to work with the exact value without too much toll on the compilation time.

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. This is also convenient when computing partial sums of series, with a fixed number of digits after the decimal point: it is a bit faster to sum with \xintiSeries the integers produced by \xintiTrunc{N} than it is to use the general \xintSeries on the decimal numbers produced by \xintTrunc{N}. These latter macros belong to the **xintseries** package.

Needless to say when using \xintTrunc or \xintiTrunc on intermediate computations the ending digits of the final result are, pending further analysis, only indications of those of the fraction an exact computation would have produced.

To get the integer part of the decimal expansion of f, use \xintiTrunc{0}{f}:

```
\xintiTrunc {0}{\xintPow {1.01}{100}}=2
\xintiTrunc {0}{\xintPow{0.123}{-10}}=1261679032
```

5 \ifcase, \ifnum, ... constructs

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

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

6 Multiple outputs

Some macros have an output consisting of more than one number, each one is then within braces. Examples of multiple-output macros are \xintDivision which gives first the quotient and then the remainder of euclidean division, \xintBezout from the **xintgcd** package which outputs five numbers, \xintFtoCv from the **xintcfrac** package which returns the list of the convergents of a fraction, ... the next two sections explain ways to deal, expandably or not, with such outputs.

See the subsection 14.44 for a rare example of a bundle macro which may return an empty string, or a number prefixed by a chain of zeros. This is the only situation where a macro from the package **xint** may output something which could require parsing through `\xintNum` before further processing by the other (integer-only) package macros from **xint**.

7 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:->1148132496415075054822783938725510662598055177841861728836634780658265418947047379704195357988766304843582650600615037495317077931186277748296 01` 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)×357-(-10)×323= 17 is a Bezout Identity.

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

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

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

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

This defines `\Out` to be macro with one parameter, `\Out{0}` gives the size `N` of the array and `\Out{n}`, for `n` from 1 to `N` then gives the `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:

CHANGED (1.06)!

```
\newcount\cnta
\newcount\cntb
\begingroup
\xintDigitsOf\xintiPow{2}{100}\to\Out
\cnta = 1
\cntb = 0
\loop
\advance \cntb \xintiSqr{\Out{\cnta}}
\ifnum \cnta < \Out{0}
\advance\cnta 1
\repeat
```

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

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

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

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

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

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

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

Indeed, `\xintiSum` is usually used as in

```
\xintiSum{{123}{-345}{\xintFac{7}}{\xintiOpp{\xintRem{3347}{591}}}}=4426
```

but in the example above each digit of 2^{100} is treated as would have been a summand enclosed within braces, due to the rules of TEX for parsing macro arguments.

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

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

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

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

8 Utilities for expandable manipulations

EXTENDED (1.06) → 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`: `\xintRev`, `\xintReverseOrder`, `\xintLen` and `\xintLength` since the first release, `\xintApply` and `\xintListWithSep` since `1.04`, `\xintRevWithBraces`, `\xintCSVtoList`, `\xintNthElt` with `1.06`, and `\xintApplyUnbraced`, new with `1.06b`.

As an example the following code uses only expandable operations:

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

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

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

9 Exceptions (error messages)

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

```
\xintError:ArrayIndexIsNegative
\xintError:ArrayIndexBeyondLimit
\xintError:FactorialOfNegativeNumber
\xintError:FactorialOfTooBigNumber
\xintError:DivisionByZero
\xintError:NaN
\xintError:FractionRoundedToZero
\xintError:NotAnInteger
\xintError:ExponentTooBig
\xintError:TooBigDecimalShift
\xintError:TooBigDecimalSplit
\xintError:NoBezoutForZeros
```

10 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`.
- 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]`.
- using `[]` with a sign in the denominator `3/-5[7]`.

- loading **xintfrac** and using expressions previously producing integers as numerators or denominators: `\edef\x{\xintMul {3}{5}/\xintMul{7}{9}}`. The problem is that 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.

11 Package namespace

Inner macros of **xint**, **xintgcd**, **xintfrac**, **xintseries**, and **xintcfrac** all begin either with `\XINT_` or with `\xint_`.[19] The package public commands all start with `\xint`. The major forms have their initials capitalized, and lowercase forms, prefixed with `\romannumeral0`, allow definitions of further macros expanding in only two steps to their final outputs. 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.

12 Loading and usage

```
Usage with LaTeX: \usepackage{xint}
                  \usepackage{xintgcd}    % (loads xint)
                  \usepackage{xintfrac}   % (loads xint)
                  \usepackage{xintseries} % (loads xintfrac)
                  \usepackage{xintcfrac}  % (loads xintfrac)

Usage with TeX:   \input xint.sty\relax
                  \input xintgcd.sty\relax    % (loads xint)
                  \input xintfrac.sty\relax   % (loads xint)
                  \input xintseries.sty\relax % (loads xintfrac)
                  \input xintcfrac.sty\relax  % (loads xintfrac)
```

We have added, directly copied from packages by Heiko Oberdiek, a mecanism of reload and ε-TeX detection, especially for Plain TeX. As ε-TeX is required, the executable `tex` can not be used, `etex` or `pdftex` (version `1.40` or later) or ..., must be invoked.

Furthermore, the packages **xintgcd** and **xintfrac** will check for the previous loading of **xint**, and will try to load it if this was not already done. Similarly **xintseries** and **xintcfrac** do the necessary loading of **xintfrac**.

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

This is for the loading of the packages. For the actual use of the macros, note that when feeding them with negative numbers the minus sign must have category code other, as

[19] starting with release `1.06b` the style files use for macro names a more modern underscore _ rather than the @ sign. Probability of a name clash with LaTeX2e packages is now even closer to nil, and with LaTeX3 packages it is also close to nil as our control sequences are all lacking the argument specifier part of LaTeX3 function names.

is standard. Similarly the slash used for inputting fractions must be of category other, as usual. And the square brackets also must be of category code other, if used on input.

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

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

13 Installation

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

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

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

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

           xint.sty |
        xintgcd.sty |
       xintfrac.sty | --> TDS:tex/generic/xint/
     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.
```

14 Commands of the xint package

{N} (or also {M}) stands for a normalised number within braces as described in the documentation, or for a control sequence expanding (in the sense previously described) to such a number (without the braces!), or for a control sequence within braces expanding to such a number, of for material within braces which expands to such a number after repeated expansions of the first token. A count register or \numexpr expression must thus come first and be prefixed by \the or \number.

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

Some of these macros are extended by **xintfrac** to accept fractions on input, and, generally, to output a fraction. This will be mentioned and the original macro `\xintAbc` remains then available under the name `\xintiAbc`. There are also macros such as `\xintQuo` or `\xintNum` which are made to accept fractions on input, under the condition that this fraction turns out to be an integer, but still do produce pure integers without any forward slash mark nor trailing `[n]`. Again the original is still available with an additional 'i' in the name, in case it is important to skip the parsing, but here the output format is the same. See the **xintfrac** documentation for more information.

The integer-only macros are a bit more efficient, even for simple things such as determining the sign of a (long) number, as there is always some overhead due to the parsing the fraction format on input; however except if one does thousands of times the same computation with various inputs, there is no need in general to employ the integer-only variants. The exception is when the context requires that the macro returns a (possibly long) integer, with no forward slash nor trailing `[n]`. This may be because they are used in **xint** macros which remain strictly integer-only on input, such as `\xintDecSplit`, or in places where a (short) number is expected by TEX such as after an `\ifnum` or inside a `\numexpr`.

IMPORTANT!

14.1 \xintRev

`\xintRev{N}` will revert the order of the digits of the number, keeping the optional sign. Leading zeros resulting from the operation are not removed (see the `\xintNum` macro for this). As all other macros dealing with numbers it first expands its argument (in the manner described, triggered by a `\romannumeral-‘0`).

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

14.2 \xintReverseOrder

`\xintReverseOrder{`⟨*list*⟩`}` does not do any expansion of its argument and just reverses the order of the tokens in the ⟨*list*⟩.[20] Brace pairs encountered are removed once and the enclosed material does not get reverted. Spaces are gobbled.

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

14.3 \xintRevWithBraces

New in release 1.06.

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

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

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

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

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

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

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

14.4 \xintLen

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

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

Extended by xintfrac to fractions: the length of A/B[n] is the length of A plus the length of B plus the absolute value of n and minus one (an integer input as N is internally N/1[0] so the minus one means that the extended \xintLen behaves the same as the original for integers). The whole thing should sum up to less than circa 2^{31}.

14.5 \xintLength

\xintLength{⟨*list*⟩} does not do any expansion of its argument and just counts how many tokens there are (possibly none). Things enclosed in braces count as one.

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

14.6 \xintCSVtoList

New with release 1.06.

\xintCSVtoList{a,b,c...,z} returns {a}{b}{c}...{z}. The argument may be a macro. It is first expanded: this means that if the argument is a,b,.., then a, if a macro, will be expanded which may or may not be a good thing (starting the replacement text of the macro with \space stops the expansion at the first level and gobbles the space; prefixing a macro with \space stops preemptively the expansion and gobbles the space). Chains of contiguous spaces are collapsed by the TEX scanning into single spaces.

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

14.7 \xintNthElt

New in release 1.06 and modified in 1.06a.

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

```
\xintNthElt {37}{\xintFac {100}}=9
```

is the thirty-seventh digit of 100!.

```
\xintNthElt {10}{\xintFtoCv {566827/208524}}=1457/536[0]
```

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

If x=0 or x<0, the macro returns the length of the expanded list: this is not equivalent to \xintLength due to the initial full expansion of the first token, and differs from \xintLen which is to be used on numbers or fractions only. The situation with x larger than the length of the list is kept silent, the macro then returns nothing; this will perhaps be modified in future versions.

```
\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
```

14.8 \xintListWithSep

New with release 1.04.

\xintListWithSep{sep}{⟨*list*⟩} just inserts the given separator sep in-between all elements of the given list: this separator may be a macro but will not be expanded. The second argument also may be itself a macro: it is expanded as usual, *i.e.* fully for what comes first. Applying \xintListWithSep removes one level of top braces to each list constituent. An empty input gives an empty output, a singleton gives a singleton, the separator is used starting with at least two elements. Using an empty separator has the net effect of removing one-level of brace pairs from each ot the top-level braced material constituting the ⟨*list*⟩.

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

14.9 \xintApply

New with release 1.04.

\xintApply{\macro}{⟨*list*⟩} applies the one parameter command \macro to each item in the ⟨*list*⟩ (no separator) given as second argument. Each item is given in turn as parameter to \macro which is expanded (as usual, *i.e.* fully for what comes first), and the result is braced. On output, a new list with these braced results. The ⟨*list*⟩ may itself be some macro expanding (in the previously described way) to the list of tokens to which the command \macro will be applied. For example, if the ⟨*list*⟩ expands to some positive number, then each digit will be replaced by the result of applying \macro on it.

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

14.10 \xintApplyUnbraced

New in release 1.06b.

\xintApplyUnbraced{\macro}{⟨*list*⟩} is like \xintApply except that the various outputs are not again braced. The net effect is the same as doing

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

This command is useful for non-expandable things like doing macro definitions, for which braces are an inconvenience. (sorry for the silly example:)

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

14.11 \xintAssign

\xintAssign⟨*braced things*⟩\to⟨*as many cs as they are things*⟩ defines (without checking if something gets overwritten) the control sequences on the right of \to to be the complete expansions of the successive things on the left of \to enclosed within braces.

Important: a 'full' expansion (as previously described) is applied first to the material in front of \xintAssign.

As a special exception, if after this initial expansion a brace does not immediately follows \xintAssign, it is assumed that there is only one control sequence to define and it is then defined to be the complete expansion of the entire material between \xintAssign and \to.

```
\xintAssign\xintDivision{1000000000000}{133333333}\to\Q\R
    \meaning\Q: macro:->7500, \meaning\R: macro:->2500
\xintAssign\xintiPow {7}{13}\to\SevenToThePowerThirteen
        \SevenToThePowerThirteen=96889010407
```

Of course this macro and its cousins completely break usage in pure expansion contexts, as assignments are made via the \edef primitive.

14.12 \xintAssignArray

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

\xintAssignArray⟨*braced things*⟩\to\myArray first expands fully the first token then defines \myArray to be a macro with one parameter, such that \myArray{x} expands in two steps (which provoke the full expansion of the 'short' number {x}, given to a \numexpr) to give the x^th^ braced thing, itself completely expanded. \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: $(-20)\times 1000 - (-177)\times 113 = 1$.

14.13 \xintRelaxArray

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

14.14 \xintDigitsOf

This is a synonym for \xintAssignArray, to be used to define an array giving all the digits of a given 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.

14.15 \xintNum

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

```
\xintNum{+---++----+--00000000367941789479}=-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
```

14.16 \xintSgn

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

14.17 \xintOpp

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

14.18 \xintAbs

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

14.19 \xintAdd

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

14.20 \xintSub

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

14.21 \xintCmp

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

14.22 \xintGeq

`\xintGeq{N}{M}` returns 1 if the absolute value of the first number is at least equal to the absolute value of the second number. If `|N|<|M|` it returns 0.

14.23 \xintMax

`\xintMax{N}{M}` returns the largest of the two in the sense of the order structure on the relative integers (*i.e.* the right-most number if they are put on a line with positive numbers on the right): `\xintiMax {-5}{-6}=-5`. Extended by xintfrac to fractions.

14.24 \xintMin

`\xintMin{N}{M}` returns the smallest of the two in the sense of the order structure on the relative integers (*i.e.* the left-most number if they are put on a line with positive numbers on the right): `\xintiMin {-5}{-6}=-6`. Extended by xintfrac to fractions.

14.25 \xintSum

\xintSum{⟨*braced things*⟩} after expanding its argument expects to find a sequence of tokens (or braced material). Each is expanded (with the usual meaning), and the sum of all these numbers is returned.

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

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

14.26 \xintSumExpr

\xintSumExpr⟨*braced things*⟩\relax is to what \xintSum expands. The argument is then expanded (with the usual meaning) and should give a list of braced quantities or macros, each one will be expanded in turn.

```
\xintiSumExpr {123}{-98763450}{\xintFac{7}}{\xintiMul{3347}{591}}\relax=-96780210
```

Note: I am not so happy with the name which seems to suggest that the + sign should be used instead of braces. Perhaps this will change in the future.

Extended by xintfrac to fractions.

14.27 \xintMul

Modified in release 1.03.

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

14.28 \xintSqr

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

14.29 \xintPrd

\xintPrd{⟨*braced things*⟩} after expanding its argument expects to find a sequence of tokens (or braced material). Each is expanded (with the usual meaning), and the product of all these numbers is returned.

```
\xintiPrd{{-9876}{\xintFac{7}}{\xintiMul{3347}{591}}}=-98458861798080
                \xintiPrd{12345789123456789}=131681894400
```

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

```
                                2^{200}3^{100}7^{100}
=\xintiPrd {{\xintiPow {2}{200}}{\xintiPow {3}{100}}{\xintiPow {7}{100}}}
```

=267872793166157757576627951700754840232474026637401534897445961481542641296549949000044400724076572713000016531207640654562118014357199401590334353924402821243896682224892786298808438271613337 6
=\xintiPow {\xintiMul {\xintiPow {42}{9}}{43008}}{10}

Extended by xintfrac to fractions.

14.30 \xintPrdExpr

Name change in 1.06a! I apologize, but I suddenly decided that \xintProductExpr was a bad choice; so I just replaced it by the current name.

\xintPrdExpr{⟨*argument*⟩}\relax is to what \xintPrd expands ; its argument is expanded (with the usual meaning) and should give a list of braced numbers or macros. Each will be expanded when it is its turn.

\xintiPrdExpr 12345678912345678 9\relax=131681894400

Note: I am not so happy with the name which seems to suggest that the * sign should be used instead of braces. Perhaps this will change in the future.

Extended by xintfrac to fractions.

14.31 \xintFac

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

14.32 \xintPow

\xintPow{N}{x} returns N^x. When x is zero, this is 1. Some cases (N zero and x negative, |N|>1 and x negative, |N|>1 and x at least 10^9) make xint throw errors.

Extended by xintfrac to fractions. Of course, negative exponents do not then cause errors anymore.

14.33 \xintDivision

\xintDivision{N}{M} returns {quotient Q}{remainder R}. This is euclidean division: N = QM + R, 0 ≤ R < |M|. So the remainder is always non-negative and the formula N = QM + R always holds independently of the signs of N or M. Division by zero is of course an error (even if N vanishes) and returns {0}{0}.

This macro is integer only (with xintfrac loaded it accepts fractions on input, but they must be integers in disguise) and not to be confused with the xintfrac macro \xintDiv which divides one fraction by another.

14.34 \xintQuo

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

14.35 \xintRem

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

14.36 \xintFDg

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

14.37 \xintLDg

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

14.38 \xintMON, \xintMMON

New in version 1.03.

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

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

14.39 \xintOdd

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

14.40 \xintDSL

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

14.41 \xintDSR

\xintDSR{N} is decimal shift right, *i.e.* it removes the last digit (keeping the sign). For a positive number, this is the same as the quotient from the euclidean division by ten (of course, done in a more efficient manner than via the general division algorithm). For N from -9 to -1, the macro returns 0.

14.42 \xintDSH

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

14.43 \xintDSHr, \xintDSx

New in release 1.01.

\xintDSHr{x}{N} expects x to be zero or positive and it returns then a value R which is correlated to the value Q returned by \xintDSH{x}{N} in the following manner:

- if N is positive or zero, Q and R are the quotient and remainder in the euclidean division by 10^x (obtained in a more efficient manner than using \xintDivision),

- if N is negative let Q1 and R1 be the quotient and remainder in the euclidean division by 10^x of the absolute value of N. If Q1 does not vanish, then Q=-Q1 and R=R1. If Q1 vanishes, then Q=0 and R=-R1.
- for x=0, Q=N and R=0.

So one has N = 10^x Q + R if Q turns out to be zero or positive, and N = 10^x Q - R if Q turns out to be negative, which is exactly the case when N is at most -10^x.

\xintDSx{x}{N} for x negative is exactly as \xintDSH{x}{N}, *i.e.* multiplication by 10^{-x}. For x zero or positive it returns the two numbers {Q}{R} described above, each one within braces. So Q is \xintDSH{x}{N}, and R is \xintDSHr{x}{N}, but computed simultaneously.

```
\xintAssign\xintDSx {-1}{-123456789}\to\M
\meaning\M: macro:->-1234567890.
\xintAssign\xintDSx {-20}{123456789}\to\M
\meaning\M: macro:->12345676890000000000000000000.
\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
```

14.44 \xintDecSplit

This has been modified in release 1.01.

\xintDecSplit{x}{N} cuts the number into two pieces (each one within a pair of enclosing braces). First the sign if present is *removed*. Then, for x positive or null, the second piece contains the x least significant digits (*empty* if x=0) and the first piece the remaining digits (*empty* when x equals or exceeds the length of N). Leading zeros in the second piece are not removed. When x is negative the first piece contains the |x| most significant digits and the second piece the remaining digits (*empty* if |x| equals or exceeds the length of N). Leading zeros in this second piece are not removed. So the absolute value of the original number is always the concatenation of the first and second piece.

This macro's behavior for N non-negative is final and will not change. I am still hesitant about what to do with the sign of a negative N.

```
            \xintAssign\xintDecSplit {0}{-123004321}\to\L\R
\meaning\L: macro:->123004321, \meaning\R: macro:->.
            \xintAssign\xintDecSplit {5}{-123004321}\to\L\R
\meaning\L: macro:->1230, \meaning\R: macro:->04321.
            \xintAssign\xintDecSplit {9}{-123004321}\to\L\R
```

```
\meaning\L: macro:->, \meaning\R: macro:->123004321.
            \xintAssign\xintDecSplit {10}{-123004321}\to\L\R
\meaning\L: macro:->, \meaning\R: macro:->123004321.
           \xintAssign\xintDecSplit {-5}{-12300004321}\to\L\R
\meaning\L: macro:->12300, \meaning\R: macro:->004321.
          \xintAssign\xintDecSplit {-11}{-12300004321}\to\L\R
\meaning\L: macro:->12300004321, \meaning\R: macro:->.
          \xintAssign\xintDecSplit {-15}{-12300004321}\to\L\R
\meaning\L: macro:->12300004321, \meaning\R: macro:->.
```

14.45 \xintDecSplitL

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

14.46 \xintDecSplitR

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

15 Commands of the xintgcd package

This package was included in the original release 1.0 of the xint bundle.

15.1 \xintGCD

\xintGCD{N}{M} computes the greatest common divisor. It is positive, except when both N and M vanish, in which case the macro returns zero.

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

15.2 \xintBezout

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

```
     \xintAssign {{\xintBezout {10000}{1113}}}\to\X
    \meaning\X: macro:->{10000}{1113}{-131}{-1177}{1}.
\xintAssign {\xintBezout {10000}{1113}}\to\A\B\U\V\D
\A: 10000, \B: 1113, \U: -131, \V: -1177, \D: 1.
\xintAssign {\xintBezout {123456789012345}{9876543210321}}\to\A\B\U\V\D
\A: 123456789012345, \B: 9876543210321, \U: 256654313730, \V: 3208178892607,
\D: 3.
```

15.3 \xintEuclideAlgorithm

\xintEuclideAlgorithm{N}{M} applies the Euclide algorithm and keeps a copy of all quotients and remainders.

```
\xintAssign {{\xintEuclideAlgorithm {10000}{1113}}}\to\X
```

`\meaning\X: macro:->{5}{10000}{1}{1113}{8}{1096}{1}{17}{64}{8}{2}{1}{8}{0}`. The first token is the number of steps, the second is `N`, the third is the GCD, the fourth is `M` then the first quotient and remainder, the second quotient and remainder, ... until the final quotient and last (zero) remainder.

15.4 \xintBezoutAlgorithm

`\xintBezoutAlgorithm{N}{M}` applies the Euclide algorithm and keeps a copy of all quotients and remainders. Furthermore it computes the entries of the successive products of the 2 by 2 matrices $\begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}$ formed from the quotients arising in the algorithm.

```
\xintAssign {{\xintEuclideAlgorithm {10000}{1113}}}\to\X
\meaning\X: macro:->{5}{10000}{0}{1}{1}{1113}{1}{0}{8}{1096}{8}{1}{1}{17}{9}{1}{64}{8}{584}{65}{2}{1}{1177}{131}{8}{0}{10000}{1113}.
```

The first token is the number of steps, the second is `N`, then `0`, `1`, the GCD, `M`, `1`, `0`, the first quotient, the first remainder, the top left entry of the first matrix, the bottom left entry, and then these four things at each step until the end.

15.5 \xintTypesetEuclideAlgorithm

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

```
\xintTypesetEuclideAlgorithm {123456789012345}{9876543210321}
```

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

15.6 \xintTypesetBezoutAlgorithm

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

```
\xintTypesetBezoutAlgorithm {10000}{1113}
```

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

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

16 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. As in the previous documentation, x stands for something which will be internally embedded in a \numexpr, thus completely expanded and then must deliver a number obeying the TeX bounds. It may be a count register or something like 4*\count 255 + 17, etc...

f stands for a fraction (or a possibly 'long' integer), or something which expands to a fraction or a possibly long integer. See the earlier section on fraction formats.

16.1 \xintLen

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

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

16.2 \xintRaw

New with release 1.04.

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}.

```
\xintRaw{\the\numexpr 571*987\relax.123/\the\numexpr -201+59\relax}=
                          -563577123/142000
```

16.3 \xintNumerator

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

```
\xintNumerator {178000/25600000[17]}=178000000000000000000
       \xintNumerator {312.289001/20198.27}=312289001
          \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.

16.4 \xintDenominator

This returns the denominator corresponding to the internal representation of the fraction:[21]

```
\xintDenominator {178000/25600000[17]}=25600000
\xintDenominator {312.289001/20198.27}=20198270000
\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`.

16.5 \xintFrac

This is a LaTeX only command, to be used in math mode only. It will print a fraction, internally represented as something equivalent to `A/B[n]` as `\frac {A}{B}10^n`. The power of ten is omitted when `n=0`, the denominator is omitted when it has value one, the number being separated from the power of ten by a `\cdot`. `$\xintFrac {178.000/25600000}$` gives $\frac{178000}{25600000}10^{-3}$, `$\xintFrac {178.000/1}$` gives $178000\cdot 10^{-3}$, `$\xintFrac {3.5/5.7}$` gives $\frac{35}{57}$, and `$\xintFrac {\xintIrr {\xintFac{10}/\xintiSqr{\xintFac {5}}}}$` gives 252. As shown by the examples, simplification of the input (apart from removing the decimal points and moving the minus sign to the numerator) is not done automatically and must be the result of macros such as `\xintIrr` or `\xintREZ`.

16.6 \xintSignedFrac

New with release `1.04`.

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

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

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

16.7 \xintFwOver

This does the same as `\xintFrac` except that the `\over` primitive is used for the fraction (in case the denominator is not one; and a pair of braces contains the `A\over B` part). `$\xintFwOver {178.000/25600000}$` gives $\frac{178000}{25600000}10^{-3}$, `$\xintFwOver {178.000/1}$` gives $178000\cdot 10^{-3}$, `$\xintFwOver {3.5/5.7}$` gives $\frac{35}{57}$, and `$\xintFwOver {\xintIrr {\xintFac{10}/\xintiSqr{\xintFac {5}}}}$` gives 252.

16.8 \xintSignedFwOver

New with release `1.04`.

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

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

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

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

16.9 \xintREZ

This command normalizes a fraction by removing the powers of ten in its numerator and denominator: `\xintREZ {178000/25600000[17]}=178/256[15]`. As shown by the example, it does not otherwise simplify the fraction.

16.10 \xintIrr

This puts the fraction into its unique irreducible form:

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

Note that the current implementation does not cleverly first factor powers of 2 and 5, so input such as `\xintIrr {2/3[100]}` will make xintfrac do the Euclidean division of `2·10^{100}` by 3, which is a bit stupid.

16.11 \xintJrr

This also puts the fraction into its unique irreducible form:

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

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

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

But to notice the difference one would need computations with much bigger numbers than in this example.

16.12 \xintTrunc

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

```
\xintTrunc {16}{-803.2028/20905.298}=-0.0384210165289200
\xintTrunc {20}{-803.2028/20905.298}=-0.03842101652892008523
\xintTrunc {10}{\xintPow {-11}{-11}}=-0.0000000000
\xintTrunc {12}{\xintPow {-11}{-11}}=-0.000000000003
\xintTrunc {12}{\xintAdd {-1/3}{3/9}}=0
```

The digits printed are exact up to and including the last one. The identity `\xintTrunc {x}{-f}=-\xintTrunc {x}{f}` holds.[22]

[22] this is just a notation; currently `-\macro` is not valid input to any package macro, one must use `\xintOpp{\macro}` or `\xintiOpp{\macro}`.

16.13 \xintiTrunc

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

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

Differences between \xintTrunc{0}{f} and \xintiTrunc{0}{f}: the former cannot be used inside integer-only macros, and the latter removes the decimal point, and never returns -0 (and of course removes all superfluous leading zeros.)

16.14 \xintRound

New with release 1.04.

\xintRound{x}{f} returns the start of the decimal expansion of the fraction f, rounded to x digits precision after the decimal point. The argument x should be non-negative. Only when f evaluates exactly to zero does \xintRound return 0 without decimal point. When f is not zero, its sign is given in the output, also when the digits printed are all zero.

```
\xintRound {16}{-803.2028/20905.298}=-0.0384210165289201
\xintRound {20}{-803.2028/20905.298}=-0.03842101652892008523
\xintRound {10}{\xintPow {-11}{-11}}=-0.0000000000
\xintRound {12}{\xintPow {-11}{-11}}=-0.000000000004
\xintRound {12}{\xintAdd {-1/3}{3/9}}=0
```

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

-0.00000000000350493899481392497604003313162598556370...

16.15 \xintiRound

New with release 1.04.

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

```
\xintiRound {16}{-803.2028/20905.298}=-384210165289201
\xintiRound {10}{\xintPow {-11}{-11}}=0
```

Differences between \xintRound{0}{f} and \xintiRound{0}{f}: the former cannot be used inside integer-only macros, and the latter removes the decimal point, and never returns -0 (and of course removes all superfluous leading zeros.)

16.16 \xintAdd

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

16.17 \xintSub

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

16.18 \xintMul

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

16.19 \xintSqr

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

16.20 \xintPow

The original macro is extended to accept a fraction on input (the exponent must be a signed integer of course). Its output will now always be in the form A/B[n]. The original is available as \xintiPow.

16.21 \xintSum, \xintSumExpr

The original commands are extended to accept fractions on input and produce fractions on output. Their outputs will now always be in the form A/B[n]. The originals are available as \xintiSum and \xintiSumExpr.

16.22 \xintPrd, \xintPrdExpr

The originals are extended to accept fractions on input and produce fractions on output. Their outputs will now always be in the form A/B[n]. The originals are available as \xintiPrd and \xintiPrdExpr.

16.23 \xintDiv

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

16.24 \xintCmp

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

16.25 \xintMax

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

16.26 \xintMin

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

16.27 \xintAbs

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

16.28 \xintSgn

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

16.29 \xintOpp

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

16.30 \xintGeq, \xintDivision, \xintQuo, \xintRem, \xintFDg, \xintLDg, \xintMON, \xintMMON

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

16.31 \xintNum

The macro is extended to accept a fraction on input. But this fraction should reduce to an integer. If not an error will be raised. The original is available as \xintiNum.

17 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 fully expanded, they may be count registers, etc...

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

17.1 \xintSeries

\xintSeries{A}{B}{\coeff} evaluates the sum of all values of the \coeff {n} from n=A to and including n=B. The initial and final indices must obey the \numexpr constraint of expanding to numbers at most 2^31-1. The \coeff macro (which, as argument to \xintSeries is expanded only at the time of computing the successive \coeff {n}) should be defined as a one-parameter fully expandable command, providing its output from an input being an explicit number (string of digits, no need to make proviso for a count register).

```
\def\coeff #1{\xintiMON{#1}/#1.5}     % (-1)^n/(n+1/2)
\edef\w {\xintSeries {0}{50}{\coeff}} % we want to re-use it
```

```
\edef\z {\xintJrr {\w}[0]}            % the [0] for a microsecond gain.
% \xintJrr preferred to \xintIrr: a big common factor is suspected.
% But numbers much bigger would be needed to show the greater efficiency.
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} = \xintFrac\z \]
```

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}} = \frac{17390933828737094043211279210162660227871 4}{11002746715939000302527991722603972905057 5}$$

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

17.2 \xintiSeries

`\xintiSeries{A}{B}{\coeff}` evaluates the sum of `\coeff {n}` from `n=A` to and including `n=B`. The initial and final indices are given to a `\numexpr` expression. The `\coeff` macro (which, as argument to `\xintiSeries` is expanded only at the time of computing `\coeff {n}`) should be defined as a one-parameter fully expandable command, accepting on input an explicit number, and returning a (long) integer in the format understood by the integer-only `\xintiAdd`.

```
\def\coeff #1{\xintiTrunc {40}{\xintMON{#1}/#1.5}}%
% better:
\def\coeff #1{\xintiTrunc {40}
   {\the\numexpr 2*\xintiMON{#1}\relax/\the\numexpr 2*#1+1\relax [0]}}%
% better still:
\def\coeff #1{\xintiTrunc {40}
 {\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}}%
% (-1)^n/(n+1/2) times 10^40, truncated to an integer.
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} \approx
        \xintTrunc {40}{\xintiSeries {0}{50}{\coeff}[-40]}\dots\]
```

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

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}} \approx 1.58059930649352504123678950695672641448 10$$

We should have cut out at least the last two digits: truncating errors originating with the first coefficients of the sum will never go away, and each truncation introduces an uncertainty in the last digit, so as we have 40 terms, we should trash the last two digits, or at least round at 38 digits. It is interesting to compare with the computation where rounding rather than truncation is used, and with the decimal expansion of the exactly computed partial sum of the series:

```
\def\coeff #1{\xintiRound {40} % rounding at 40
  {\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}}%
% (-1)^n/(n+1/2) times 10^40, rounded to an integer.
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} \approx
        \xintTrunc {40}{\xintiSeries {0}{50}{\coeff}[-40]}\]
\def\exactcoeff #1%
  {\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}%
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12}
   = \xintTrunc {50}{\xintSeries {0}{50}{\exactcoeff}}\dots\]
```

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}} \approx 1.58059930649352504123678950695672641448 04\,68680288 367$$

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}} = 1.58059930649352504123678950695672641448 06\,86802883 67\ldots$$

This shows indeed that our sum of truncated terms estimated wrongly the 39th and 40th digits of the exact result[23] and that the sum of rounded terms fared a bit better.

17.3 \xintRationalSeries

New with release 1.04.

`\xintRationalSeries{A}{B}{f}{\ratio}` evaluates the sum of `F(n)`[24] from `n=A` up to and including `n=B`, with the parameter `f` being (or expanding to) the value `F(A)` and `\ratio` being a one-parameter expandable command, accepting on input an explicit number `n` and producing after (full iterated) expansion (of the first token) `F(n)/F(n-1)`. The initial and final indices are given to a `\numexpr` expression.

```
\def\ratio #1{2/#1[0]}% 2/n, comes from the series of exp(2)
\cnta 0 % previously declared count
\loop
\edef\z {\xintRationalSeries {0}{\cnta}{1}{\ratio }}%
\noindent$\sum_{n=0}^{\the\cnta} \frac{2^n}{n!}=
          \xintTrunc{12}\z\dots=
          \xintFrac\z=\xintFrac{\xintIrr\z}$\vtop to 5pt{}\endgraf
\ifnum\cnta<20 \advance\cnta 1 \repeat
```

$\sum_{n=0}^{0} \frac{2^n}{n!} = 1.000000000000\cdots = 1 = 1$
$\sum_{n=0}^{1} \frac{2^n}{n!} = 3.000000000000\cdots = 3 = 3$
$\sum_{n=0}^{2} \frac{2^n}{n!} = 5.000000000000\cdots = \frac{10}{2} = 5$
$\sum_{n=0}^{3} \frac{2^n}{n!} = 6.333333333333\cdots = \frac{38}{6} = \frac{19}{3}$
$\sum_{n=0}^{4} \frac{2^n}{n!} = 7.000000000000\cdots = \frac{168}{24} = 7$
$\sum_{n=0}^{5} \frac{2^n}{n!} = 7.266666666666\cdots = \frac{872}{120} = \frac{109}{15}$
$\sum_{n=0}^{6} \frac{2^n}{n!} = 7.355555555555\cdots = \frac{5296}{720} = \frac{331}{45}$
$\sum_{n=0}^{7} \frac{2^n}{n!} = 7.380952380952\cdots = \frac{37200}{5040} = \frac{155}{21}$
$\sum_{n=0}^{8} \frac{2^n}{n!} = 7.387301587301\cdots = \frac{297856}{40320} = \frac{2327}{315}$
$\sum_{n=0}^{9} \frac{2^n}{n!} = 7.388712522045\cdots = \frac{2681216}{362880} = \frac{20947}{2835}$
$\sum_{n=0}^{10} \frac{2^n}{n!} = 7.388994708994\cdots = \frac{26813184}{3628800} = \frac{34913}{4725}$
$\sum_{n=0}^{11} \frac{2^n}{n!} = 7.389046015712\cdots = \frac{294947072}{39916800} = \frac{164591}{22275}$
$\sum_{n=0}^{12} \frac{2^n}{n!} = 7.389054566832\cdots = \frac{3539368960}{479001600} = \frac{691283}{93555}$
$\sum_{n=0}^{13} \frac{2^n}{n!} = 7.389055882389\cdots = \frac{46011804672}{6227020800} = \frac{14977801}{2027025}$
$\sum_{n=0}^{14} \frac{2^n}{n!} = 7.389056070325\cdots = \frac{644165281792}{87178291200} = \frac{314533829}{42567525}$
$\sum_{n=0}^{15} \frac{2^n}{n!} = 7.389056095384\cdots = \frac{9662479259648}{1307674368000} = \frac{4718007451}{638512875}$
$\sum_{n=0}^{16} \frac{2^n}{n!} = 7.389056098516\cdots = \frac{154599668219904}{20922789888000} = \frac{1572669151}{212837625}$
$\sum_{n=0}^{17} \frac{2^n}{n!} = 7.389056098884\cdots = \frac{2628194359869440}{355687428096000} = \frac{16041225341}{2170943775}$
$\sum_{n=0}^{18} \frac{2^n}{n!} = 7.389056098925\cdots = \frac{47307498477912064}{6402373705728000} = \frac{103122162907}{13956067125}$
$\sum_{n=0}^{19} \frac{2^n}{n!} = 7.389056098930\cdots = \frac{898842471080853504}{121645100408832000} = \frac{4571749222213}{618718975875}$
$\sum_{n=0}^{20} \frac{2^n}{n!} = 7.389056098930\cdots = \frac{17976849421618118656}{2432902008176640000} = \frac{68576238333199}{9280784638125}$

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

[24] the macro is designed to be useful when `F(n)/F(n-1)` is a rational function of `n` but it may be used of course with any sort of general term.

Such computations would become quickly completely inaccessible via the \xintSeries macros, as the factorials in the denominators would get all multiplied together: the raw addition and subtraction on fractions just blindly multiplies denominators! Whereas \xintRationalSeries evaluate the partial sums via a less silly iterative scheme.

```
\def\ratio #1{-1/#1[0]}% -1/n, comes from the series of exp(-1)
\cnta 0 % previously declared count
\loop
\edef\z {\xintRationalSeries {0}{\cnta}{1}{\ratio }}%
\noindent$\sum_{n=0}^{\the\cnta} \frac{(-1)^n}{n!}=
           \xintTrunc{20}\z\dots=\xintFrac{\z}=\xintFrac{\xintIrr\z}$
         \vtop to 5pt{}\endgraf
\ifnum\cnta<20 \advance\cnta 1 \repeat
```

$\sum_{n=0}^{0} \frac{(-1)^n}{n!} = 1.00000000000000000000\cdots = 1 = 1$
$\sum_{n=0}^{1} \frac{(-1)^n}{n!} = 0\cdots = 0 = 0$
$\sum_{n=0}^{2} \frac{(-1)^n}{n!} = 0.50000000000000000000\cdots = \frac{1}{2} = \frac{1}{2}$
$\sum_{n=0}^{3} \frac{(-1)^n}{n!} = 0.33333333333333333333\cdots = \frac{2}{6} = \frac{1}{3}$
$\sum_{n=0}^{4} \frac{(-1)^n}{n!} = 0.37500000000000000000\cdots = \frac{9}{24} = \frac{3}{8}$
$\sum_{n=0}^{5} \frac{(-1)^n}{n!} = 0.36666666666666666666\cdots = \frac{44}{120} = \frac{11}{30}$
$\sum_{n=0}^{6} \frac{(-1)^n}{n!} = 0.36805555555555555555\cdots = \frac{265}{720} = \frac{53}{144}$
$\sum_{n=0}^{7} \frac{(-1)^n}{n!} = 0.36785714285714285714\cdots = \frac{1854}{5040} = \frac{103}{280}$
$\sum_{n=0}^{8} \frac{(-1)^n}{n!} = 0.36788194444444444444\cdots = \frac{14833}{40320} = \frac{2119}{5760}$
$\sum_{n=0}^{9} \frac{(-1)^n}{n!} = 0.36787918871252204585\cdots = \frac{133496}{362880} = \frac{16687}{45360}$
$\sum_{n=0}^{10} \frac{(-1)^n}{n!} = 0.36787946428571428571\cdots = \frac{1334961}{3628800} = \frac{16481}{44800}$
$\sum_{n=0}^{11} \frac{(-1)^n}{n!} = 0.36787943923360590027\cdots = \frac{14684570}{39916800} = \frac{1468457}{3991680}$
$\sum_{n=0}^{12} \frac{(-1)^n}{n!} = 0.36787944132128159905\cdots = \frac{176214841}{479001600} = \frac{16019531}{43545600}$
$\sum_{n=0}^{13} \frac{(-1)^n}{n!} = 0.36787944116069116069\cdots = \frac{2290792932}{6227020800} = \frac{63633137}{172972800}$
$\sum_{n=0}^{14} \frac{(-1)^n}{n!} = 0.36787944117216190628\cdots = \frac{32071101049}{87178291200} = \frac{2467007773}{6706022400}$
$\sum_{n=0}^{15} \frac{(-1)^n}{n!} = 0.36787944117139718991\cdots = \frac{481066515734}{1307674368000} = \frac{34361893981}{93405312000}$
$\sum_{n=0}^{16} \frac{(-1)^n}{n!} = 0.36787944117144498468\cdots = \frac{7697064251745}{20922789888000} = \frac{15549624751}{42268262400}$
$\sum_{n=0}^{17} \frac{(-1)^n}{n!} = 0.36787944117144217323\cdots = \frac{130850092279664}{355687428096000} = \frac{8178130767479}{22230464256000}$
$\sum_{n=0}^{18} \frac{(-1)^n}{n!} = 0.36787944117144232942\cdots = \frac{2355301661033953}{6402373705728000} = \frac{138547156531409}{376610217984000}$
$\sum_{n=0}^{19} \frac{(-1)^n}{n!} = 0.36787944117144232120\cdots = \frac{44750731559645106}{121645100408832000} = \frac{92079694567171}{250298560512000}$
$\sum_{n=0}^{20} \frac{(-1)^n}{n!} = 0.36787944117144232161\cdots = \frac{895014631192902121}{2432902008176640000} = \frac{4282366656425369}{11640679464960000}$

We can incorporate an indeterminate if we define \ratio to be a macro with two parameters: \def\ratioexp #1#2{\xintDiv{#1}{#2}}% x/n: x=#1, n=#2. Then, if \x expands to some fraction x, the command

```
\xintRationalSeries {0}{b}{1}{\ratioexp{\x}}
```

will compute $\sum_{n=0}^{n=b} x^n/n!$:

```
\cnta 0
\def\ratioexp #1#2{\xintDiv{#1}{#2}}% #1/#2
\loop
\noindent
```

```
$\sum_{n=0}^{\the\cnta} (.57)^n/n! = \xintTrunc {50}
     {\xintRationalSeries {0}{\cnta}{1}{\ratioexp{.57}}}\dots$
     \vtop to 5pt {}\endgraf
\ifnum\cnta<50 \advance\cnta 10 \repeat
```

$\sum_{n=0}^{0}(.57)^n/n! = 1.00000000000000000000000000000000000000000000000000\dots$
$\sum_{n=0}^{10}(.57)^n/n! = 1.76826705137947002480668058035714285714285714285714\dots$
$\sum_{n=0}^{20}(.57)^n/n! = 1.76826705143373515162089324271187082272833005529082\dots$
$\sum_{n=0}^{30}(.57)^n/n! = 1.76826705143373515162089339282382144915484884979430\dots$
$\sum_{n=0}^{40}(.57)^n/n! = 1.76826705143373515162089339282382144915485219867776\dots$
$\sum_{n=0}^{50}(.57)^n/n! = 1.76826705143373515162089339282382144915485219867776\dots$

Observe that in this last example the `x` was directly inserted; if it had been a more complicated explicit fraction it would have been worthwile to use `\ratioexp\x` with `\x` defined to expand to its value. In the further situation where this fraction `x` is not explicit but itself defined via a complicated, and time-costly, formula, it should be noted that `\xintRationalSeries` will do again the evaluation of `\x` for each term of the partial sum. The easiest is thus when `x` can be defined as an `\edef`. If however, you are in an expandable-only context and cannot store in a macro like `\x` the value to be used, a variant of `\xintRationalSeries` is needed which will first evaluate this `\x` and then use this result without recomputing it. This is `\xintRationalSeriesX`, documented next.

Here is a slightly more complicated evaluation:

```
\cnta 1
\loop \edef\z {\xintRationalSeries
                    {\cnta}
                    {2*\cnta-1}
                    {\xintiPow {\the\cnta}{\cnta}/\xintFac{\cnta}}
                    {\ratioexp{\the\cnta}}}%
\edef\w {\xintRationalSeries {0}{2*\cnta-1}{1}{\ratioexp{\the\cnta}}}%
\noindent
$\sum_{n=\the\cnta}^{\the\numexpr 2*\cnta-1\relax} \frac{\the\cnta^n}{n!}/%
          \sum_{n=0}^{\the\numexpr 2*\cnta-1\relax} \frac{\the\cnta^n}{n!} =
          \xintTrunc{8}{\xintDiv\z\w}\dots$ \vtop to 5pt{}\endgraf
\ifnum\cnta<20 \advance\cnta 1 \repeat
```

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

17.4 \xintRationalSeriesX

New with release 1.04.

\xintRationalSeriesX{A}{B}{\first}{\ratio}{\x} evaluates the sum of F(n,x) from n=A up to and including n=B, where \x expands to a fraction x, \first is a one-parameter macro such that \first{\x} expands in two steps at most to the first term F(A,x) of the series, and \ratio is a two parameter macro such that \ratio{\x}{n} expands to the ratio F(n,x)/F(n-1,x). Hence, this is a parametrized version of \xintRationalSeries, where the parameter \x is evaluated only once at the beginning of the computation, and can thus itself be the yet unevaluated result of a previous computation.

Note the subtle differences between

```
\xintRationalSeries {a}{b}{\first}{\ratio{\x}}
\xintRationalSeriesX {a}{b}{\first}{\ratio}{\x}
```

First the location of braces differ... then, in the first one \first is a macro expanding to a fractional number, but in the X one, it is a one-parameter macro which will use \x. The \ratio macro is in both cases a two-parameters macro, the difference is that in the X variant the \x will be evaluated at the very beginning whereas the former variant replaces it by its evaluation each time it needs it (which is bad if this evaluation is time-costly, but good if it just a big explicit fraction encapsulated in a macro).

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

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

1.099999999999083906...	1.499954310225476533...	1.870485649686617459...
1.199999998111624029...	1.599659266069210466...	1.907197560339468199...
1.299999835744121464...	1.698137473697423757...	1.845117565491393752...
1.399996091955359088...	1.791898112718884531...	1.593831932293536053...

These completely exact operations rapidly create numbers with many digits. Let us print in full the raw fractions created by the operation illustrated above:

E(L(1[-1]))=435534952734304993753128478305695755446525998418916420656
308534427154141471013807206588202981046013155342233701289165089056830056
936564478988779520000000000/395940866122425193243875570782668457763038822
4000000000000000000[-90] (length of numerator: 155)

`E(L(12[-2]))`=4434537700544174654421092523472648247118935991604117296038825841980841532261080707075058900962803059710371332802034641237155887714188380658982959014134632946402759999397422009303463626532643541704863984316744555312271367954598414044364800000000/39594086612242519324387557078266845776303882240000000000000000000000[-180] (length of numerator: 245)

`E(L(123[-3]))`=4446415926519417771542541488488548661989549715526163900742959135317921138508647797623508008144169817627741486630524932175667597540979774207315163733367897227307654961390791852295451022482823911996210292377938117401221109197354331611327571689558640177108818505853950798598438316179662071953915678034718321474363029365556301004800000000/3959408661224251932438755707826684577630388224000000000000000000000[-270] (length of numerator: 335)

We see that the denominators here remain the same, as our input only had various powers of ten as denominators, and **xintfrac** efficiently assemble (some only, as we can see) powers of ten. Notice that 1 more digit in an input denominator seems to mean 90 more in the raw output. We can check that with some other test cases:

`E(L(1/7))`=5181385161173226049160748331648333448838405901330061681251253466743091335325539480471366915857159004497689259144894523418643519242240000000000/45337120162108979178809662782137765289223265381758152546654836095087089601022689942796465342115407786358809263904208715776000000000000000000[0] (length of numerator: 141; length of denominator: 141)

`E(L(1/71))`=164799489177219556498025955806107098256158101756209369864657152282149780083067798039175325186850716609293467854603842163754716919123274624394132188208895310089982001627351524910000588238596565380887916286153347403881434316800000000/16251060738309150710228315926583043448560635097998286551792304600401711584442548604911127392639471285026166742651015948354491747514663603304596379819982611548681495538153647264137927630891689041426777132144944742400000000000000000000[0] (length of numerator: 232; length of denominator: 232)

`E(L(1/712))`=2096231738801631206754816378972162002839689022482032389431369022641828655597172664063419763257670013571094529806073912714380791950739593015282540060879081568881295675202690117154599691546887990896257382714338565353779187008849807986411970218551170786297803168353530430674157534972120128999850190174947982205517824000000000/2093291722337673799732719862311619975662927884547744846526034295741465968177583093786412050480958301357075221213896546903011983961080605724903426024563430558292203346913309844190901402018394162270065876670575550330002721292096217682473000829618103432600036119035084894266166648343032219206471638591733760000000000000000000[0] (length of numerator: 322; length of denominator: 322)

For info the last fraction put into irreducible form still has 288 digits in its denominator.[25] The first conclusion is that 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. The second conclusion is that

[25] putting this fraction in irreducible form takes more time than is typical of the other computations in this document; so exceptionally I have hard-coded the 288 in the document source.

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 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. Floating point representation of numbers is currently unimplemented in xint. But fixed point computations are available via the commands \xintTrunc and \xintRound.

17.5 \xintPowerSeries

\xintPowerSeries{A}{B}{\coeff}{x} evaluates the sum of \coeff{n}·x^n from n=A up to and including n=B. 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 (in the now usual meaning) command, accepting on input an explicit number.

The x can be either a fraction directly input or a macro expanding to such a fraction. It is actually more efficient to encapsulate an explicit fraction x in such a macro (say \x), 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). With release 1.04 the Horner scheme for polynomial evaluation is used, this avoids a denominator build-up which was plaguing the 1.03 version. [26]

> 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\x {5/17[0]}
\[ \sum_{n=0}^{n=20} \Bigl(\frac 5{17}\Bigr)^n
 =\xintFrac{\xintIrr{\xintPowerSeries {0}{20}{\geom}{\x}}}
 =\xintFrac{\xintiSub{\xintiPow {17}{21}}{\xintiPow{5}{21}}%
                    /\xintiMul{12}{\xintiPow {17}{20}}}\]
% a parser for arbitrary algebraic expressions with the +,-,/,*,and ^
```

[26] with powers x^k, from k=0 to N, a denominator d of x became d^{1+2+...+N}, which is bad. With the 1.04 method, the part of the denominator originating from x does not accumulate to more than d^N.

```
% operations would be dearly appreciated here ; implementing a completely
% expandable one would be quite a lot of work, even if we plagiarize l3fp!
```

$$\sum_{n=0}^{n=20}\Big(\frac{5}{17}\Big)^n = \frac{5757661159377657976885341}{406423140664757252240 1601} = \frac{690919339125318957226240 92}{48770776879770870268819212}$$

```
\def\coefflog #1{1/#1[0]}% 1/n
\def\x {1/2[0]}%
\[ \log 2 \approx \sum_{n=1}^{20} \frac1{n\cdot 2^n}
    = \xintFrac {\xintIrr {\xintPowerSeries {1}{20}{\coefflog}{\x}}}\]
\[ \log 2 \approx \sum_{n=1}^{50} \frac1{n\cdot 2^n}
    = \xintFrac {\xintIrr {\xintPowerSeries {1}{50}{\coefflog}{\x}}}\]
```

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

$$\log 2 \approx \sum_{n=1}^{50} \frac{1}{n\cdot 2^n} = \frac{6046346975175226566357988455973921 9}{8723034796579283922394620817833984 0}$$

```
\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}{\x}}\dots
\endgraf
\ifnum \cnta < 30 \advance\cnta 1 \repeat
```

1. 0.500000000000...
2. 0.625000000000...
3. 0.666666666666...
4. 0.682291666666...
5. 0.688541666666...
6. 0.691145833333...
7. 0.692261904761...
8. 0.692750186011...
9. 0.692967199900...
10. 0.693064856150...
11. 0.693109245355...
12. 0.693129590407...
13. 0.693138980431...
14. 0.693143340085...
15. 0.693145374590...
16. 0.693146328265...
17. 0.693146777052...
18. 0.693146988980...
19. 0.693147089367...
20. 0.693147137051...
21. 0.693147159757...
22. 0.693147170594...
23. 0.693147175777...
24. 0.693147178261...
25. 0.693147179453...
26. 0.693147180026...
27. 0.693147180302...
28. 0.693147180435...
29. 0.693147180499...
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\x {1/25[0]}% 1/5^2
\[\mathrm{Arctg}(\frac15)\approx
```

```
            \frac15\sum_{n=0}^{15} \frac{(-1)^n}{(2n+1)25^n}
= \xintFrac{\xintIrr {\xintDiv
            {\xintPowerSeries {0}{15}{\coeffarctg}{\x}}{5}}}\]
```

$$\mathrm{Arctg}(\frac{1}{5}) \approx \frac{1}{5}\sum_{n=0}^{15}\frac{(-1)^n}{(2n+1)25^n} = \frac{1659187265191229558953917932691 68}{8405393041530624032020568847656 25}$$

17.6 \xintPowerSeriesX

New with release 1.04.

This is the same as \xintPowerSeries apart from the fact that the last parameter (aka x), is first expanded before being then used. If the x parameter is to be an explicit big fraction f with many (dozens) digits, rather than using f directly it is slightly better to have some macro \x \def'ined to expand to the explicit f and use \xintPowerSeries; but if f has not yet been evaluated and will be the output of a complicated expansion of some \x, and if, due to an expanding only context, an \edef\z{\x} is no option, then \xintPowerSeriesX should be used with \x as last parameter. This \x will be expanded (as usual) and then its (explicit) output will be used. The reason why \xintPowerSeries doesn't do the same is that explicit fractions with many (dozens) digits slow down a bit the processing as there is some shuffling of tokens going on. With \xintPowerSeriesX the slowing down in token shuffling due to a very big fraction will not be avoided, but the far worse cost of re-doing each time the computations leading to such a fraction will be. The constraints of expandability make it impossible to encapsulate the result of this initial computation in a macro and have the best of both worlds.

```
\def\ratioexp  #1#2{\xintDiv {#1}{#2}}% x/n
% These are the (-1)^{n-1}/n of the log(1+h) series:
\def\coefflog #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}%
% Let L(h) be the first 10 terms of the log(1+h) series and
% let E(t) be the first 10 terms of the exp(t) series.
% The following computes L(E(a/10)-1) for a=1,..., 12.
\cnta 1
\loop
\noindent\xintTrunc {18}{%
   \xintPowerSeriesX {1}{10}{\coefflog}
  {\xintSub
      {\xintRationalSeries {0}{9}{1[0]}{\ratioexp{\the\cnta[-1]}}}
      {1}}}\dots
\endgraf
\ifnum\cnta < 12 \advance \cnta 1 \repeat
```

0.099999999998556159...	0.499511320760604148...	-1.597091692317639401...
0.199999995263443554...	0.593980619762352217...	-12.648937932093322763...
0.299999338075041781...	0.645144282733914916...	-66.259639046914679687...
0.399974460740121112...	0.398118280111436442...	-304.768437445462801227...

17.7 \xintFxPtPowerSeries

\xintFxPtPowerSeries{A}{B}{\coeff}{x}{D} computes the sum of \coeff{n}·x^n from n=A to n=B 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 only expanded (in the usual meaning) when it is used inside the computations. Idem for x. If x itself is some complicated macro it is thus better to use the variant \xintFxPtPowerSeriesX which expands it first and then uses the result of that expansion.

The current (1.04) implementation is: the first power x^A is computed exactly, then *truncated*. Then each successive power is obtained from the previous one by multiplication by the exact value of x, and truncated. And \coeff{n}·x^n is obtained from that by multiplying by \coeff{n} (untruncated) and then truncating. Finally the sum is computed exactly. Apart from that \xintFxPtPowerSeries (where FxPt means 'fixed-point') is like \xintPowerSeries.

There should be a variant for things of the type $\sum c_n \frac{x^n}{n!}$ to avoid having to compute the factorial from scratch at each coefficient, the same way \xintFxPtPowerSeries does not compute x^n from scratch at each n. Perhaps in the next package release.

$$e^{-\frac{1}{2}} \approx$$

1.00000000000000000000	0.60653056795634920635	0.60653065971263344622
0.50000000000000000000	0.60653066483754960317	0.60653065971263342289
0.62500000000000000000	0.60653065945526069224	0.60653065971263342361
0.60416666666666666667	0.60653065972437513778	0.60653065971263342359
0.60677083333333333333	0.60653065971214266299	0.60653065971263342359
0.60651041666666666667	0.60653065971265234943	0.60653065971263342359
0.60653211805555555555	0.60653065971263274611	

```
\def\coeffexp #1{1/\xintFac {#1}[0]}% 1/n!
\def\x {-1/2[0]}% [0] for faster input parsing
\def\ApproxExp #1#2{\xintFxPtPowerSeries {0}{#1}{\coeffexp}{\x}{#2}}%
\cnta 0 % previously declared \count register
\noindent\loop
$\ApproxExp {\cnta}{20}$\\ % truncates 20 digits after decimal point
\ifnum\cnta<19 \advance\cnta 1 \repeat\par
% One should **not** trust the final digits, as the potential truncation
% errors of up to 10^{-20} per term accumulate and never disappear! (the
% effect is attenuated by the alternating signs in the series). We  can
% confirm that the last two digits (of our evaluation of the nineteenth
% partial sum) are wrong via the evaluation with more digits:

\xintFxPtPowerSeries {0}{19}{\coeffexp}{\x}{25}= 0.6065306597126334236037992
```

It is no difficulty for xintfrac to compute exactly, with the help of \xintPowerSeries, the nineteenth partial sum, and to then give (the start of) its exact decimal expansion:

$$\texttt{\xintPowerSeries \{0\}\{19\}\{\coeffexp\}\{\x} = \frac{38682746160036397317757}{63777066403145711616000}$$
$$= 0.60653065971263342360379915212 6\ldots$$

Thus, one should always estimate a priori how many ending digits are not reliable: if there are N terms and N has k digits, then digits up to but excluding the last k may usually be trusted. If we are optimistic and the series is alternating we may even replace N with $\sqrt{N}$ to get the number k of digits possibly of dubious significance.

17.8 \xintFxPtPowerSeriesX

New with release 1.04.

\xintFxPtPowerSeriesX{A}{B}{\coeff}{\x}{D} computes, exactly as \xintFxPtPowerSeries, the sum of \coeff{n}·\x^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 \x is first expanded and it is the result of this which is used in the computations.

Let us illustrate this on the numerical exploration of the identity

log(1+x) = -log(1/(1+x))

Let L(h)=log(1+h), and D(h)=L(h)+L(-h/(1+h)). Theoretically thus, D(h)=0 but we shall evaluate L(h) and -h/(1+h) keeping only 10 terms of their respective series. We will assume |h|<0.5. With only ten terms kept in the power series we do not have quite 3 digits precision as 2^10=1024. So it wouldn't make sense to evaluate things more precisely than, say circa 5 digits after the decimal points.

```
\cnta 0
\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}% (-1)^{n-
1}/n
\def\coeffalt #1{\the\numexpr\ifodd#1 -1\else1\fi\relax [0]}%   (-1)^n
\loop
\noindent \hbox to 2.5cm {\hss\texttt{D(\the\cnta/100): }}%
\xintAdd {\xintFxPtPowerSeriesX {1}{10}{\coefflog}{\the\cnta [-2]}{5}}
         {\xintFxPtPowerSeriesX {1}{10}{\coefflog}
             {\xintFxPtPowerSeriesX {1}{10}{\coeffalt}{\the\cnta [-2]}{5}}
          {5}}\endgraf
\ifnum\cnta < 49 \advance\cnta 7 \repeat
```

D(0/100): 0/1[0] D(28/100): 4/1[-5]
D(7/100): 2/1[-5] D(35/100): 4/1[-5]
D(14/100): 2/1[-5] D(42/100): 9/1[-5]
D(21/100): 3/1[-5] D(49/100): 42/1[-5]

Let's say we evaluate functions on [-1/2,+1/2] with values more or less also in [-1/2,+1/2] and we want to keep 4 digits of precision. So, roughly we need at least 14 terms in series like the geometric or log series. Let's make this 15. Then it doesn't make sense to compute intermediate summands with more than 6 digits precision. So we compute with 6 digits precision but return only 4 digits (rounded) after the decimal point. This result with 4 post-decimal points precision is then used as input to the next evaluation.

```
\loop
\noindent \hbox to 2.5cm {\hss\texttt{D(\the\cnta/100): }}%
\xintRound{4}
 {\xintAdd {\xintFxPtPowerSeriesX {1}{15}{\coefflog}{\the\cnta [-2]}{6}}
           {\xintFxPtPowerSeriesX {1}{15}{\coefflog}
                  {\xintRound {4}{\xintFxPtPowerSeriesX {1}{15}{\coeffalt}
                                 {\the\cnta [-2]}{6}}}
            {6}}%
 }\endgraf
\ifnum\cnta < 49 \advance\cnta 7 \repeat
```

D(0/100): 0 D(28/100): -0.0001
D(7/100): 0.0000 D(35/100): -0.0001
D(14/100): 0.0000 D(42/100): -0.0000
D(21/100): -0.0001 D(49/100): -0.0001

Not bad... I have cheated a bit: the 'four-digits precise' numeric evaluations were left unrounded in the final addition. However the inner rounding to four digits worked fine and made the next step faster than it would have been with longer inputs. The morale is that one should not use the raw results of `\xintFxPtPowerSeriesX` with the `D` digits with which it was computed, as the last are to be considered garbage. Rather, one should keep from the output only some smaller number of digits. This will make further computations faster and not less precise. I guess there should be some command to do this final truncating, or better, rounding, at a given number `D'`<`D` of digits. Maybe for the next release.

17.9 Computing $\log 2$ and π

In this final section, the use of `\xintFxPtPowerSeries` (and `\xintPowerSeries`) will be illustrated on the (expandable... why make things simple when it is so easy to make them difficult!) computations of the first digits of the decimal expansion of the familiar constants $\log 2$ and π.

Let us start with $\log 2$. We will get it from this formula (which is left as an exercise):

```
log(2)=-2 log(1-13/256)-5 log(1-1/9)
```

The number of terms to be kept in the log series, for a desired precision of `10^{-D}` was roughly estimated without much theoretical analysis. Computing exactly the partial sums with `\xintPowerSeries` and then printing the truncated values, from `D=0` up to `D=100` showed that it worked in terms of quality of the approximation. Because of possible strings of zeros or nines in the exact decimal expansion (in the present case of $\log 2$, strings of zeros around the fourtieth and the sixtieth decimals), this does not mean though that all digits printed were always exact. In the end one always end up having to compute at some higher level of desired precision to validate the earlier result.

Then we tried with `\xintFxPtPowerSeries`: this is worthwile only for `D`'s at least 50, as the exact evaluations are faster (with these short-length `x`'s) for a lower number of digits. And as expected the degradation in the quality of approximation was in this range of the order of two or three digits. This meant roughly that the 3+1=4 ending digits were wrong. Again, we ended up having to compute with five more digits and compare with the earlier value to validate it. We use truncation rather than rounding because our goal is not to obtain the correct rounded decimal expansion but the correct exact truncated one.

```
\def\coefflog #1{1/#1[0]}% 1/n
\def\xa {13/256[0]}%  we will compute log(1-13/256)
\def\xb {1/9[0]}%     we will compute log(1-1/9)
\def\LogTwo #1%
%  get log(2)=-2log(1-13/256)- 5log(1-1/9)
{% we want to use \printnumber, hence need something expanding in two steps
 % only, so we use here the \romannumeral0 method
  \romannumeral0\expandafter\LogTwoDoIt \expandafter
    % Nb Terms for 1/9:
  {\the\numexpr #1*150/143\expandafter}\expandafter
    % Nb Terms for 13/256:
  {\the\numexpr #1*100/129\expandafter}\expandafter
    % We print #1 digits, but we know the ending ones are garbage
  {\the\numexpr #1\relax}% allows #1 to be a count register
}%
\def\LogTwoDoIt #1#2#3%
```

```
%  #1=nb of terms for 1/9, #2=nb of terms for 13/256,
{% #3=nb of digits for computations, also used for printing
 \xinttrunc {#3} % lowercase form to stop the \romannumeral0 expansion!
 {\xintAdd
  {\xintMul {2}{\xintFxPtPowerSeries {1}{#2}{\coefflog}{\xa}{#3}}}
  {\xintMul {5}{\xintFxPtPowerSeries {1}{#1}{\coefflog}{\xb}{#3}}}%
 }%
}%
\noindent $\log 2 \approx \LogTwo {60}\dots$\endgraf
\noindent\phantom{$\log 2$}${}\approx{}$\printnumber{\LogTwo {65}}\dots\endgraf
\noindent\phantom{$\log 2$}${}\approx{}$\printnumber{\LogTwo {70}}\dots\endgraf
```

$\log 2 \approx 0.693147180559945309417232121458176568075500134360255254120484\ldots$
$\approx 0.6931471805599453094172321214581765680755001343602552541206800071 1\ldots$
$\approx 0.693147180559945309417232121458176568075500134360255254120680009493372 3\ldots$

Here is the code doing an exact evaluation of the partial sums. We have added a +1 to the number of digits for estimating the number of terms to keep from the log series: we experimented that this gets exactly the first D digits, for all values from D=0 to D=100, except in one case (D=40) where the last digit is wrong. For values of D higher than 100 it is more efficient to use the code using \xintFxPtPowerSeries.

```
\def\LogTwo #1% get log(2)=-2log(1-13/256)- 5log(1-1/9)
{%
    \romannumeral0\expandafter\LogTwoDoIt \expandafter
    {\the\numexpr (#1+1)*150/143\expandafter}\expandafter
    {\the\numexpr (#1+1)*100/129\expandafter}\expandafter
    {\the\numexpr #1\relax}%
}%
\def\LogTwoDoIt #1#2#3%
{%   #3=nb of digits for truncating an EXACT partial sum
  \xinttrunc {#3}
    {\xintAdd
      {\xintMul {2}{\xintPowerSeries {1}{#2}{\coefflog}{\xa}}}
      {\xintMul {5}{\xintPowerSeries {1}{#1}{\coefflog}{\xb}}}%
    }%
}%
```

Let us turn now to Pi, computed with the Machin formula. Again the numbers of terms to keep in the two arctg series were roughly estimated, and some experimentations showed that removing the last three digits was enough (at least for D=0-100 range). And the algorithm does print the correct digits when used with D=1000 (to be convinced of that one needs to run it for D=1000 and again, say for D=1010.) A theoretical analysis could help confirm that this algorithm always gets better than 10^{-D} precision, but again, strings of zeros or nines encountered in the decimal expansion may falsify the ending digits, nines may be zeros (and the last non-nine one should be increased) and zeros may be nine (and the last non-zero one should be decreased).

```
% pi = 16 Arctg(1/5) - 4 Arctg(1/239) (John Machin's formula)
\def\coeffarctg #1{\the\numexpr\ifodd#1 -1\else1\fi\relax/%
                                        \the\numexpr 2*#1+1\relax [0]}%
% the above computes (-1)^n/(2n+1).
```

```
% Alternatives:
% \def\coeffarctg #1{1/\the\numexpr\xintiMON{#1}*(2*#1+1)\relax }%
% The [0] can *not* be used above, as the denominator is signed.
% \def\coeffarctg #1{\xintiMON{#1}/\the\numexpr 2*#1+1\relax [0]}%
\def\xa {1/25[0]}%      1/5^2, the [0] for faster parsing
\def\xb {1/57121[0]}% 1/239^2, the [0] for faster parsing
\def\Machin #1{% \Machin {\mycount} is allowed
    \romannumeral0\expandafter\MachinA \expandafter
     % number of terms for arctg(1/5):
    {\the\numexpr (#1+3)*5/7\expandafter}\expandafter
     % number of terms for arctg(1/239):
    {\the\numexpr (#1+3)*10/45\expandafter}\expandafter
     % do the computations with 3 additional digits:
    {\the\numexpr #1+3\expandafter}\expandafter
     % allow #1 to be a count register:
    {\the\numexpr #1\relax }}%
\def\MachinA #1#2#3#4%
% #4: digits to keep after decimal point for final printing
% #3=#4+3: digits for evaluation of the necessary number of terms
% to be kept in the arctangent series, also used to truncate each
% individual summand.
{\xinttrunc {#4} % must be lowercase to stop \romannumeral0!
 {\xintSub
  {\xintMul {16/5}{\xintFxPtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}}
  {\xintMul {4/239}{\xintFxPtPowerSeries {0}{#2}{\coeffarctg}{\xb}{#3}}}%
 }}%
\[ \pi = \Machin {60}\dots \]
```

$$\pi = 3.14159265358979323846264338327950288419716939937510582097494 4\ldots$$

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 44 seconds last time I tried (and for 200 digits it is less than 1 second). As mentioned in the introduction, the file pi.tex by D. ROEGEL shows that orders of magnitude faster computations are possible within TEX, but recall our constraints of complete expandability and be merciful, please.

Why truncating rather than rounding? One of our main competitors on the market of scientific computing, a canadian product (not encumbered with expandability constraints, and having barely ever heard of TEX ;-), prints numbers rounded in the last digit. Why didn't we follow suit in the macros `\xintFxPtPowerSeries` and `\xintFxPtPowerSeriesX`? To round at `D` digits, and excluding a rewrite or cloning of the division algorithm which anyhow would add to it some overhead in its final steps, **xintfrac** needs to truncate at `D+1`, then round. And rounding loses information! So, with more time spent, we obtain a worst result than the one truncated at `D+1` (one could imagine that additions and so on, done with only `D` digits, cost less; true, but this is a negligeable effect per summand compared to the additional cost for this term of having been truncated at `D+1` then rounded). Rounding is the way to go when setting up algorithms to evaluate functions destined to be composed one after the other: exact algebraic operations with many summands and an `x` variable which is a fraction are costly and create an even bigger fraction; replacing `x` 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.

18 Commands of the xintcfrac package

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

18.1 Package overview

A *simple* continued fraction has coefficients `[c0,c1,...,cN]` (usually called partial quotients, but I really dislike this entrenched terminology), where `c0` is a positive or negative integer and the others are positive integers. As we will see it is possible with **xintcfrac** to specify the coefficient function `c:n->cn`. Note that the index then starts at zero as indicated. With the `amsmath` macro `\cfrac` one can display such a continued fraction as

$$c_0 + \cfrac{1}{c_1 + \cfrac{1}{c_2 + \cfrac{1}{c_3 + \cfrac{1}{\ddots}}}}$$

Here is a concrete example:

$$\frac{208341}{66317} = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{292 + \cfrac{1}{2}}}}}$$

But the difference with `amsmath`'s `\cfrac` is that this was input as

```
\[ \xintFrac {208341/66317}=\xintCFrac {208341/66317} \]
```

The command `\xintCFrac` produces in two expansion steps the whole thing with the many chained `\cfrac`'s and all necessary braces, ready to be printed, in math mode. This is LaTeX only and with the `amsmath` package (we shall mention another method for Plain TeX users of `amstex`).

A *generalized* continued fraction has the same structure but the numerators are not restricted to be ones, and numbers used in the continued fraction may be arbitrary, also fractions, irrationals, indeterminates. The *centered* continued fraction associated to a rational number is an example:

```
\[ \xintFrac {915286/188421}=\xintGCFrac {\xintFtoCC {915286/188421}} \]
```

$$\frac{915286}{188421} = 5 - \cfrac{1}{7 + \cfrac{1}{39 - \cfrac{1}{53 - \cfrac{1}{13}}}} = 4 + \cfrac{1}{1 + \cfrac{1}{6 + \cfrac{1}{38 + \cfrac{1}{1 + \cfrac{1}{51 + \cfrac{1}{1 + \cfrac{1}{12}}}}}}}$$

The command \xintGCFrac, contrarily to \xintCFrac, does not compute anything, it just typesets. Here, it is the command \xintFtoCC which did the computation of the centered continued fraction of f. Its output has the 'inline format' described in the next paragraph. In the display, we also used \xintCFrac (code not shown), for comparison of the two types of continued fractions.

A generalized continued fraction may be input 'inline' as:

```
a0+b0/a1+b1/a2+b2/...../a(n-1)+b(n-1)/an
```

Fractions among the coefficients are allowed but they must be enclosed within braces. Signed integers may be left without braces (but the + signs are mandatory). Or, they may be macros expanding (in two steps) to some number or fractional number.

```
\xintGCFrac {1+-1/57+\xintPow {-3}{7}/\xintQuo {132}{25}}
```

$$\frac{1907}{1902} = 1 - \cfrac{1}{57 - \cfrac{2187}{5}}$$

The left hand side was obtained with the following code:

```
\xintFrac{\xintGCtoF {1+-1/57+\xintPow {-3}{7}/\xintQuo {132}{25}}}
```

It uses the macro \xintGCtoF to convert a generalized fraction from the 'inline format' to the fraction it evaluates to.

A simple continued fraction is a special case of a generalized continued fraction and may be input as such to macros expecting the 'inline format', for example `-7+1/6+1/19+1/1+1/33`. There is a simpler comma separated format:

```
\xintFrac{\xintCstoF{-7,6,19,1,33}}=\xintCFrac{\xintCstoF{-7,6,19,1,33}}
```

$$\frac{-28077}{4108} = -7 + \cfrac{1}{6 + \cfrac{1}{19 + \cfrac{1}{1 + \cfrac{1}{33}}}}$$

This comma separated format may also be used with fractions among the coefficients: of course in that case, computing with \xintFtoCs from the resulting f its real coefficients will give a new comma separated list with only integers. This list has no spaces: the spaces

in the display below arise from the math mode processing.

```
\xintFrac{1084483/398959}=[\xintFtoCs{1084483/398959}]
```

$$\frac{1084483}{398959} = [2,1,2,1,1,4,1,1,6,1,1,8,1,1,10,2]$$

If one prefers other separators, one can use \xintFtoCx whose first argument will be the separator to be used.

```
\xintFrac{2721/1001}=\xintFtoCx {+1/(}{2721/1001})\cdots)
```

$$\frac{2721}{1001} = 2 + 1/(1 + 1/(2 + 1/(1 + 1/(1 + 1/(4 + 1/(1 + 1/(1 + 1/(6 + 1/(2)\cdots)$$

People using Plain TEX and amstex can achieve the same effect as \xintCFrac with: `$$\xintFwOver{2721/1001}=\xintFtoCx {+\cfrac1\\ }{2721/1001}\endcfrac$$`

Using \xintFtoCx with first argument an empty pair of braces {} will return the list of the coefficients of the continued fraction of f, without separator, and each one enclosed in a pair of group braces. This can then be manipulated by the non-expandable macro \xintAssignArray or the expandable ones \xintApply and \xintListWithSep.

As a shortcut to using \xintFtoCx with separator 1+/, there is \xintFtoGC:

```
2721/1001=\xintFtoGC {2721/1001}
```

```
2721/1001=2+1/1+1/2+1/1+1/1+1/4+1/1+1/1+1/6+1/2
```

Let us compare in that case with the output of \xintFtoCC:

```
2721/1001=\xintFtoCC {2721/1001}
```

```
2721/1001=3+-1/4+-1/2+1/5+-1/2+1/7+-1/2
```

The '\printnumber' macro which we use to print long numbers can also be useful on long continued fractions.

```
\printnumber{\xintFtoCC {350370189063507202043510 49/%
                         24424173788619740455818 0}}
```

143+1/2+1/5+-1/4+-1/4+-1/4+-1/3+1/2+1/2+1/6+-1/22+1/2+1/10+-1/5+-1/11+-1/3+1/4+-1/2+1/2+1/4+-1/2+1/23+1/3+1/8+-1/6+-1/9. If we apply \xintGCtoF to this generalized continued fraction, we discover that the original fraction was reducible:

```
\xintGCtoF {143+1/2+...+-1/9}=2897319801297630107/20197107104701740[0]
```

When a generalized continued fraction is built with integers, and numerators are only 1's or -1's, the produced fraction is irreducible. And if we compute it again with the last sub-fraction omitted we get another irreducible fraction related to the bigger one by a Bezout identity. Doing this here we get:

```
\xintGCtoF {143+1/2+...+-1/6}=328124887710626729/2287346221788023[0]
```

and indeed:

$$\begin{vmatrix} 2897319801297630107 & 328124887710626729 \\ 20197107104701740 & 2287346221788023 \end{vmatrix} = 1$$

More generally the various fractions obtained from the truncation of a continued fraction to its initial terms are called the convergents. The commands of xintcfrac such as \xintFtoCv, \xintFtoCCv, and others which compute such convergents, return them as a list of braced items, with no separator. This list can then be treated either with \xintAssignArray, or \xintListWithSep, or any other way (but then, some TEX programming knowledge will be necessary). Here is an example:

```
$$\xintFrac{915286/188421}\to \xintListWithSep {,}%
```

```
{\xintApply{\xintFrac}{\xintFtoCv{915286/188421}}}$$
```

$$\frac{915286}{188421} \to 4, 5, \frac{34}{7}, \frac{1297}{267}, \frac{1331}{274}, \frac{69178}{14241}, \frac{70509}{14515}, \frac{915286}{188421}$$

```
$$\xintFrac{915286/188421}\to \xintListWithSep {,}%
{\xintApply{\xintFrac}{\xintFtoCCv{915286/188421}}}$$
```

$$\frac{915286}{188421} \to 5, \frac{34}{7}, \frac{1331}{274}, \frac{70509}{14515}, \frac{915286}{188421}$$

We thus see that the 'centered convergents' obtained with \xintFtoCCv are among the fuller list of convergents as returned by \xintFtoCv.

Here is a more complicated use of \xintApply and \xintListWithSep. We first define a macro which will be applied to each convergent:

```
\newcommand{\mymacro}[1]{$\xintFrac{#1}=[\xintFtoCs{#1}]$\vtop to 6pt{}}
```

Next, we use the following code:

```
$\xintFrac{49171/18089}\to{}$
\xintListWithSep {, }{\xintApply{\mymacro}{\xintFtoCv{49171/18089}}}
```

It produces:

$\frac{49171}{18089} \to 2 = [2]$, $3 = [3]$, $\frac{8}{3} = [2, 1, 2]$, $\frac{11}{4} = [2, 1, 3]$, $\frac{19}{7} = [2, 1, 2, 2]$, $\frac{87}{32} = [2, 1, 2, 1, 1, 4]$, $\frac{106}{39} = [2, 1, 2, 1, 1, 5]$, $\frac{193}{71} = [2, 1, 2, 1, 1, 4, 2]$, $\frac{1264}{465} = [2, 1, 2, 1, 1, 4, 1, 1, 6]$, $\frac{1457}{536} = [2, 1, 2, 1, 1, 4, 1, 1, 7]$, $\frac{2721}{1001} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 2]$, $\frac{23225}{8544} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8]$, $\frac{49171}{18089} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 2]$.

The macro \xintCntoF allows to specify the coefficients as functions of the index. The values to which expand the coefficient function do not have to be integers.

```
\def\cn #1{\xintiPow {2}{#1}}% 2^n
\[\xintFrac{\xintCntoF {6}{\cn}}=\xintCFrac [l]{\xintCntoF {6}{\cn}}\]
```

$$\frac{3541373}{2449193} = 1 + \cfrac{1}{2 + \cfrac{1}{4 + \cfrac{1}{8 + \cfrac{1}{16 + \cfrac{1}{32 + \cfrac{1}{64}}}}}}$$

Notice the use of the optional argument [l] to \xintCFrac. Other possibilities are [r] and (default) [c].

```
\def\cn #1{\xintPow {2}{-#1}}% 1/2^n
\[\xintFrac{\xintCntoF {6}{\cn}} = \xintGCFrac [r]{\xintCntoGC {6}{\cn}}
```

```
= [\xintFtoCs {\xintCntoF {6}{\cn}}]\]
```

$$\frac{3159019}{2465449} = 1 + \cfrac{1}{\frac{1}{2} + \cfrac{1}{\frac{1}{4} + \cfrac{1}{\frac{1}{8} + \cfrac{1}{\frac{1}{16} + \cfrac{1}{\frac{1}{32} + \frac{1}{\frac{1}{64}}}}}}} = [1,3,1,1,4,14,1,1,1,1,79,2,1,1,2]$$

We used \xintCntoGC as we wanted to display also the continued fraction and not only the fraction returned by \xintCntoF.

There are also \xintGCntoF and \xintGCntoGC which allow the same for generalized fractions. The following initial portion of a generalized continued fraction for π:

$$\frac{92736}{29520} = \cfrac{4}{1 + \cfrac{1}{3 + \cfrac{4}{5 + \cfrac{9}{7 + \cfrac{16}{9 + \frac{25}{11}}}}}} = 3.1414634146\ldots$$

was obtained with this code:

```
\def\an #1{\the\numexpr 2*#1+1\relax }%
\def\bn #1{\the\numexpr (#1+1)*(#1+1)\relax }%
\[ \xintFrac{\xintDiv {4}{\xintGCntoF {5}{\an}{\bn}}} =
       \cfrac{4}{\xintGCFrac{\xintGCntoGC {5}{\an}{\bn}}} =
\xintTrunc {10}{\xintDiv {4}{\xintGCntoF {5}{\an}{\bn}}}\dots\]
```

We see that the quality of approximation is not fantastic compared to the simple continued fraction of π with about as many terms:

```
\[ \xintFrac{\xintCstoF{3,7,15,1,292,1,1}}=
   \xintGCFrac{3+1/7+1/15+1/1+1/292+1/1+1/1}=
   \xintTrunc{10}{\xintCstoF{3,7,15,1,292,1,1}}\dots\]
```

$$\frac{208341}{66317} = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{292 + \cfrac{1}{1 + \frac{1}{1}}}}}} = 3.1415926534\ldots$$

To conclude this overview of most of the package functionalities, let us explore the convergents of Euler's number e.

```
\def\cn #1{\the\numexpr\ifcase \numexpr #1+3-3*((#1+2)/3)\relax
                           1\or1\or2*(#1/3)\fi\relax }
% produces the pattern 1,1,2,1,1,4,1,1,6,1,1,8,... which are the
% coefficients of the simple continued fraction of e-1.
\cnta 0
\def\mymacro #1{\advance\cnta by 1
                \noindent
                \hbox to 3em {\hfil\small\texttt{\the\cnta.} }%
                $\xintTrunc {30}{\xintAdd {1[0]}{#1}}\dots=
                 \xintFrac{\xintAdd {1[0]}{#1}}$}%
\xintListWithSep{\vtop to 6pt{}\vbox to 12pt{}\par}
    {\xintApply\mymacro{\xintiCstoCv{\xintCntoCs {35}{\cn}}}}
```

The volume of computation is kept minimal by the following steps:

- a comma separated list of the first 36 coefficients is produced by \xintCntoCs,
- this is then given to \xintiCstoCv which produces the list of the convergents (there is also \xintCstoCv, but our coefficients being integers we used the infinitesimally faster \xintiCstoCv),
- then the whole list was converted into a sequence of one-line paragraphs, each convergent becomes the argument to a macro printing it together with its decimal expansion with 30 digits after the decimal point.
- A count register \cnta was used to give a line count serving as a visual aid: we could also have done that in an expandable way, but well, let's relax from time to time...

1. $2.000000000000000000000000000000\cdots = 2$
2. $3.000000000000000000000000000000\cdots = 3$
3. $2.666666666666666666666666666666\cdots = \frac{8}{3}$
4. $2.750000000000000000000000000000\cdots = \frac{11}{4}$
5. $2.714285714285714285714285714285\cdots = \frac{19}{7}$
6. $2.718750000000000000000000000000\cdots = \frac{87}{32}$
7. $2.717948717948717948717948717948\cdots = \frac{106}{39}$
8. $2.718309859154929577464788732394\cdots = \frac{193}{71}$
9. $2.718279569892473118279569892473\cdots = \frac{1264}{465}$
10. $2.718283582089552238805970149253\cdots = \frac{1457}{536}$
11. $2.718281718281718281718281718281\cdots = \frac{2721}{1001}$
12. $2.718281835205992509363295880149\cdots = \frac{23225}{8544}$
13. $2.718281822943949711891042430591\cdots = \frac{25946}{9545}$
14. $2.718281828735695726684725523798\cdots = \frac{49171}{18089}$
15. $2.718281828445401318035025074172\cdots = \frac{517656}{190435}$
16. $2.718281828470583721777828930962\cdots = \frac{566827}{208524}$

17. $2.718281828458563411277850606202\cdots = \frac{1084483}{398959}$
18. $2.718281828459065114074529546648\cdots = \frac{13580623}{4996032}$
19. $2.718281828459028013207065591026\cdots = \frac{14665106}{5394991}$
20. $2.718281828459045851404621084949\cdots = \frac{28245729}{10391023}$
21. $2.718281828459045213521983758221\cdots = \frac{410105312}{150869313}$
22. $2.718281828459045254624795027092\cdots = \frac{438351041}{161260336}$
23. $2.718281828459045234757560631479\cdots = \frac{848456353}{312129649}$
24. $2.718281828459045235379013372772\cdots = \frac{14013652689}{5155334720}$
25. $2.718281828459045235343535532787\cdots = \frac{14862109042}{5467464369}$
26. $2.718281828459045235360753230188\cdots = \frac{28875761731}{10622799089}$
27. $2.718281828459045235360274593941\cdots = \frac{534625820200}{196677847971}$
28. $2.718281828459045235360299120911\cdots = \frac{563501581931}{207300647060}$
29. $2.718281828459045235360287179900\cdots = \frac{1098127402131}{403978495031}$
30. $2.718281828459045235360287478611\cdots = \frac{22526049624551}{8286870547680}$
31. $2.718281828459045235360287464726\cdots = \frac{23624177026682}{8690849042711}$
32. $2.718281828459045235360287471503\cdots = \frac{46150226651233}{16977719590391}$
33. $2.718281828459045235360287471349\cdots = \frac{1038929163353808}{382200680031313}$
34. $2.718281828459045235360287471355\cdots = \frac{1085079390005041}{399178399621704}$
35. $2.718281828459045235360287471352\cdots = \frac{2124008553358849}{781379079653017}$
36. $2.718281828459045235360287471352\cdots = \frac{52061284670617417}{19152276311294112}$

The actual computation of the list of all 36 convergents accounts for only 8% of the total time (total time equal to about 5 hundredths of a second in my testing, on my laptop): another 80% is occupied with the computation of the truncated decimal expansions (and the addition of 1 to everything as the formula gives the continued fraction of $e-1$). One can with no problem compute much bigger convergents. Let's get the 200th convergent. It turns out to have the same first 268 digits after the decimal point as $e-1$. Higher convergents get more and more digits in proportion to their index: the 500th convergent already gets 799 digits correct! To allow speedy compilation of the source of this document when the need arises, I limit here to the 200th convergent (getting the 500th took about 1.2s on my laptop last time I tried, and the 200th convergent is obtained ten times faster).

```
\edef\z {\xintCntoF {199}{\cn}}%
\begingroup\parindent 0pt \leftskip 2.5cm
\indent\llap {Numerator = }\printnumber{\xintNumerator\z}\par
\indent\llap {Denominator = }\printnumber{\xintDenominator\z}\par
\indent\llap {Expansion = }\printnumber{\xintTrunc{268}\z}\dots
\par\endgroup
```

Numerator = 5689640388718962675975238923158078752938890176679174460572320245471922969611182301752438601749953108177313670124170860974963432938290 6

Denominator = 331123817669737619306256360816356753365468823729314438156205615463246659728581865461337692063148916019550614570592553376611426452172 23

Expansion = 1.718281828459045235360287471352662497757247093699959574966967627724076630353547594571382178525166427427466391932003059921817413596629043572900334295260595630738132328627943490763233829880753195251019011573834187930702154089149934884167509244761460668082264800168477411 8...

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.

18.2 \xintCFrac

`\xintCFrac{f}` is a math-mode only, LaTeX with `amsmath` only, macro which first computes then displays with the help of `\cfrac` the simple continued fraction corresponding to the given fraction (or macro expanding in two steps to one such). It admits an optional argument which may be `[l]`, `[r]` or (the default) `[c]` to specify the location of the one's in the numerators of the sub-fractions. Each coefficient is typeset using the `\xintFrac` macro from the xintfrac package.

18.3 \xintGCFrac

`\xintGCFrac{a+b/c+d/e+f/g+h/...}` uses similarly `\cfrac` to typeset a generalized continued fraction in inline format. It admits the same optional argument as `\xintCFrac`.

```
\[\xintGCFrac {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}}\]
```

$$1+\cfrac{3375\cdot 10^{-3}}{\frac{1}{7}-\cfrac{\frac{3}{5}}{720}}$$

As can be seen this is typesetting macro, although it does proceed to the evaluation of the coefficients themselves. See `\xintGCtoF` if you are impatient to see this fraction computed. Numerators and denominators are made arguments to the `\xintFrac` macro.

18.4 \xintGCtoGCx

New with release 1.05.

`\xintGCtoGCx{sepa}{sepb}{a+b/c+d/e+f/...+x/y}` returns the list of the coefficients of the generalized continued fraction of `f`, each one within a pair of braces, and separated with the help of `sepa` and `sepb`. Thus

`\xintGCtoGCx :;{1+2/3+4/5+6/7}` gives 1:2;3:4;5:6;7

Plain TeX+`amstex` users may be interested in:

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

18.5 \xintFtoCs

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

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

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

18.6 \xintFtoCx

`\xintFtoCx{sep}{f}` returns the list of the coefficients of the simple continued fraction of `f`, withing group braces and separated with the help of `sep`.

```
$$\xintFtoCx {+\cfrac1\\ }{f}\endcfrac$$
```

will display the continued fraction in `\cfrac` format, with Plain TEX and `amstex`.

18.7 \xintFtoGC

`\xintFtoGC{f}` does the same as `\xintFtoCx{+1/}{f}`. Its output may thus be used in the package macros expecting such an 'inline format'. This continued fraction is a *simple* one, not a *generalized* one, but as it is produced in the format used for user input of generalized continued fractions, the macro was called `\xintFtoGC` rather than `\xintFtoC` for example.

```
566827/208524=\xintFtoGC {566827/208524}
```

566827/208524=2+1/1+1/2+1/1+1/1+1/4+1/1+1/1+1/6+1/1+1/1+1/8+1/1+1/1+1/11

18.8 \xintFtoCC

`\xintFtoCC{f}` returns the 'centered' continued fraction of `f`, in 'inline format'.

```
566827/208524=\xintFtoCC {566827/208524}
```

566827/208524=3+-1/4+-1/2+1/5+-1/2+1/7+-1/2+1/9+-1/2+1/11

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

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

18.9 \xintFtoCv

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

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

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

18.10 \xintFtoCCv

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

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

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

18.11 \xintCstoF

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

```
\[\xintGCFrac {-1+1/3+1/-5+1/7+1/-9+1/11+1/-13}
=\xintSignedFrac{\xintCstoF {-1,3,-5,7,-9,11,-13}}
=\xintSignedFrac{\xintGCtoF {-1+1/3+1/-5+1/7+1/-9+1/11+1/-13}}\]
```

$$-1+\cfrac{1}{3+\cfrac{1}{-5+\cfrac{1}{7+\cfrac{1}{-9+\cfrac{1}{11+\cfrac{1}{-13}}}}}} = -\frac{75887}{118187} = -\frac{75887}{118187}$$

```
\xintGCFrac{{1/2}+1/{1/3}+1/{1/4}+1/{1/5}}=
   \xintFrac{\xintCstoF {1/2,1/3,1/4,1/5}}
```

$$\frac{1}{2}+\cfrac{1}{\frac{1}{3}+\cfrac{1}{\frac{1}{4}+\frac{1}{\frac{1}{5}}}} = \frac{159}{66}$$

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).

18.12 \xintCstoCv

\xintCstoCv{a,b,c,d,...,z} returns the list of the corresponding convergents. It is allowed to use fractions as coefficients (the computed convergents have then no reason to be the real convergents of the final fraction). When the coefficients are integers, the convergents are irreducible fractions, but otherwise it is of course not necessarily the case.

```
\xintListWithSep:{\xintCstoCv{1,2,3,4,5,6}}
1/1[0]:3/2[0]:10/7[0]:43/30[0]:225/157[0]:1393/972[0]
\xintListWithSep:{\xintCstoCv{1,1/2,1/3,1/4,1/5,1/6}}
```

```
1/1[0]:3/1[0]:9/7[0]:45/19[0]:225/159[0]:1575/729[0]
```

I know that these `[0]` are a bit annoying[27] but this is the way **xintfrac** likes to reception fractions: this format is best for further processing by the bundle macros. For 'inline' printing, one may apply `\xintRaw` and for display in math mode `\xintFrac`.

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

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

18.13 \xintCstoGC

`\xintCstoGC{a,b,..,z}` transforms a comma separated list (or something expanding to such a list) into an 'inline format' continued fraction `{a}+1/{b}+1/...+1/{z}`. The coefficients are just copied and put within braces, without expansion. The output can then be used in `\xintGCFrac` for example.

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

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

18.14 \xintGCtoF

`\xintGCtoF{a+b/c+d/e+f/g+.....+v/w+x/y}` computes the fraction defined by the inline generalized continued fraction. Coefficients may be fractions but must then be put within braces. They can be macros. The plus signs are mandatory.

```
\[\xintGCFrac {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}} =
\xintFrac{\xintGCtoF {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}}} =
\xintFrac{\xintIrr{\xintGCtoF
                  {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}}}}\]
```

$$1+\cfrac{3375\cdot 10^{-3}}{\frac{1}{7}-\frac{\frac{3}{5}}{720}} = \frac{88629000}{3579000} = \frac{29543}{1193}$$

```
\[ \xintGCFrac{{1/2}+{2/3}/{4/5}+{1/2}/{1/5}+{3/2}/{5/3}} =
   \xintFrac{\xintGCtoF {{1/2}+{2/3}/{4/5}+{1/2}/{1/5}+{3/2}/{5/3}}} \]
```

$$\frac{1}{2}+\cfrac{\frac{2}{3}}{\frac{4}{5}+\cfrac{\frac{1}{2}}{\frac{1}{5}+\frac{\frac{3}{2}}{\frac{5}{3}}}} = \frac{4270}{4140}$$

[27] and the awful truth is that it is added forcefully by `\xintCstoCv` at the last step...

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.

18.15 \xintGCtoCv

`\xintGCtoCv{a+b/c+d/e+f/g+.....+v/w+x/y}` returns the list of the corresponding convergents. The coefficients may be fractions, but must then be inside braces. Or they may be macros, too.

The convergents will in the general case be reducible. To put them into irreducible form, one needs one more step, for example it can be done with `\xintApply\xintIrr`.

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

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

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

18.16 \xintCntoF

`\xintCntoF{N}{\macro}` computes the fraction `f` having coefficients `c(j)=\macro{j}` for `j=0,1,...,N`. The `N` parameter is given to a `\numexpr`. The values of the coefficients, as returned by `\macro` do not have to be positive, nor integers, and it is thus not necessarily the case that the original `c(j)` are the true coefficients of the final `f`.

```
\def\macro #1{\the\numexpr 1+#1*#1\relax}\xintCntoF {5}{\macro}
```

72625/49902[0]

18.17 \xintGCntoF

`\xintGCntoF{N}{\macroA}{\macroB}` returns the fraction `f` corresponding to the inline generalized continued fraction `a0+b0/a1+b1/a2+...+b(N-1)/aN`, with `a(j)=\macroA{j}` and `b(j)=\macroB{j}`. The `N` parameter is given to a `\numexpr`.

$$1+\cfrac{1}{2-\cfrac{1}{3+\cfrac{1}{1-\cfrac{1}{2+\cfrac{1}{3-\cfrac{1}{1}}}}}} = \frac{39}{25}$$

There is also `\xintGCntoGC` to get the 'inline format' continued fraction. The previous display was obtained with:

```
\def\coeffA #1{\the\numexpr #1+4-3*((#1+2)/3)\relax }%
\def\coeffB #1{\xintMON{#1}}% (-1)^n
```

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

18.18 \xintCntoCs

`\xintCntoCs{N}{\macro}` produces the comma separated list of the corresponding coefficients, from `n=0` to `n=N`. The `N` is given to a `\numexpr`.

```
\def\macro #1{\the\numexpr 1+#1*#1\relax}\xintCntoCs {5}{\macro}
```

1,2,5,10,17,26

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

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

18.19 \xintCntoGC

`\xintCntoGC{N}{\macro}` evaluates the `c(j)=\macro{j}` from `j=0` to `j=N` and returns a continued fraction written in inline format: `{c(0)}+1/{c(1)}+1/...+1/{c(N)}`. The parameter `N` is given to a `\numexpr`. The coefficients, after expansion, are, as shown, being enclosed in an added pair of braces, they may thus be fractions.

```
\def\macro #1{\the\numexpr\ifodd#1 -1-#1\else1+#1\fi\relax/%
              \the\numexpr 1+#1*#1\relax}
\edef\x{\xintCntoGC {5}{\macro}}\texttt{\meaning\x}
\[\xintGCFrac{\xintCntoGC {5}{\macro}}\]
```

```
macro:->{1/1}+1/{-2/2}+1/{3/5}+1/{-4/10}+1/{5/17}+1/{-6/26}
```

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

18.20 \xintGCntoGC

`\xintGCntoGC{N}{\macroA}{\macroB}` evaluates the coefficients and then returns the corresponding `{a0}+{b0}/{a1}+{b1}/{a2}+...+{b(N-1)}/{aN}` inline generalized fraction. `N` is givent to a `\numexpr`. As shown, the coefficients are enclosed into added pairs of braces, and may thus be fractions.

```
\def\an #1{\the\numexpr #1*#1*#1+1\relax}%
\def\bn #1{\the\numexpr \xintiMON{#1}*(#1+1)\relax}%
```

```
\texttt{\xintGCntoGC {5}{\an}{\bn}}%
       ${}=\xintGCFrac {\xintGCntoGC {5}{\an}{\bn}}
        = \displaystyle\xintFrac    {\xintGCntoF  {5}{\an}{\bn}}$\par
1+1/2+-2/9+3/28+-4/65+5/126
```

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

18.21 \xintiCstoF, \xintiGCtoF, \xintiCstoCv, \xintiGCtoCv

The same as the corresponding macros without the 'i', but for integer-only input. Infinitesimally faster; to notice the higher efficiency one would need to use them with an input having (at least) hundreds of coefficients.

18.22 \xintGCtoGC

`\xintGCtoGC{a+b/c+d/e+f/g+.....+v/w+x/y}` expands (with the usual meaning) each one of the coefficients and returns an inline continued fraction of the same type, each expanded coefficient being enclosed withing braces.

```
\edef\x {\xintGCtoGC
  {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}+\xintCstoF {2,-7,-5}/16}}
\texttt{\meaning\x}
```

```
macro:->{1}+{3375/1[-3]}/{1/7}+{-3/5}/{720}+{67/36[0]}/{16}
```

To be honest I have, it seems, forgotten why I wrote this macro in the first place.

19 Package xint implementation

The commenting of the macros is currently (2013/05/14) very sparse.

Contents

19.1 Catcodes, ε-TEX and reload detection

The method for package identification and reload detection is copied verbatim from the packages by Heiko Oberdiek.

The method for catcodes was also inspired by these packages, we proceed slightly differently. 1.05 adds a \relax near the end of \XINT_restorecatcodes_endinput. Plain TeX users following the doc instruction to do \input xint.sty\relax were anyhow protected from any side effect. I didn't realize earlier that the \endinput would not have had the effect of stopping the scanning from the last \the\catcode61.

Starting with version 1.06 of the package, also ‘ must be sanitized, because we replace everywhere in the code the twice-expansion done with \expandafter by the systematic use of \romannumeral-‘0.

Starting with version 1.06b I decide that I suffer from an indigestion of @ signs, so I replace them all with underscores _, à la LATEX3. As this change makes it a bit more difficult to access the few private macros which were mentioned in the user documentation, I renamed them with only letters.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode95=11   % _  (starting with 1.06b, used inside cs names)
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \expandafter\let\expandafter\x\csname ver@xint.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xint}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax % plain-TeX, first loading
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
      \else
        \y{xint}{I was already loaded, aborting input}%
        \aftergroup\endinput
      \fi
    \fi
  \fi
  \def\ChangeCatcodesIfInputNotAborted
  {%
      \endgroup
      \edef\XINT_restorecatcodes_endinput
      {%
        \catcode96=\the\catcode96   % `
        \catcode47=\the\catcode47   % /
        \catcode41=\the\catcode41   % )
        \catcode40=\the\catcode40   % (
        \catcode42=\the\catcode42   % *
        \catcode43=\the\catcode43   % +
        \catcode62=\the\catcode62   % >
        \catcode60=\the\catcode60   % <
        \catcode58=\the\catcode58   % :
```

```
      \catcode46=\the\catcode46   % .
      \catcode45=\the\catcode45   % -
      \catcode44=\the\catcode44   % ,
      \catcode35=\the\catcode35   % #
      \catcode95=\the\catcode95   % _
      \catcode125=\the\catcode125 % }
      \catcode123=\the\catcode123 % {
      \endlinechar=\the\endlinechar
      \catcode13=\the\catcode13   % ^^M
      \catcode32=\the\catcode32   %
      \catcode61=\the\catcode61\relax   % =
      \noexpand\endinput
    }%
    \def\XINT_setcatcodes
    {%
      \catcode61=12   % =
      \catcode32=10   % space
      \catcode13=5    % ^^M
      \endlinechar=13 %
      \catcode123=1   % {
      \catcode125=2   % }
      \catcode95=11   % _ (replaces @ everywhere, starting with 1.06b)
      \catcode35=6    % #
      \catcode44=12   % ,
      \catcode45=12   % -
      \catcode46=12   % .
      \catcode58=11   % : (made letter for error cs)
      \catcode60=12   % <
      \catcode62=12   % >
      \catcode43=12   % +
      \catcode42=12   % *
      \catcode40=12   % (
      \catcode41=12   % )
      \catcode47=12   % /
      \catcode96=12   % `
    }%
    \XINT_setcatcodes
}%
\ChangeCatcodesIfInputNotAborted
```

19.2 Package identification

Copied verbatim from Heiko Oberdiek's packages.

```
\begingroup
  \catcode64=11 % @
  \catcode91=12 % [
  \catcode93=12 % ]
  \catcode58=12 % : (does not really matter, was letter)
```

```
  \expandafter\ifx\csname ProvidesPackage\endcsname\relax
    \def\x#1#2#3[#4]{\endgroup
      \immediate\write-1{Package: #3 #4}%
      \xdef#1{#4}%
    }%
  \else
    \def\x#1#2[#3]{\endgroup
      #2[{#3}]%
      \ifx#1\@undefined
        \xdef#1{#3}%
      \fi
      \ifx#1\relax
        \xdef#1{#3}%
      \fi
    }%
  \fi
\expandafter\x\csname ver@xint.sty\endcsname
\ProvidesPackage{xint}%
  [2013/05/14 v1.06b Expandable operations on long numbers (jfB)]%
```

19.3 Token management macros

```
\def\xint_gobble_     {}%
\def\xint_gobble_i    #1{}%
\def\xint_gobble_ii   #1#2{}%
\def\xint_gobble_iii  #1#2#3{}%
\def\xint_gobble_iv   #1#2#3#4{}%
\def\xint_gobble_v    #1#2#3#4#5{}%
\def\xint_gobble_vi   #1#2#3#4#5#6{}%
\def\xint_gobble_vii  #1#2#3#4#5#6#7{}%
\def\xint_gobble_viii #1#2#3#4#5#6#7#8{}%
\def\xint_firstoftwo  #1#2{#1}%
\def\xint_secondoftwo #1#2{#2}%
\def\xint_firstoftwo_andstop  #1#2{ #1}%
\def\xint_secondoftwo_andstop #1#2{ #2}%
\def\xint_exchangetwo_keepbraces_andstop #1#2{ {#2}{#1}}%
\def\xint_minus_andstop { -}%
\def\xint_gob_til_r     #1\R {}%
\def\xint_gob_til_w     #1\W {}%
\def\xint_gob_til_z     #1\Z {}%
\def\xint_gob_til_zero  #10{}%
\def\xint_gob_til_one   #11{}%
\def\xint_gob_til_zeros_iv #10000{}%
\def\xint_gob_til_relax      #1\relax {}%
\def\xint_gob_til_xint_undef #1\xint_undef {}%
\def\xint_gob_til_xint_relax #1\xint_relax {}%
\def\xint_UDzerofork      #10\dummy  #2#3\krof {#2}%
\def\xint_UDsignfork      #1-\dummy  #2#3\krof {#2}%
\def\xint_UDwfork         #1\W\dummy #2#3\krof {#2}%
```

```
\def\xint_UDzerosfork     #100\dummy #2#3\krof {#2}%
\def\xint_UDonezerofork   #110\dummy #2#3\krof {#2}%
\def\xint_UDzerominusfork #10-\dummy #2#3\krof {#2}%
\def\xint_UDsignsfork     #1--\dummy #2#3\krof {#2}%
\def\xint_afterfi #1#2\fi {\fi #1}%
\let\xint_relax\relax
\def\xint_braced_xint_relax {\xint_relax }%
```

19.4 \xintRev, \xintReverseOrder

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

```
\def\xintRev {\romannumeral0\xintrev }%
\def\xintrev #1%
{%
    \expandafter\XINT_rev_fork
    \romannumeral-`0#1\xint_relax % empty #1 ok
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
    \xint_relax
}%
\def\XINT_rev_fork #1%
{%
    \xint_UDsignfork
      #1\dummy {\expandafter\xint_minus_andstop
                \romannumeral0\XINT_rord_main {}}%
       -\dummy {\XINT_rord_main {}#1}%
    \krof
}%
\def\XINT_Rev         {\romannumeral0\XINT_rev }%
\def\xintReverseOrder {\romannumeral0\XINT_rev }%
\def\XINT_rev #1%
{%
    \XINT_rord_main {}#1%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
}%
\def\XINT_rord_main #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint_undef #9\XINT_rord_cleanup\xint_undef
    \XINT_rord_main {#9#8#7#6#5#4#3#2#1}%
}%
\def\XINT_rord_cleanup\xint_undef\XINT_rord_main #1#2\xint_relax
{%
    \expandafter\space\xint_gob_til_xint_relax #1%
}%
```

19.5 \xintRevWithBraces

New with 1.06. Makes the expansion of its argument and then reverses the resulting tokens or braced tokens, adding a pair of braces to each (thus, maintaining it when it was already there.)

```
\def\xintRevWithBraces         {\romannumeral0\xintrevwithbraces }%
\def\xintRevWithBracesNoExpand {\romannumeral0\xintrevwithbracesnoexpand }%
\def\xintrevwithbraces #1%
{%
    \expandafter\XINT_revwbr_loop % #1 empty ok
    \romannumeral-`0#1\xint_relax\xint_relax\xint_relax\xint_relax
                       \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\def\xintrevwithbracesnoexpand #1%
{%
    \romannumeral0\XINT_revwbr_loop
    #1\xint_relax\xint_relax\xint_relax\xint_relax
      \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\def\XINT_revwbr_loop #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint_relax #9\XINT_revwbr_finish_a\xint_relax
    \XINT_revwbr_loop {{#9}{#8}{#7}{#6}{#5}{#4}{#3}{#2}#1}%
}%
\def\XINT_revwbr_finish_a\xint_relax\XINT_revwbr_loop #1#2\Z
{%
    \XINT_revwbr_finish_b #2\R\R\R\R\R\R\R\Z #1%
}%
\def\XINT_revwbr_finish_b #1#2#3#4#5#6#7#8\Z
{%
    \xint_gob_til_r
            #1\XINT_revwbr_finish_c 8%
            #2\XINT_revwbr_finish_c 7%
            #3\XINT_revwbr_finish_c 6%
            #4\XINT_revwbr_finish_c 5%
            #5\XINT_revwbr_finish_c 4%
            #6\XINT_revwbr_finish_c 3%
            #7\XINT_revwbr_finish_c 2%
            \R\XINT_revwbr_finish_c 1\Z
}%
\def\XINT_revwbr_finish_c #1#2\Z
{%
    \expandafter\expandafter\expandafter
        \space
    \csname xint_gobble_\romannumeral #1\endcsname
}%
```

19.6 \xintLen, \xintLength

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

```
\def\xintiLen {\romannumeral0\xintilen }%
\def\xintilen #1%
{%
    \expandafter\XINT_length_fork
    \romannumeral-`0#1\xint_relax\xint_relax\xint_relax\xint_relax
                        \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\let\xintLen\xintiLen \let\xintlen\xintilen
\def\XINT_Len #1%
{%
    \romannumeral0\XINT_length_fork
    #1\xint_relax\xint_relax\xint_relax\xint_relax
      \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\def\XINT_length_fork #1%
{%
    \expandafter\XINT_length_loop
    \xint_UDsignfork
      #1\dummy {{0}}%
       -\dummy {{0}#1}%
    \krof
}%
\def\XINT_Length {\romannumeral0\XINT_length }%
\def\XINT_length #1%
{%
    \XINT_length_loop
    {0}#1\xint_relax\xint_relax\xint_relax\xint_relax
         \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\let\xintLength\XINT_Length
\def\XINT_length_loop #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint_relax #9\XINT_length_finish_a\xint_relax
    \expandafter\XINT_length_loop\expandafter {\the\numexpr #1+8\relax}%
}%
\def\XINT_length_finish_a\xint_relax
    \expandafter\XINT_length_loop\expandafter #1#2\Z
{%
    \XINT_length_finish_b #2\W\W\W\W\W\W\W\Z {#1}%
}%
\def\XINT_length_finish_b #1#2#3#4#5#6#7#8\Z
{%
    \xint_gob_til_w
```

```
            #1\XINT_length_finish_c 8%
            #2\XINT_length_finish_c 7%
            #3\XINT_length_finish_c 6%
            #4\XINT_length_finish_c 5%
            #5\XINT_length_finish_c 4%
            #6\XINT_length_finish_c 3%
            #7\XINT_length_finish_c 2%
            \W\XINT_length_finish_c 1\Z
}%
\def\XINT_length_finish_c #1#2\Z #3%
   {\expandafter\space\the\numexpr #3-#1\relax}%
```

19.7 \xintCSVtoList

`\xintCSVtoList {a,b,..,z}` returns `{a}{b}...{z}`. The comma separated list may be a macro which is first expanded. Each chain of spaces from the initial input will be collapsed as usual by the TeX initial scanning First included in release 1.06.

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

```
            #3\XINT_csvtol_finish_c 6%
            #4\XINT_csvtol_finish_c 5%
            #5\XINT_csvtol_finish_c 4%
            #6\XINT_csvtol_finish_c 3%
            #7\XINT_csvtol_finish_c 2%
            \R\XINT_csvtol_finish_c 1\Z
}%
\def\XINT_csvtol_finish_c #1#2\Z
{%
    \csname XINT_csvtol_finish_d\romannumeral #1\endcsname
}%
\def\XINT_csvtol_finish_dviii #1#2#3#4#5#6#7#8#9{ #9}%
\def\XINT_csvtol_finish_dvii  #1#2#3#4#5#6#7#8#9{ #9{#1}}%
\def\XINT_csvtol_finish_dvi   #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}}%
\def\XINT_csvtol_finish_dv    #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}}%
\def\XINT_csvtol_finish_div   #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}}%
\def\XINT_csvtol_finish_diii  #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}{#5}}%
\def\XINT_csvtol_finish_dii   #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}{#5}{#6}}%
\def\XINT_csvtol_finish_di    #1#2#3#4#5#6#7#8#9%
                                          { #9{#1}{#2}{#3}{#4}{#5}{#6}{#7}}%
```

19.8 \xintListWithSep

```
\xintListWithSep {sep}{{a}{b}...{z}} returns a sep b sep .... sep z
Included in release 1.04. The 'sep' can be \par's: the macro xintlistwithsep
etc... are all declared long. 'sep' does not have to be a single token. The list
may be a macro it is first expanded. 1.06 modifies the 'feature' of returning
sep if the list is empty: the output is now empty in that case. (sep was not
used for a one element list, but strangely it was for a zero-element list).
```

```
\def\xintListWithSep {\romannumeral0\xintlistwithsep }%
% \def\xintListWithSepNoExpand {\romannumeral0\xintlistwithsepnoexpand }%
\long\def\xintlistwithsep #1#2%
    {\expandafter\XINT_lws\expandafter {\romannumeral-`0#2}{#1}}%
\long\def\XINT_lws #1#2{\XINT_lws_start {#2}#1\Z }%
% \long\def\xintlistwithsepnoexpand #1#2%
%     {\romannumeral0\XINT_lws_start {#1}#2\Z }%
\long\def\XINT_lws_start #1#2%
{%
    \xint_gob_til_z #2\XINT_lws_dont\Z
    \XINT_lws_loop_a {#2}{#1}%
}%
\long\def\XINT_lws_dont\Z\XINT_lws_loop_a #1#2{ }%
\long\def\XINT_lws_loop_a #1#2#3%
{%
    \xint_gob_til_z #3\XINT_lws_end\Z
    \XINT_lws_loop_b {#1}{#2#3}{#2}%
}%
\long\def\XINT_lws_loop_b #1#2{\XINT_lws_loop_a {#1#2}}%
```

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

19.9 \xintNthElt

\xintNthElt {i}{{a}{b}...{z}} (or 'tokens' abcd...z) returns the i th element (one pair of braces removed). The list is first expanded. First included in release 1.06. With 1.06a, a value of i = 0 (or negative) makes the macro return the length. This is different from \xintLen which is for numbers (checks sign) and different from \xintLength which does not first expand its argument.

```
\def\xintNthElt         {\romannumeral0\xintnthelt }%
% \def\xintNthEltNoExpand {\romannumeral0\xintntheltnoexpand }%
\def\xintnthelt #1#2%
{%
    \expandafter\XINT_nthelt\expandafter {\romannumeral-`0#2}%
                                         {\numexpr #1\relax }%
}%
% \def\xintntheltnoexpand #1#2%
% {%
%     \romannumeral0\XINT_nthelt {#2}{\numexpr #1\relax}%
% }%
\def\XINT_nthelt #1#2%
{%
    \ifnum #2>0
          \xint_afterfi {\XINT_nthelt_loop_a {#2}}%
    \else
          \xint_afterfi {\XINT_length_loop {0}}%
    \fi #1\xint_relax\xint_relax\xint_relax\xint_relax
          \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\def\XINT_nthelt_loop_a #1%
{%
    \ifnum #1>8
        \expandafter\XINT_nthelt_loop_b
    \else
        \expandafter\XINT_nthelt_getit
    \fi
    {#1}%
}%
\def\XINT_nthelt_loop_b #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint_relax #9\XINT_nthelt_silentend\xint_relax
    \expandafter\XINT_nthelt_loop_a\expandafter{\the\numexpr #1-8\relax}%
}%
\def\XINT_nthelt_silentend #1\Z { }%
\def\XINT_nthelt_getit #1%
{%
    \expandafter\expandafter\expandafter\XINT_nthelt_finish
    \csname xint_gobble_\romannumeral\numexpr#1-1\endcsname
```

```
}%
\def\XINT_nthelt_finish #1#2\Z
{%
    \xint_UDwfork
      #1\dummy { }%
      \W\dummy { #1}%
    \krof
}%
```

19.10 \xintApply

```
\xintApply {\macro}{{a}{b}...{z}} returns {\macro{a}}...{\macro{b}} where each
instance of \macro is expanded. The list is first expanded. Introduced with re-
lease 1.04
```

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

19.11 \xintApplyUnbraced

```
\xintApplyUnbraced {\macro}{{a}{b}...{z}} returns \macro{a}...\macro{b} where
each instance of \macro is expanded using \romannumeral-`0. 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
```

```
\def\xintApplyUnbraced {\romannumeral0\xintapplyunbraced }%
% \def\xintApplyUnbracedNoExpand {\romannumeral0\xintapplyunbracednoexpand }%
\def\xintapplyunbraced #1#2%
{%
    \expandafter\XINT_applyunbr\expandafter {\romannumeral-`0#2}%
    {#1}%
}%
```

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

19.12 \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 an
exception in the xint bundle, they do not care at all about compatibility with
expansion-only contexts.
  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 \xin-
tRelaxArray) 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 \xintAssig-
nArray is ok.
```

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

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

```
          \csname\xint_arrayname\the\count 255\endcsname{\xint_temp }%
       \expandafter\XINT_assignarray_loop
    \fi
}%
\def\XINT_assignarray_end_a #1%
{%
    \expandafter\XINT_assignarray_end_b\expandafter #1%
}%
\def\XINT_assignarray_end_b #1#2#3%
{%
    \expandafter\XINT_assignarray_end_c
    \expandafter #1\expandafter #2\expandafter {#3}%
}%
\def\XINT_assignarray_end_c #1#2#3#4%
{%
    \def #4##1%
    {%
        \romannumeral0\expandafter #1\expandafter{\the\numexpr ##1}%
    }%
    \def #1##1%
    {%
        \ifnum ##1< 0
            \xint_afterfi {\xintError:ArrayIndexIsNegative\space 0}%
        \else
            \xint_afterfi {%
              \ifnum ##1> #2
                  \xint_afterfi {\xintError:ArrayIndexBeyondLimit\space 0}%
              \else
                  \xint_afterfi
                  {\expandafter\expandafter\expandafter
                   \space\csname #3##1\endcsname}%
              \fi}%
        \fi
     }%
}%
\let\xintDigitsOf\xintAssignArray
```

19.13 \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\Z
Attention, ceci n'est utilisé que pour des chaînes de chiffres, et donc le com-
portement avec des {..} ou autres espaces n'a fait l'objet d'aucune attention
```

```
\def\XINT_RQ #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_r #9\XINT_RQ_end_a\R\XINT_RQ {#9#8#7#6#5#4#3#2#1}%
}%
```

```
\def\XINT_RQ_end_a\R\XINT_RQ #1#2\Z
{%
    \XINT_RQ_end_b #1\Z
}%
\def\XINT_RQ_end_b #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_r
            #8\XINT_RQ_end_viii
            #7\XINT_RQ_end_vii
            #6\XINT_RQ_end_vi
            #5\XINT_RQ_end_v
            #4\XINT_RQ_end_iv
            #3\XINT_RQ_end_iii
            #2\XINT_RQ_end_ii
            \R\XINT_RQ_end_i
            \Z #2#3#4#5#6#7#8%
}%
\def\XINT_RQ_end_viii #1\Z #2#3#4#5#6#7#8#9\Z { #9}%
\def\XINT_RQ_end_vii  #1\Z #2#3#4#5#6#7#8#9\Z { #8#9000}%
\def\XINT_RQ_end_vi   #1\Z #2#3#4#5#6#7#8#9\Z { #7#8#900}%
\def\XINT_RQ_end_v    #1\Z #2#3#4#5#6#7#8#9\Z { #6#7#8#90}%
\def\XINT_RQ_end_iv   #1\Z #2#3#4#5#6#7#8#9\Z { #5#6#7#8#9}%
\def\XINT_RQ_end_iii  #1\Z #2#3#4#5#6#7#8#9\Z { #4#5#6#7#8#9000}%
\def\XINT_RQ_end_ii   #1\Z #2#3#4#5#6#7#8#9\Z { #3#4#5#6#7#8#900}%
\def\XINT_RQ_end_i      \Z #1#2#3#4#5#6#7#8\Z { #1#2#3#4#5#6#7#80}%
```

19.14 \XINT_cuz

```
\def\xint_cleanupzeros_andstop #1#2#3#4%
{%
    \expandafter\space\the\numexpr #1#2#3#4\relax
}%
\def\xint_cleanupzeros_nospace #1#2#3#4%
{%
    \the\numexpr #1#2#3#4\relax
}%
\def\XINT_rev_andcuz #1%
{%
    \expandafter\xint_cleanupzeros_andstop
    \romannumeral0\XINT_rord_main {}#1%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
}%
```

```
routine CleanUpZeros. Utilisée en particulier par la soustraction.
INPUT: longueur **multiple de 4** (<-- ATTENTION)
OUTPUT: on a retiré tous les leading zéros, on n'est **plus* nécessairement de
```

longueur 4n
Délimiteur pour _main: \W\W\W\W\W\W\W\Z avec SEPT \W

```
\def\XINT_cuz #1%
{%
    \XINT_cuz_loop #1\W\W\W\W\W\W\W\Z%
}%
\def\XINT_cuz_loop #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_w #8\xint_cuz_end_a\W
    \xint_gob_til_z #8\xint_cuz_end_A\Z
    \XINT_cuz_check_a {#1#2#3#4#5#6#7#8}%
}%
\def\xint_cuz_end_a #1\XINT_cuz_check_a #2%
{%
    \xint_cuz_end_b #2%
}%
\def\xint_cuz_end_b #1#2#3#4#5\Z
{%
    \expandafter\space\the\numexpr #1#2#3#4\relax
}%
\def\xint_cuz_end_A \Z\XINT_cuz_check_a #1{ 0}%
\def\XINT_cuz_check_a #1%
{%
    \expandafter\XINT_cuz_check_b\the\numexpr #1\relax
}%
\def\XINT_cuz_check_b #1%
{%
    \xint_gob_til_zero #1\xint_cuz_backtoloop 0\XINT_cuz_stop #1%
}%
\def\XINT_cuz_stop #1\W #2\Z{ #1}%
\def\xint_cuz_backtoloop 0\XINT_cuz_stop 0{\XINT_cuz_loop }%
```

19.15 \XINT_isOne

Added in 1.03. Attention: does not do any expansion.

```
\def\XINT_isOne #1{\romannumeral0\XINT_isone #1\W\Z }%
\def\XINT_isone #1#2%
{%
    \xint_gob_til_one #1\XINT_isone_b 1%
    \expandafter\space\expandafter 0\xint_gob_til_z #2%
}%
\def\XINT_isone_b #1\xint_gob_til_z #2%
{%
    \xint_gob_til_w #2\XINT_isone_yes \W
    \expandafter\space\expandafter 0\xint_gob_til_z
}%
\def\XINT_isone_yes #1\Z { 1}%
```

19.16 \xintNum

```
For example \xintNum {----+-++++---+----000000000000003}
1.05 defines \xintiNum, which allows redefinition of \xintNum by xintfrac.sty
Slightly modified in 1.06b (\R->\xint_relax) to avoid initial re-scan of in-
put stack (while still allowing empty #1)

\def\xintiNum {\romannumeral0\xintinum }%
\def\xintinum #1%
{%
    \expandafter\XINT_num_loop
    \romannumeral-`0#1\xint_relax\xint_relax\xint_relax\xint_relax
                       \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\let\xintNum\xintiNum \let\xintnum\xintinum
\def\XINT_num #1%
{%
    \XINT_num_loop #1\xint_relax\xint_relax\xint_relax\xint_relax
                      \xint_relax\xint_relax\xint_relax\xint_relax\Z
}%
\def\XINT_num_loop #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_xint_relax #8\XINT_num_end\xint_relax
    \XINT_num_NumEight #1#2#3#4#5#6#7#8%
}%
\def\XINT_num_end\xint_relax\XINT_num_NumEight #1\xint_relax #2\Z
{%
    \expandafter\space\the\numexpr #1+0\relax
}%
\def\XINT_num_NumEight #1#2#3#4#5#6#7#8%
{%
    \ifnum \numexpr #1#2#3#4#5#6#7#8+0= 0
      \xint_afterfi {\expandafter\XINT_num_keepsign_a
                     \the\numexpr #1#2#3#4#5#6#7#81\relax}%
    \else
      \xint_afterfi {\expandafter\XINT_num_finish
                     \the\numexpr #1#2#3#4#5#6#7#8\relax}%
    \fi
}%
\def\XINT_num_keepsign_a #1%
{%
    \xint_gob_til_one#1\XINT_num_gobacktoloop 1\XINT_num_keepsign_b
}%
\def\XINT_num_gobacktoloop 1\XINT_num_keepsign_b {\XINT_num_loop }%
\def\XINT_num_keepsign_b #1{\XINT_num_loop -}%
\def\XINT_num_finish #1\xint_relax #2\Z { #1}%
```

19.17 \xintSgn

Changed in 1.05. Earlier code was unnecessarily strange.

```
\def\xintiSgn {\romannumeral0\xintisgn }%
\def\xintisgn #1%
{%
    \expandafter\XINT_sgn \romannumeral-`0#1\Z%
}%
\let\xintSgn\xintiSgn \let\xintsgn\xintisgn
\def\XINT_Sgn #1{\romannumeral0\XINT_sgn #1\Z }%
\def\XINT_sgn #1#2\Z
{%
    \xint_UDzerominusfork
      #1-\dummy  { 0}%
      0#1\dummy  { -1}%
       0-\dummy  { 1}%
    \krof
}%
```

19.18 \xintOpp

```
\def\xintiOpp {\romannumeral0\xintiopp }%
\def\xintiopp #1%
{%
    \expandafter\XINT_opp \romannumeral-`0#1%
}%
\let\xintOpp\xintiOpp \let\xintopp\xintiopp
\def\XINT_Opp #1{\romannumeral0\XINT_opp #1}%
\def\XINT_opp #1%
{%
    \xint_UDzerominusfork
      #1-\dummy  { 0}%      zero
      0#1\dummy  { }%     negative
       0-\dummy  { -#1}%  positive
    \krof
}%
```

19.19 \xintAbs

```
\def\xintiAbs {\romannumeral0\xintiabs }%
\def\xintiabs #1%
{%
    \expandafter\XINT_abs \romannumeral-`0#1%
}%
\let\xintAbs\xintiAbs \let\xintabs\xintiabs
\def\XINT_Abs #1{\romannumeral0\XINT_abs #1}%
\def\XINT_abs #1%
{%
    \xint_UDsignfork
```

```
      #1\dummy  { }%
       -\dummy  { #1}%
    \krof
}%
```

```
-----------------------------------------------------------------
-----------------------------------------------------------------
ARITHMETIC OPERATIONS: ADDITION, SUBTRACTION, SUMS, MULTIPLICATION, PRODUCTS,
FACTORIAL, POWERS, EUCLIDEAN DIVISION.
  Release 1.03 re-organizes sub-routines to facilitate future developments: the
diverse variants of addition, with diverse conditions on inputs and output are
first listed; they will be used in multiplication, or in the summation, or in
the power routines. I am aware that the commenting is close to non-existent,
sorry about that.
  ADDITION I: \XINT_add_A
  INPUT:
\romannumeral0\XINT_add_A 0{}<N1>\W\X\Y\Z <N2>\W\X\Y\Z
1. <N1> et <N2> renversés
2. de longueur 4n (avec des leading zéros éventuels)
3. l'un des deux ne doit pas se terminer par 0000
[Donc on peut avoir 0000 comme input si l'autre est >0 et ne se termine pas en
0000 bien sûr]. On peut avoir l'un des deux vides. Mais alors l'autre ne doit
être ni vide ni 0000.
  OUTPUT: la somme <N1>+<N2>, ordre normal, plus sur 4n, pas de leading zeros
La procédure est plus rapide lorsque <N1> est le plus court des deux.
Nota bene: (30 avril 2013). J'ai une version qui est deux fois plus rapide sur
des nombres d'environ 1000 chiffres chacun, et qui commence à être avantageuse
pour des nombres d'au moins 200 chiffres. Cependant il serait vraiment compliqué
d'en étendre l'utilisation aux emplois de l'addition dans les autres routines,
comme celle de multiplication ou celle de division; et son implémentation ajouterait
au minimum la mesure de la longueur des summands.
```

```
\def\XINT_add_A #1#2#3#4#5#6%
{%
    \xint_gob_til_w #3\xint_add_az\W
    \XINT_add_AB #1{#3#4#5#6}{#2}%
}%
\def\xint_add_az\W\XINT_add_AB #1#2%
{%
    \XINT_add_AC_checkcarry #1%
}%
```

```
ici #2 est prévu pour l'addition, mais attention il devra être renversé pour
\numexpr. #3 = résultat partiel. #4 = chiffres qui restent. On vérifie si le
deuxième nombre s'arrête.
```

```
\def\XINT_add_AB #1#2#3#4\W\X\Y\Z #5#6#7#8%
{%
    \xint_gob_til_w #5\xint_add_bz\W
    \XINT_add_ABE #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
}%
```

```
\def\XINT_add_ABE #1#2#3#4#5#6%
{%
    \expandafter\XINT_add_ABEA\the\numexpr #1+10#5#4#3#2+#6\relax.%
}%
\def\XINT_add_ABEA #1#2#3.#4%
{%
    \XINT_add_A  #2{#3#4}%
}%
```

ici le deuxième nombre est fini #6 part à la poubelle, #2#3#4#5 est le #2 dans \XINT_add_AB on ne vérifie pas la retenue cette fois, mais les fois suivantes

```
\def\xint_add_bz\W\XINT_add_ABE #1#2#3#4#5#6%
{%
    \expandafter\XINT_add_CC\the\numexpr #1+10#5#4#3#2\relax.%
}%
\def\XINT_add_CC #1#2#3.#4%
{%
    \XINT_add_AC_checkcarry  #2{#3#4}% on va examiner et \'eliminer #2
}%
```

retenue plus chiffres qui restent de l'un des deux nombres. #2 = résultat partiel #3#4#5#6 = summand, avec plus significatif à droite

```
\def\XINT_add_AC_checkcarry #1%
{%
    \xint_gob_til_zero #1\xint_add_AC_nocarry 0\XINT_add_C
}%
\def\xint_add_AC_nocarry 0\XINT_add_C #1#2\W\X\Y\Z
{%
    \expandafter
    \xint_cleanupzeros_andstop
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
    #1%
}%
\def\XINT_add_C #1#2#3#4#5%
{%
    \xint_gob_til_w #2\xint_add_cz\W
    \XINT_add_CD {#5#4#3#2}{#1}%
}%
\def\XINT_add_CD #1%
{%
    \expandafter\XINT_add_CC\the\numexpr 1+10#1\relax.%
}%
\def\xint_add_cz\W\XINT_add_CD #1#2{ 1#2}%
```

```
Addition II: \XINT_addr_A.
INPUT: \romannumeral0\XINT_addr_A 0{}<N1>\W\X\Y\Z <N2>\W\X\Y\Z
  Comme \XINT_add_A, la différence principale c'est qu'elle donne son résul-
tat aussi *sur 4n*, renversé. De plus cette variante accepte que l'un ou même
les deux inputs soient vides. Utilisé par la sommation et par la division (pour
les quotients). Et aussi par la multiplication d'ailleurs.
INPUT: comme pour \XINT_add_A
1. <N1> et <N2> renversés
2. de longueur 4n (avec des leading zéros éventuels)
3. l'un des deux ne doit pas se terminer par 0000
OUTPUT: la somme <N1>+<N2>, *aussi renversée* et *sur 4n*
```

```
\def\XINT_addr_A #1#2#3#4#5#6%
{%
    \xint_gob_til_w #3\xint_addr_az\W
    \XINT_addr_B #1{#3#4#5#6}{#2}%
}%
\def\xint_addr_az\W\XINT_addr_B #1#2%
{%
    \XINT_addr_AC_checkcarry #1%
}%
\def\XINT_addr_B #1#2#3#4\W\X\Y\Z #5#6#7#8%
{%
    \xint_gob_til_w #5\xint_addr_bz\W
    \XINT_addr_E #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
}%
\def\XINT_addr_E #1#2#3#4#5#6%
{%
    \expandafter\XINT_addr_ABEA\the\numexpr #1+10#5#4#3#2+#6\relax
}%
\def\XINT_addr_ABEA #1#2#3#4#5#6#7%
{%
    \XINT_addr_A  #2{#7#6#5#4#3}%
}%
\def\xint_addr_bz\W\XINT_addr_E #1#2#3#4#5#6%
{%
    \expandafter\XINT_addr_CC\the\numexpr #1+10#5#4#3#2\relax
}%
\def\XINT_addr_CC #1#2#3#4#5#6#7%
{%
    \XINT_addr_AC_checkcarry  #2{#7#6#5#4#3}%
}%
\def\XINT_addr_AC_checkcarry #1%
{%
    \xint_gob_til_zero #1\xint_addr_AC_nocarry 0\XINT_addr_C
}%
\def\xint_addr_AC_nocarry 0\XINT_addr_C #1#2\W\X\Y\Z { #1#2}%
\def\XINT_addr_C #1#2#3#4#5%
{%
    \xint_gob_til_w #2\xint_addr_cz\W
```

```
    \XINT_addr_D {#5#4#3#2}{#1}%
}%
\def\XINT_addr_D #1%
{%
    \expandafter\XINT_addr_CC\the\numexpr 1+10#1\relax
}%
\def\xint_addr_cz\W\XINT_addr_D #1#2{ #21000}%
```

```
ADDITION III, \XINT_addm_A
INPUT:\romannumeral0\XINT_addm_A 0{}<N1>\W\X\Y\Z <N2>\W\X\Y\Z
1. <N1> et <N2> renversés
2. <N1> de longueur 4n ; <N2> non
3. <N2> est *garanti au moins aussi long* que <N1>
OUTPUT: la somme <N1>+<N2>, ordre normal, pas sur 4n, leading zeros retirés.
Utilisé par la multiplication.
```

```
\def\XINT_addm_A #1#2#3#4#5#6%
{%
    \xint_gob_til_w #3\xint_addm_az\W
    \XINT_addm_AB #1{#3#4#5#6}{#2}%
}%
\def\xint_addm_az\W\XINT_addm_AB #1#2%
{%
    \XINT_addm_AC_checkcarry #1%
}%
\def\XINT_addm_AB #1#2#3#4\W\X\Y\Z #5#6#7#8%
{%
    \XINT_addm_ABE #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
}%
\def\XINT_addm_ABE #1#2#3#4#5#6%
{%
    \expandafter\XINT_addm_ABEA\the\numexpr #1+10#5#4#3#2+#6\relax.%
}%
\def\XINT_addm_ABEA #1#2#3.#4%
{%
    \XINT_addm_A  #2{#3#4}%
}%
\def\XINT_addm_AC_checkcarry #1%
{%
    \xint_gob_til_zero #1\xint_addm_AC_nocarry 0\XINT_addm_C
}%
\def\xint_addm_AC_nocarry 0\XINT_addm_C #1#2\W\X\Y\Z
{%
    \expandafter
    \xint_cleanupzeros_andstop
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
```

```
      \xint_relax
    #1%
}%
\def\XINT_addm_C #1#2#3#4#5%
{%
    \xint_gob_til_w
    #5\xint_addm_cw
    #4\xint_addm_cx
    #3\xint_addm_cy
    #2\xint_addm_cz
    \W\XINT_addm_CD {#5#4#3#2}{#1}%
}%
\def\XINT_addm_CD #1%
{%
    \expandafter\XINT_addm_CC\the\numexpr 1+10#1\relax.%
}%
\def\XINT_addm_CC #1#2#3.#4%
{%
    \XINT_addm_AC_checkcarry  #2{#3#4}%
}%
\def\xint_addm_cw
    #1\xint_addm_cx
    #2\xint_addm_cy
    #3\xint_addm_cz
    \W\XINT_addm_CD
{%
    \expandafter\XINT_addm_CDw\the\numexpr 1+#1#2#3\relax.%
}%
\def\XINT_addm_CDw #1.#2#3\X\Y\Z
{%
    \XINT_addm_end #1#3%
}%
\def\xint_addm_cx
    #1\xint_addm_cy
    #2\xint_addm_cz
    \W\XINT_addm_CD
{%
    \expandafter\XINT_addm_CDx\the\numexpr 1+#1#2\relax.%
}%
\def\XINT_addm_CDx #1.#2#3\Y\Z
{%
    \XINT_addm_end #1#3%
}%
\def\xint_addm_cy
    #1\xint_addm_cz
    \W\XINT_addm_CD
{%
    \expandafter\XINT_addm_CDy\the\numexpr 1+#1\relax.%
}%
```

```
\def\XINT_addm_CDy  #1.#2#3\Z
{%
    \XINT_addm_end #1#3%
}%
\def\xint_addm_cz\W\XINT_addm_CD #1#2#3{\XINT_addm_end #1#3}%
\def\XINT_addm_end #1#2#3#4#5%
    {\expandafter\space\the\numexpr #1#2#3#4#5\relax}%
```

```
ADDITION IV, variante \XINT_addp_A
INPUT: \romannumeral0\XINT_addp_A 0{}<N1>\W\X\Y\Z <N2>\W\X\Y\Z
1. <N1> et <N2> renversés
2. <N1> de longueur 4n ; <N2> non
3. <N2> est *garanti au moins aussi long* que <N1>
OUTPUT: la somme <N1>+<N2>, dans l'ordre renversé, sur 4n, et en faisant at-
tention de ne pas terminer en 0000. Utilisé par la multiplication servant pour
le calcul des puissances.
```

```
\def\XINT_addp_A #1#2#3#4#5#6%
{%
    \xint_gob_til_w #3\xint_addp_az\W
    \XINT_addp_AB #1{#3#4#5#6}{#2}%
}%
\def\xint_addp_az\W\XINT_addp_AB #1#2%
{%
    \XINT_addp_AC_checkcarry #1%
}%
\def\XINT_addp_AC_checkcarry #1%
{%
    \xint_gob_til_zero #1\xint_addp_AC_nocarry 0\XINT_addp_C
}%
\def\xint_addp_AC_nocarry 0\XINT_addp_C
{%
    \XINT_addp_F
}%
\def\XINT_addp_AB #1#2#3#4\W\X\Y\Z #5#6#7#8%
{%
    \XINT_addp_ABE #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
}%
\def\XINT_addp_ABE #1#2#3#4#5#6%
{%
    \expandafter\XINT_addp_ABEA\the\numexpr #1+10#5#4#3#2+#6\relax
}%
\def\XINT_addp_ABEA #1#2#3#4#5#6#7%
{%
   \XINT_addp_A  #2{#7#6#5#4#3}%<-- attention on met donc \'a droite
}%
\def\XINT_addp_C #1#2#3#4#5%
{%
    \xint_gob_til_w
    #5\xint_addp_cw
```

```
    #4\xint_addp_cx
    #3\xint_addp_cy
    #2\xint_addp_cz
    \W\XINT_addp_CD   {#5#4#3#2}{#1}%
}%
\def\XINT_addp_CD #1%
{%
    \expandafter\XINT_addp_CC\the\numexpr 1+10#1\relax
}%
\def\XINT_addp_CC #1#2#3#4#5#6#7%
{%
    \XINT_addp_AC_checkcarry  #2{#7#6#5#4#3}%
}%
\def\xint_addp_cw
    #1\xint_addp_cx
    #2\xint_addp_cy
    #3\xint_addp_cz
    \W\XINT_addp_CD
{%
    \expandafter\XINT_addp_CDw\the\numexpr 1+10#1#2#3\relax
}%
\def\XINT_addp_CDw #1#2#3#4#5#6%
{%
    \xint_gob_til_zeros_iv #2#3#4#5\XINT_addp_endDw_zeros
                           0000\XINT_addp_endDw #2#3#4#5%
}%
\def\XINT_addp_endDw_zeros 0000\XINT_addp_endDw 0000#1\X\Y\Z{ #1}%
\def\XINT_addp_endDw #1#2#3#4#5\X\Y\Z{ #5#4#3#2#1}%
\def\xint_addp_cx
    #1\xint_addp_cy
    #2\xint_addp_cz
    \W\XINT_addp_CD
{%
    \expandafter\XINT_addp_CDx\the\numexpr 1+100#1#2\relax
}%
\def\XINT_addp_CDx #1#2#3#4#5#6%
{%
    \xint_gob_til_zeros_iv #2#3#4#5\XINT_addp_endDx_zeros
                           0000\XINT_addp_endDx #2#3#4#5%
}%
\def\XINT_addp_endDx_zeros 0000\XINT_addp_endDx 0000#1\Y\Z{ #1}%
\def\XINT_addp_endDx #1#2#3#4#5\Y\Z{ #5#4#3#2#1}%
\def\xint_addp_cy
    #1\xint_addp_cz
    \W\XINT_addp_CD
{%
    \expandafter\XINT_addp_CDy\the\numexpr 1+1000#1\relax
}%
\def\XINT_addp_CDy #1#2#3#4#5#6%
```

```
{%
    \xint_gob_til_zeros_iv #2#3#4#5\XINT_addp_endDy_zeros
                           0000\XINT_addp_endDy #2#3#4#5%
}%
\def\XINT_addp_endDy_zeros 0000\XINT_addp_endDy 0000#1\Z{ #1}%
\def\XINT_addp_endDy #1#2#3#4#5\Z{ #5#4#3#2#1}%
\def\xint_addp_cz\W\XINT_addp_CD #1#2{ #21000}%
\def\XINT_addp_F #1#2#3#4#5%
{%
    \xint_gob_til_w
    #5\xint_addp_Gw
    #4\xint_addp_Gx
    #3\xint_addp_Gy
    #2\xint_addp_Gz
    \W\XINT_addp_G   {#2#3#4#5}{#1}%
}%
\def\XINT_addp_G #1#2%
{%
    \XINT_addp_F {#2#1}%
}%
\def\xint_addp_Gw
    #1\xint_addp_Gx
    #2\xint_addp_Gy
    #3\xint_addp_Gz
    \W\XINT_addp_G #4%
{%
    \xint_gob_til_zeros_iv  #3#2#10\XINT_addp_endGw_zeros
                            0000\XINT_addp_endGw #3#2#10%
}%
\def\XINT_addp_endGw_zeros 0000\XINT_addp_endGw 0000#1\X\Y\Z{ #1}%
\def\XINT_addp_endGw #1#2#3#4#5\X\Y\Z{ #5#1#2#3#4}%
\def\xint_addp_Gx
    #1\xint_addp_Gy
    #2\xint_addp_Gz
    \W\XINT_addp_G #3%
{%
    \xint_gob_til_zeros_iv  #2#100\XINT_addp_endGx_zeros
                           0000\XINT_addp_endGx #2#100%
}%
\def\XINT_addp_endGx_zeros 0000\XINT_addp_endGx 0000#1\Y\Z{ #1}%
\def\XINT_addp_endGx #1#2#3#4#5\Y\Z{ #5#1#2#3#4}%
\def\xint_addp_Gy
    #1\xint_addp_Gz
    \W\XINT_addp_G #2%
{%
    \xint_gob_til_zeros_iv   #1000\XINT_addp_endGy_zeros
                           0000\XINT_addp_endGy #1000%
}%
\def\XINT_addp_endGy_zeros 0000\XINT_addp_endGy 0000#1\Z{ #1}%
```

```
\def\XINT_addp_endGy #1#2#3#4#5\Z{ #5#1#2#3#4}%
\def\xint_addp_Gz\W\XINT_addp_G #1#2{ #2}%
```

19.20 \xintAdd

```
\def\xintiAdd {\romannumeral0\xintiadd }%
\def\xintiadd #1%
{%
    \expandafter\xint_add\expandafter{\romannumeral-`0#1}%
}%
\let\xintAdd\xintiAdd \let\xintadd\xintiadd
\def\xint_add #1#2%
{%
    \expandafter\XINT_add_fork \romannumeral-`0#2\Z #1\Z
}%
\def\XINT_Add #1#2{\romannumeral0\XINT_add_fork #2\Z #1\Z }%
\def\XINT_add #1#2{\XINT_add_fork #2\Z #1\Z }%
```

```
ADDITION Ici #1#2 vient du *deuxième* argument de \xintAdd et #3#4 donc du *pre-
mier* [algo plus efficace lorsque le premier est plus long que le second]
```

```
\def\XINT_add_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\dummy \XINT_add_secondiszero
      #3\dummy \XINT_add_firstiszero
       0\dummy
        {\xint_UDsignsfork
          #1#3\dummy \XINT_add_minusminus          % #1 = #3 = -
           #1-\dummy \XINT_add_minusplus           % #1 = -
           #3-\dummy \XINT_add_plusminus           % #3 = -
            --\dummy \XINT_add_plusplus
         \krof }%
    \krof
    {#2}{#4}#1#3%
}%
\def\XINT_add_secondiszero #1#2#3#4{ #4#2}%
\def\XINT_add_firstiszero #1#2#3#4{ #3#1}%
```

```
#1 vient du *deuxième* et #2 vient du *premier*
```

```
\def\XINT_add_minusminus #1#2#3#4%
{%
    \expandafter\xint_minus_andstop%
    \romannumeral0\XINT_add_pre {#2}{#1}%
}%
\def\XINT_add_minusplus #1#2#3#4%
{%
    \XINT_sub_pre {#4#2}{#1}%
}%
\def\XINT_add_plusminus #1#2#3#4%
```

```
{%
    \XINT_sub_pre {#3#1}{#2}%
}%
\def\XINT_add_plusplus #1#2#3#4%
{%
    \XINT_add_pre {#4#2}{#3#1}%
}%
\def\XINT_add_pre #1%
{%
  \expandafter\XINT_add__pre\expandafter
  {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z }%
}%
\def\XINT_add__pre #1#2%
{%
    \expandafter\XINT_add_A
        \expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
        \W\X\Y\Z #1\W\X\Y\Z
}%
```

19.21 \xintSub

```
\def\xintiSub {\romannumeral0\xintisub }%
\def\xintisub #1%
{%
    \expandafter\xint_sub\expandafter{\romannumeral-`0#1}%
}%
\let\xintSub\xintiSub \let\xintsub\xintisub
\def\xint_sub #1#2%
{%
    \expandafter\XINT_sub_fork \romannumeral-`0#2\Z #1\Z
}%
\def\XINT_Sub #1#2{\romannumeral0\XINT_sub_fork #2\Z #1\Z }%
\def\XINT_sub #1#2{\XINT_sub_fork #2\Z #1\Z }%
```

```
 SOUSTRACTION #3#4-#1#2: #3#4 vient du *premier* #1#2 vient du *second*
```

```
\def\XINT_sub_fork #1#2\Z #3#4\Z
{%
    \xint_UDsignsfork
          #1#3\dummy \XINT_sub_minusminus
           #1-\dummy \XINT_sub_minusplus   % attention, #3=0 possible
           #3-\dummy \XINT_sub_plusminus   % attention, #1=0 possible
            --\dummy {\xint_UDzerofork
                      #1\dummy \XINT_sub_secondiszero
                      #3\dummy \XINT_sub_firstiszero
                       0\dummy \XINT_sub_plusplus
                      \krof }%
    \krof
    {#2}{#4}#1#3%
}%
```

```
\def\XINT_sub_secondiszero #1#2#3#4{ #4#2}%
\def\XINT_sub_firstiszero  #1#2#3#4{ -#3#1}%
\def\XINT_sub_plusplus #1#2#3#4%
{%
    \XINT_sub_pre {#4#2}{#3#1}%
}%
\def\XINT_sub_minusminus #1#2#3#4%
{%
    \XINT_sub_pre {#1}{#2}%
}%
\def\XINT_sub_minusplus #1#2#3#4%
{%
    \xint_gob_til_zero #4\xint_sub_mp0\XINT_add_pre {#4#2}{#1}%
}%
\def\xint_sub_mp0\XINT_add_pre #1#2{ #2}%
\def\XINT_sub_plusminus #1#2#3#4%
{%
    \xint_gob_til_zero #3\xint_sub_pm0\expandafter\xint_minus_andstop%
    \romannumeral0\XINT_add_pre {#2}{#3#1}%
}%
\def\xint_sub_pm #1\XINT_add_pre #2#3{ -#2}%
\def\XINT_sub_pre #1%
{%
  \expandafter\XINT_sub__pre\expandafter
  {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z }%
}%
\def\XINT_sub__pre #1#2%
{%
    \expandafter\XINT_sub_A
        \expandafter1\expandafter{\expandafter}%
    \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
        \W\X\Y\Z #1 \W\X\Y\Z
}%
```

```
\romannumeral0\XINT_sub_A 1{}<N1>\W\X\Y\Z<N2>\W\X\Y\Z
N1 et N2 sont présentés à l'envers ET ON A RAJOUTÉ DES ZÉROS POUR QUE LEURS LONGUEURS
À CHACUN SOIENT MULTIPLES DE 4, MAIS AUCUN NE SE TERMINE EN 0000.
output: N2 - N1
Elle donne le résultat dans le **bon ordre**, avec le bon signe, et sans zéros
superflus.
```

```
\def\XINT_sub_A #1#2#3\W\X\Y\Z #4#5#6#7%
{%
    \xint_gob_til_w
    #4\xint_sub_az
    \W\XINT_sub_B #1{#4#5#6#7}{#2}#3\W\X\Y\Z
}%
\def\XINT_sub_B #1#2#3#4#5#6#7%
{%
    \xint_gob_til_w
```

```
1153     #4\xint_sub_bz
1154     \W\XINT_sub_onestep #1#2{#7#6#5#4}{#3}%
1155 }%
```

d'abord la branche principale #6 = 4 chiffres de N1, plus significatif en *premier*, #2#3#4#5 chiffres de N2, plus significatif en *dernier* On veut N2 - N1.

```
1156 \def\XINT_sub_onestep #1#2#3#4#5#6%
1157 {%
1158     \expandafter\XINT_sub_backtoA\the\numexpr 11#5#4#3#2-#6+#1-1\relax.%
1159 }%
```

ON PRODUIT LE RÉSULTAT DANS LE BON ORDRE

```
1160 \def\XINT_sub_backtoA #1#2#3.#4%
1161 {%
1162     \XINT_sub_A #2{#3#4}%
1163 }%
1164 \def\xint_sub_bz
1165     \W\XINT_sub_onestep #1#2#3#4#5#6#7%
1166 {%
1167     \xint_UDzerofork
1168       #1\dummy  \XINT_sub_C   % une retenue
1169        0\dummy  \XINT_sub_D   % pas de retenue
1170     \krof
1171     {#7}#2#3#4#5%
1172 }%
1173 \def\XINT_sub_D #1#2\W\X\Y\Z
1174 {%
1175     \expandafter
1176     \xint_cleanupzeros_andstop
1177     \romannumeral0%
1178     \XINT_rord_main {}#2%
1179       \xint_relax
1180         \xint_undef\xint_undef\xint_undef\xint_undef
1181         \xint_undef\xint_undef\xint_undef\xint_undef
1182       \xint_relax
1183     #1%
1184 }%
1185 \def\XINT_sub_C #1#2#3#4#5%
1186 {%
1187     \xint_gob_til_w
1188     #2\xint_sub_cz
1189     \W\XINT_sub_AC_onestep {#5#4#3#2}{#1}%
1190 }%
1191 \def\XINT_sub_AC_onestep #1%
1192 {%
1193     \expandafter\XINT_sub_backtoC\the\numexpr 11#1-1\relax.%
1194 }%
1195 \def\XINT_sub_backtoC #1#2#3.#4%
1196 {%
```

```
    \XINT_sub_AC_checkcarry #2{#3#4}% la retenue va \^etre examin\'ee
}%
\def\XINT_sub_AC_checkcarry #1%
{%
    \xint_gob_til_one #1\xint_sub_AC_nocarry 1\XINT_sub_C
}%
\def\xint_sub_AC_nocarry 1\XINT_sub_C #1#2\W\X\Y\Z
{%
    \expandafter
    \XINT_cuz_loop
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
    #1\W\W\W\W\W\W\W\Z
}%
\def\xint_sub_cz\W\XINT_sub_AC_onestep #1%
{%
    \XINT_cuz
}%
\def\xint_sub_az\W\XINT_sub_B #1#2#3#4#5#6#7%
{%
    \xint_gob_til_w
    #4\xint_sub_ez
    \W\XINT_sub_Eenter #1{#3}#4#5#6#7%
}%
```

le premier nombre continue, le résultat sera < 0.

```
\def\XINT_sub_Eenter #1#2%
{%
    \expandafter
    \XINT_sub_E\expandafter1\expandafter{\expandafter}%
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
    \W\X\Y\Z #1%
}%
\def\XINT_sub_E #1#2#3#4#5#6%
{%
    \xint_gob_til_w #3\xint_sub_F\W
    \XINT_sub_Eonestep #1{#6#5#4#3}{#2}%
}%
\def\XINT_sub_Eonestep #1#2%
{%
```

```
    \expandafter\XINT_sub_backtoE\the\numexpr 109999-#2+#1\relax.%
}%
\def\XINT_sub_backtoE #1#2#3.#4%
{%
    \XINT_sub_E #2{#3#4}%
}%
\def\xint_sub_F\W\XINT_sub_Eonestep #1#2#3#4%
{%
    \xint_UDonezerofork
      #4#1\dummy {\XINT_sub_Fdec 0}% soustraire  1. Et faire signe -
      #1#4\dummy {\XINT_sub_Finc 1}% additionner 1. Et faire signe -
        10\dummy  \XINT_sub_DD     % terminer. Mais avec signe -
    \krof
    {#3}%
}%
\def\XINT_sub_DD {\expandafter\xint_minus_andstop\romannumeral0\XINT_sub_D }%
\def\XINT_sub_Fdec #1#2#3#4#5#6%
{%
    \xint_gob_til_w #3\xint_sub_Fdec_finish\W
    \XINT_sub_Fdec_onestep #1{#6#5#4#3}{#2}%
}%
\def\XINT_sub_Fdec_onestep #1#2%
{%
    \expandafter\XINT_sub_backtoFdec\the\numexpr 11#2+#1-1\relax.%
}%
\def\XINT_sub_backtoFdec #1#2#3.#4%
{%
    \XINT_sub_Fdec #2{#3#4}%
}%
\def\xint_sub_Fdec_finish\W\XINT_sub_Fdec_onestep #1#2%
{%
    \expandafter\xint_minus_andstop\romannumeral0\XINT_cuz
}%
\def\XINT_sub_Finc #1#2#3#4#5#6%
{%
    \xint_gob_til_w #3\xint_sub_Finc_finish\W
    \XINT_sub_Finc_onestep #1{#6#5#4#3}{#2}%
}%
\def\XINT_sub_Finc_onestep #1#2%
{%
    \expandafter\XINT_sub_backtoFinc\the\numexpr 10#2+#1\relax.%
}%
\def\XINT_sub_backtoFinc #1#2#3.#4%
{%
    \XINT_sub_Finc #2{#3#4}%
}%
\def\xint_sub_Finc_finish\W\XINT_sub_Finc_onestep #1#2#3%
{%
    \xint_UDzerofork
```

```
      #1\dummy {\expandafter\xint_minus_andstop\xint_cleanupzeros_nospace}%
       0\dummy { -1}%
    \krof
    #3%
}%
\def\xint_sub_ez\W\XINT_sub_Eenter #1%
{%
    \xint_UDzerofork
      #1\dummy \XINT_sub_K % il y a une retenue
       0\dummy \XINT_sub_L % pas de retenue
    \krof
}%
\def\XINT_sub_L #1\W\X\Y\Z {\XINT_cuz_loop #1\W\W\W\W\W\W\W\Z }%
\def\XINT_sub_K #1%
{%
    \expandafter
    \XINT_sub_KK\expandafter1\expandafter{\expandafter}%
    \romannumeral0%
    \XINT_rord_main {}#1%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
}%
\def\XINT_sub_KK #1#2#3#4#5#6%
{%
    \xint_gob_til_w #3\xint_sub_KK_finish\W
    \XINT_sub_KK_onestep #1{#6#5#4#3}{#2}%
}%
\def\XINT_sub_KK_onestep #1#2%
{%
    \expandafter\XINT_sub_backtoKK\the\numexpr 109999-#2+#1\relax.%
}%
\def\XINT_sub_backtoKK #1#2#3.#4%
{%
    \XINT_sub_KK #2{#3#4}%
}%
\def\xint_sub_KK_finish\W\XINT_sub_KK_onestep #1#2#3%
{%
    \expandafter\xint_minus_andstop
    \romannumeral0\XINT_cuz_loop #3\W\W\W\W\W\W\W\Z
}%
```

19.22 \xintCmp

```
\def\xintiCmp {\romannumeral0\xinticmp }%
\def\xinticmp #1%
{%
    \expandafter\xint_cmp\expandafter{\romannumeral-`0#1}%
}%
```

```
\let\xintCmp\xintiCmp \let\xintcmp\xinticmp
\def\xint_cmp #1#2%
{%
    \expandafter\XINT_cmp_fork \romannumeral-`0#2\Z #1\Z
}%
\def\XINT_Cmp #1#2{\romannumeral0\XINT_cmp_fork #2\Z #1\Z }%

 COMPARAISON
 1 si #3#4>#1#2, 0 si #3#4=#1#2, -1 si #3#4<#1#2
 #3#4 vient du *premier*, #1#2 vient du *second*
\def\XINT_cmp_fork #1#2\Z #3#4\Z
{%
    \xint_UDsignsfork
          #1#3\dummy \XINT_cmp_minusminus
           #1-\dummy \XINT_cmp_minusplus
           #3-\dummy \XINT_cmp_plusminus
            --\dummy {\xint_UDzerosfork
                      #1#3\dummy \XINT_cmp_zerozero
                       #10\dummy \XINT_cmp_zeroplus
                       #30\dummy \XINT_cmp_pluszero
                        00\dummy \XINT_cmp_plusplus
                      \krof }%
    \krof
    {#2}{#4}#1#3%
}%
\def\XINT_cmp_minusplus #1#2#3#4{ 1}%
\def\XINT_cmp_plusminus #1#2#3#4{ -1}%
\def\XINT_cmp_zerozero  #1#2#3#4{ 0}%
\def\XINT_cmp_zeroplus  #1#2#3#4{ 1}%
\def\XINT_cmp_pluszero  #1#2#3#4{ -1}%
\def\XINT_cmp_plusplus #1#2#3#4%
{%
    \XINT_cmp_pre {#4#2}{#3#1}%
}%
\def\XINT_cmp_minusminus #1#2#3#4%
{%
    \XINT_cmp_pre {#1}{#2}%
}%
\def\XINT_cmp_pre #1%
{%
  \expandafter\XINT_cmp__pre\expandafter
  {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z }%
}%
\def\XINT_cmp__pre #1#2%
{%
    \expandafter\XINT_cmp_A
    \expandafter1\expandafter{\expandafter}%
    \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
        \W\X\Y\Z #1\W\X\Y\Z
```

```
}%

 COMPARAISON
 N1 et N2 sont présentés à l'envers ET ON A RAJOUTÉ DES ZÉROS POUR QUE LEUR LONGUEURS
 À CHACUN SOIENT MULTIPLES DE 4, MAIS AUCUN NE SE TERMINE EN 0000. routine ap-
 pelée via
 \XINT_cmp_A 1{}<N1>\W\X\Y\Z<N2>\W\X\Y\Z
 ATTENTION RENVOIE 1 SI N1 < N2, 0 si N1 = N2, -1 si N1 > N2
\def\XINT_cmp_A #1#2#3\W\X\Y\Z #4#5#6#7%
{%
    \xint_gob_til_w #4\xint_cmp_az\W
    \XINT_cmp_B #1{#4#5#6#7}{#2}#3\W\X\Y\Z
}%
\def\XINT_cmp_B #1#2#3#4#5#6#7%
{%
    \xint_gob_til_w#4\xint_cmp_bz\W
    \XINT_cmp_onestep #1#2{#7#6#5#4}{#3}%
}%
\def\XINT_cmp_onestep #1#2#3#4#5#6%
{%
    \expandafter\XINT_cmp_backtoA\the\numexpr 11#5#4#3#2-#6+#1-1\relax.%
}%
\def\XINT_cmp_backtoA #1#2#3.#4%
{%
    \XINT_cmp_A #2{#3#4}%
}%
\def\xint_cmp_bz\W\XINT_cmp_onestep #1\Z { 1}%
\def\xint_cmp_az\W\XINT_cmp_B #1#2#3#4#5#6#7%
{%
    \xint_gob_til_w #4\xint_cmp_ez\W
    \XINT_cmp_Eenter #1{#3}#4#5#6#7%
}%
\def\XINT_cmp_Eenter #1\Z { -1}%
\def\xint_cmp_ez\W\XINT_cmp_Eenter #1%
{%
    \xint_UDzerofork
      #1\dummy \XINT_cmp_K             %     il y a une retenue
       0\dummy \XINT_cmp_L             %     pas de retenue
    \krof
}%
\def\XINT_cmp_K #1\Z { -1}%
\def\XINT_cmp_L #1{\XINT_OneIfPositive_main #1}%
\def\XINT_OneIfPositive #1%
{%
    \XINT_OneIfPositive_main #1\W\X\Y\Z%
}%
\def\XINT_OneIfPositive_main #1#2#3#4%
{%
    \xint_gob_til_z #4\xint_OneIfPositive_terminated\Z
```

```
    \XINT_OneIfPositive_onestep #1#2#3#4%
}%
\def\xint_OneIfPositive_terminated\Z\XINT_OneIfPositive_onestep\W\X\Y\Z { 0}%
\def\XINT_OneIfPositive_onestep #1#2#3#4%
{%
    \expandafter\XINT_OneIfPositive_check\the\numexpr #1#2#3#4\relax
}%
\def\XINT_OneIfPositive_check #1%
{%
    \xint_gob_til_zero #1\xint_OneIfPositive_backtomain 0%
    \XINT_OneIfPositive_finish #1%
}%
\def\XINT_OneIfPositive_finish #1\W\X\Y\Z{ 1}%
\def\xint_OneIfPositive_backtomain 0\XINT_OneIfPositive_finish 0%
                    {\XINT_OneIfPositive_main }%
```

19.23 \xintGeq

```
 PLUS GRAND OU ÉGAL attention compare les **valeurs absolues**
```

```
\def\xintiGeq {\romannumeral0\xintigeq }%
\def\xintigeq #1%
{%
    \expandafter\xint_geq\expandafter {\romannumeral-`0#1}%
}%
\let\xintGeq\xintiGeq \let\xintgeq\xintigeq
\def\xint_geq #1#2%
{%
    \expandafter\XINT_geq_fork \romannumeral-`0#2\Z #1\Z
}%
\def\XINT_Geq #1#2{\romannumeral0\XINT_geq_fork #2\Z #1\Z }%
```

```
 PLUS GRAND OU ÉGAL ATTENTION, TESTE les VALEURS ABSOLUES
```

```
\def\XINT_geq_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\dummy \XINT_geq_secondiszero % |#1#2|=0
      #3\dummy \XINT_geq_firstiszero % |#1#2|>0
       0\dummy {\xint_UDsignsfork
                  #1#3\dummy \XINT_geq_minusminus
                   #1-\dummy \XINT_geq_minusplus
                   #3-\dummy \XINT_geq_plusminus
                    --\dummy \XINT_geq_plusplus
                \krof }%
    \krof
    {#2}{#4}#1#3%
}%
\def\XINT_geq_secondiszero     #1#2#3#4{ 1}%
\def\XINT_geq_firstiszero      #1#2#3#4{ 0}%
```

```
\def\XINT_geq_plusplus   #1#2#3#4{\XINT_geq_pre {#4#2}{#3#1}}%
\def\XINT_geq_minusminus #1#2#3#4{\XINT_geq_pre {#2}{#1}}%
\def\XINT_geq_minusplus  #1#2#3#4{\XINT_geq_pre {#4#2}{#1}}%
\def\XINT_geq_plusminus  #1#2#3#4{\XINT_geq_pre {#2}{#3#1}}%
\def\XINT_geq_pre #1%
{%
  \expandafter\XINT_geq__pre\expandafter
  {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\R\Z }%
}%
\def\XINT_geq__pre #1#2%
{%
    \expandafter\XINT_geq_A
    \expandafter1\expandafter{\expandafter}%
    \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\R\Z
        \W\X\Y\Z #1 \W\X\Y\Z
}%
```

```
 PLUS GRAND OU ÉGAL
 N1 et N2 sont présentés à l'envers ET ON A RAJOUTÉ DES ZÉROS POUR QUE LEURS LONGUEURS
 À CHACUN SOIENT MULTIPLES DE 4, MAIS AUCUN NE SE TERMINE EN 0000
 routine appelée via
 \romannumeral0\XINT_geq_A 1{}<N1>\W\X\Y\Z<N2>\W\X\Y\Z
 ATTENTION RENVOIE 1 SI N1 < N2 ou N1 = N2 et 0 si N1 > N2
```

```
\def\XINT_geq_A #1#2#3\W\X\Y\Z #4#5#6#7%
{%
    \xint_gob_til_w #4\xint_geq_az\W
    \XINT_geq_B #1{#4#5#6#7}{#2}#3\W\X\Y\Z
}%
\def\XINT_geq_B #1#2#3#4#5#6#7%
{%
    \xint_gob_til_w #4\xint_geq_bz\W
    \XINT_geq_onestep #1#2{#7#6#5#4}{#3}%
}%
\def\XINT_geq_onestep #1#2#3#4#5#6%
{%
    \expandafter\XINT_geq_backtoA\the\numexpr 11#5#4#3#2-#6+#1-1\relax.%
}%
\def\XINT_geq_backtoA #1#2#3.#4%
{%
    \XINT_geq_A #2{#3#4}%
}%
\def\xint_geq_bz\W\XINT_geq_onestep #1\W\X\Y\Z { 1}%
\def\xint_geq_az\W\XINT_geq_B #1#2#3#4#5#6#7%
{%
    \xint_gob_til_w #4\xint_geq_ez\W
    \XINT_geq_Eenter #1%
}%
\def\XINT_geq_Eenter #1\W\X\Y\Z { 0}%
\def\xint_geq_ez\W\XINT_geq_Eenter #1%
```

```
{%
    \xint_UDzerofork
      #1\dummy { 0}             %     il y a une retenue
       0\dummy { 1}             %     pas de retenue
    \krof
}%
```

19.24 \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.

```
\def\xintiMax {\romannumeral0\xintimax }%
\def\xintimax #1%
{%
    \expandafter\xint_max\expandafter {\romannumeral-`0#1}%
}%
\let\xintMax\xintiMax \let\xintmax\xintimax
\def\xint_max #1#2%
{%
    \expandafter\XINT_max_pre\expandafter {\romannumeral-`0#2}{#1}%
}%
\def\XINT_max_pre #1#2{\XINT_max_fork #1\Z #2\Z {#2}{#1}}%
\def\XINT_Max #1#2{\romannumeral0\XINT_max_fork #2\Z #1\Z {#1}{#2}}%
```

#3#4 vient du *premier*, #1#2 vient du *second*

```
\def\XINT_max_fork #1#2\Z #3#4\Z
{%
    \xint_UDsignsfork
          #1#3\dummy \XINT_max_minusminus  % A < 0, B < 0
           #1-\dummy \XINT_max_minusplus   % B < 0, A >= 0
           #3-\dummy \XINT_max_plusminus   % A < 0, B >= 0
            --\dummy {\xint_UDzerosfork
                      #1#3\dummy \XINT_max_zerozero % A = B = 0
                       #10\dummy \XINT_max_zeroplus % B = 0, A > 0
                       #30\dummy \XINT_max_pluszero % A = 0, B > 0
                        00\dummy \XINT_max_plusplus % A, B > 0
                      \krof }%
    \krof
    {#2}{#4}#1#3%
}%
```

A = #4#2, B = #3#1

```
\def\XINT_max_zerozero  #1#2#3#4{\xint_firstoftwo_andstop }%
\def\XINT_max_zeroplus  #1#2#3#4{\xint_firstoftwo_andstop }%
\def\XINT_max_pluszero  #1#2#3#4{\xint_secondoftwo_andstop }%
\def\XINT_max_minusplus #1#2#3#4{\xint_firstoftwo_andstop }%
```

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

```
 #3=-, #4=-, #1 = |B| = -B, #2 = |A| = -A
```

```
\def\XINT_max_minusminus #1#2#3#4%
{%
    \ifodd\XINT_Geq {#1}{#2}
      \expandafter\xint_firstoftwo_andstop
    \else
      \expandafter\xint_secondoftwo_andstop
    \fi
}%
```

19.25 \xintMin

```
\def\xintiMin {\romannumeral0\xintimin }%
\def\xintimin #1%
{%
    \expandafter\xint_min\expandafter {\romannumeral-`0#1}%
}%
\let\xintMin\xintiMin \let\xintmin\xintimin
\def\xint_min #1#2%
{%
    \expandafter\XINT_min_pre\expandafter {\romannumeral-`0#2}{#1}%
}%
\def\XINT_min_pre #1#2{\XINT_min_fork #1\Z #2\Z {#2}{#1}}%
\def\XINT_Min #1#2{\romannumeral0\XINT_min_fork #2\Z #1\Z {#1}{#2}}%
```

```
 #3#4 vient du *premier*, #1#2 vient du *second*
```

```
\def\XINT_min_fork #1#2\Z #3#4\Z
{%
    \xint_UDsignsfork
          #1#3\dummy \XINT_min_minusminus  % A < 0, B < 0
           #1-\dummy \XINT_min_minusplus   % B < 0, A >= 0
           #3-\dummy \XINT_min_plusminus   % A < 0, B >= 0
            --\dummy {\xint_UDzerosfork
                      #1#3\dummy \XINT_min_zerozero % A = B = 0
                       #10\dummy \XINT_min_zeroplus % B = 0, A > 0
                       #30\dummy \XINT_min_pluszero % A = 0, B > 0
                        00\dummy \XINT_min_plusplus % A, B > 0
                      \krof }%
```

```
1589     \krof
1590     {#2}{#4}#1#3%
1591 }%
```

```
A = #4#2, B = #3#1
```

```
1592 \def\XINT_min_zerozero  #1#2#3#4{\xint_firstoftwo_andstop }%
1593 \def\XINT_min_zeroplus  #1#2#3#4{\xint_secondoftwo_andstop }%
1594 \def\XINT_min_pluszero  #1#2#3#4{\xint_firstoftwo_andstop }%
1595 \def\XINT_min_minusplus #1#2#3#4{\xint_secondoftwo_andstop }%
1596 \def\XINT_min_plusminus #1#2#3#4{\xint_firstoftwo_andstop }%
1597 \def\XINT_min_plusplus  #1#2#3#4%
1598 {%
1599     \ifodd\XINT_Geq {#4#2}{#3#1}
1600       \expandafter\xint_secondoftwo_andstop
1601     \else
1602       \expandafter\xint_firstoftwo_andstop
1603     \fi
1604 }%
```

```
#3=-, #4=-, #1 = |B| = -B, #2 = |A| = -A
```

```
1605 \def\XINT_min_minusminus #1#2#3#4%
1606 {%
1607     \ifodd\XINT_Geq {#1}{#2}
1608       \expandafter\xint_secondoftwo_andstop
1609     \else
1610       \expandafter\xint_firstoftwo_andstop
1611     \fi
1612 }%
```

19.26 \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 possi-
ble).
```

```
1613 \def\xintiSum {\romannumeral0\xintisum }%
1614 \def\xintisum #1{\xintisumexpr #1\relax }%
1615 \def\xintiSumExpr {\romannumeral0\xintisumexpr }%
1616 \def\xintisumexpr {\expandafter\XINT_sumexpr\romannumeral-`0}%
1617 \let\xintSum\xintiSum \let\xintsum\xintisum
1618 \let\xintSumExpr\xintiSumExpr \let\xintsumexpr\xintisumexpr
1619 \def\XINT_sumexpr {\XINT_sum_loop {0000}{0000}}%
1620 \def\XINT_sum_loop #1#2#3%
1621 {%
1622     \expandafter\XINT_sum_checksign\romannumeral-`0#3\Z {#1}{#2}%
```

```
}%
\def\XINT_sum_checksign #1%
{%
    \xint_gob_til_relax #1\XINT_sum_finished\relax
    \xint_gob_til_zero #1\XINT_sum_skipzeroinput0%
    \xint_UDsignfork
      #1\dummy \XINT_sum_N
       -\dummy {\XINT_sum_P #1}%
    \krof
}%
\def\XINT_sum_finished #1\Z #2#3%
{%
    \XINT_sub_A 1{}#3\W\X\Y\Z #2\W\X\Y\Z
}%
\def\XINT_sum_skipzeroinput #1\krof #2\Z {\XINT_sum_loop }%
\def\XINT_sum_P #1\Z #2%
{%
    \expandafter\XINT_sum_loop\expandafter
    {\romannumeral0\expandafter
     \XINT_addr_A\expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\Z
    \W\X\Y\Z #2\W\X\Y\Z }%
}%
\def\XINT_sum_N #1\Z #2#3%
{%
    \expandafter\XINT_sum_NN\expandafter
    {\romannumeral0\expandafter
     \XINT_addr_A\expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\Z
    \W\X\Y\Z #3\W\X\Y\Z }{#2}%
}%
\def\XINT_sum_NN #1#2{\XINT_sum_loop {#2}{#1}}%
```

19.27 \xintMul

```
\def\xintiMul {\romannumeral0\xintimul }%
\def\xintimul #1%
{%
    \expandafter\xint_mul\expandafter {\romannumeral-`0#1}%
}%
\let\xintMul\xintiMul \let\xintmul\xintimul
\def\xint_mul #1#2%
{%
    \expandafter\XINT_mul_fork \romannumeral-`0#2\Z #1\Z
}%
\def\XINT_Mul #1#2{\romannumeral0\XINT_mul_fork #2\Z #1\Z }%
```

```
MULTIPLICATION
Ici #1#2 = 2e input et #3#4 = 1er input
```

Release 1.03 adds some overhead to first compute and compare the lengths of the two inputs. The algorithm is asymmetrical and whether the first input is the longest or the shortest sometimes has a strong impact. 50 digits times 1000 digits used to be 5 times faster than 1000 digits times 50 digits. With the new code, the user input order does not matter as it is decided by the routine what is best. This is important for the extension to fractions, as there is no way then to generally control or guess the most frequent sizes of the inputs besides actually computing their lengths.

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

```
}%
\def\XINT_mul_choice_a #1#2%
{%
    \expandafter\XINT_mul_choice_b\expandafter{#2}{#1}%
}%
\def\XINT_mul_choice_b #1#2%
{%
    \ifnum #1<5
       \expandafter\XINT_mul_choice_littlebyfirst
    \else
    \ifnum #2<5
      \expandafter\expandafter\expandafter\XINT_mul_choice_littlebysecond
      \else
      \expandafter\expandafter\expandafter\XINT_mul_choice_compare
      \fi
    \fi
    {#1}{#2}%
}%
\def\XINT_mul_choice_littlebyfirst #1#2#3#4%
{%
    \expandafter\XINT_mul_M
    \expandafter{\the\numexpr #3\expandafter}%
    \romannumeral0\XINT_RQ {}#4\R\R\R\R\R\R\R\Z \Z\Z\Z\Z
}%
\def\XINT_mul_choice_littlebysecond #1#2#3#4%
{%
    \expandafter\XINT_mul_M
    \expandafter{\the\numexpr #4\expandafter}%
    \romannumeral0\XINT_RQ {}#3\R\R\R\R\R\R\R\Z \Z\Z\Z\Z
}%
\def\XINT_mul_choice_compare #1#2%
{%
    \ifnum #1>#2
        \expandafter \XINT_mul_choice_i
    \else
        \expandafter \XINT_mul_choice_ii
    \fi
    {#1}{#2}%
}%
\def\XINT_mul_choice_i #1#2%
{%
   \ifnum #1<\numexpr\ifcase \numexpr (#2-3)/4\relax
                       \or 330\or 168\or 109\or 80\or 66\or 52\else 0\fi\relax
       \expandafter\XINT_mul_choice_same
   \else
       \expandafter\XINT_mul_choice_permute
   \fi
}%
\def\XINT_mul_choice_ii #1#2%
```

```
{%
   \ifnum #2<\numexpr\ifcase \numexpr (#1-3)/4\relax
                       \or 330\or 168\or 109\or 80\or 66\or 52\else 0\fi\relax
       \expandafter\XINT_mul_choice_permute
   \else
       \expandafter\XINT_mul_choice_same
   \fi
}%
\def\XINT_mul_choice_same #1#2%
{%
    \expandafter\XINT_mul_enter
    \romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\Z
    \W\X\Y\Z #2\W\X\Y\Z
}%
\def\XINT_mul_choice_permute #1#2%
{%
    \expandafter\XINT_mul_enter
    \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\Z
    \W\X\Y\Z #1\W\X\Y\Z
}%
```

Cette portion de routine d'addition se branche directement sur _addr_ lorsque le premier nombre est épuisé, ce qui est garanti arriver avant le second nombre. Elle produit son résultat toujours sur 4n, renversé. Ses deux inputs sont garantis sur 4n.

```
\def\XINT_mul_Ar #1#2#3#4#5#6%
{%
    \xint_gob_til_z #6\xint_mul_br\Z\XINT_mul_Br #1{#6#5#4#3}{#2}%
}%
\def\xint_mul_br\Z\XINT_mul_Br #1#2%
{%
     \XINT_addr_AC_checkcarry #1%
}%
\def\XINT_mul_Br #1#2#3#4\W\X\Y\Z #5#6#7#8%
{%
    \expandafter\XINT_mul_ABEAr
    \the\numexpr #1+10#2+#8#7#6#5\relax.{#3}#4\W\X\Y\Z
}%
\def\XINT_mul_ABEAr #1#2#3#4#5#6.#7%
{%
    \XINT_mul_Ar #2{#7#6#5#4#3}%
}%
```

<< Petite >> multiplication. mul_Mr renvoie le résultat *à l'envers*, sur *4n*
\romannumeral0\XINT_mul_Mr {<n>}<N>\Z\Z\Z\Z
Fait la multiplication de <N> par <n>, qui est < 10000. <N> est présenté *à l'envers*, sur *4n*. Lorsque <n> vaut 0, donne 0000.

```
\def\XINT_mul_Mr #1%
{%
```

```
    \expandafter\XINT_mul_Mr_checkifzeroorone\expandafter{\the\numexpr #1}%
}%
\def\XINT_mul_Mr_checkifzeroorone #1%
{%
    \ifcase #1
      \expandafter\XINT_mul_Mr_zero
    \or
      \expandafter\XINT_mul_Mr_one
    \else
      \expandafter\XINT_mul_Nr
    \fi
    {0000}{}{#1}%
}%
\def\XINT_mul_Mr_zero #1\Z\Z\Z\Z { 0000}%
\def\XINT_mul_Mr_one #1#2#3#4\Z\Z\Z\Z { #4}%
\def\XINT_mul_Nr #1#2#3#4#5#6#7%
{%
    \xint_gob_til_z #4\xint_mul_pr\Z\XINT_mul_Pr {#1}{#3}{#7#6#5#4}{#2}{#3}%
}%
\def\XINT_mul_Pr #1#2#3%
{%
    \expandafter\XINT_mul_Lr\the\numexpr 10000#1+#2*#3\relax
}%
\def\XINT_mul_Lr 1#1#2#3#4#5#6#7#8#9%
{%
    \XINT_mul_Nr {#1#2#3#4}{#9#8#7#6#5}%
}%
\def\xint_mul_pr\Z\XINT_mul_Pr #1#2#3#4#5%
{%
    \xint_gob_til_zeros_iv #1\XINT_mul_Mr_end_nocarry 0000%
    \XINT_mul_Mr_end_carry #1{#4}%
}%
\def\XINT_mul_Mr_end_nocarry 0000\XINT_mul_Mr_end_carry 0000#1{ #1}%
\def\XINT_mul_Mr_end_carry #1#2#3#4#5{ #5#4#3#2#1}%
```

```
 << Petite >> multiplication. renvoie le résultat *à l'endroit*, avec *nettoy-
 age des leading zéros*.
 \romannumeral0\XINT_mul_M {<n>}<N>\Z\Z\Z\Z
 Fait la multiplication de <N> par <n>, qui est < 10000. <N> est présenté *à l'envers*,
 sur *4n*.
```

```
\def\XINT_mul_M #1%
{%
    \expandafter\XINT_mul_M_checkifzeroorone\expandafter{\the\numexpr #1}%
}%
\def\XINT_mul_M_checkifzeroorone #1%
{%
    \ifcase #1
      \expandafter\XINT_mul_M_zero
    \or
```

```
      \expandafter\XINT_mul_M_one
    \else
      \expandafter\XINT_mul_N
    \fi
    {0000}{}{#1}%
}%
\def\XINT_mul_M_zero #1\Z\Z\Z\Z { 0}%
\def\XINT_mul_M_one #1#2#3#4\Z\Z\Z\Z
{%
    \expandafter\xint_cleanupzeros_andstop\romannumeral0\XINT_rev{#4}%
}%
\def\XINT_mul_N #1#2#3#4#5#6#7%
{%
    \xint_gob_til_z #4\xint_mul_p\Z\XINT_mul_P {#1}{#3}{#7#6#5#4}{#2}{#3}%
}%
\def\XINT_mul_P #1#2#3%
{%
    \expandafter\XINT_mul_L\the\numexpr 10000#1+#2*#3\relax
}%
\def\XINT_mul_L 1#1#2#3#4#5#6#7#8#9%
{%
    \XINT_mul_N {#1#2#3#4}{#5#6#7#8#9}%
}%
\def\xint_mul_p\Z\XINT_mul_P #1#2#3#4#5%
{%
    \XINT_mul_M_end #1#4%
}%
\def\XINT_mul_M_end #1#2#3#4#5#6#7#8%
{%
    \expandafter\space\the\numexpr #1#2#3#4#5#6#7#8\relax
}%
```

```
 Routine de multiplication principale délimiteur \W\X\Y\Z
 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>\W\X\Y\Z <N2>\W\X\Y\Z
 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
```

```
\def\XINT_mul_enter #1\W\X\Y\Z #2#3#4#5%
{%
    \xint_gob_til_w
    #5\xint_mul_enterw
    #4\xint_mul_enterx
    #3\xint_mul_entery
    #2\xint_mul_enterz
    \W\XINT_mul_start {#2#3#4#5}#1\W\X\Y\Z
}%
\def\xint_mul_enterw
```

```
    #1\xint_mul_enterx
    #2\xint_mul_entery
    #3\xint_mul_enterz
    \W\XINT_mul_start  #4#5\W\X\Y\Z \X\Y\Z
{%
    \XINT_mul_M {#3#2#1}#5\Z\Z\Z\Z
}%
\def\xint_mul_enterx
    #1\xint_mul_entery
    #2\xint_mul_enterz
    \W\XINT_mul_start  #3#4\W\X\Y\Z \Y\Z
{%
    \XINT_mul_M {#2#1}#4\Z\Z\Z\Z
}%
\def\xint_mul_entery
    #1\xint_mul_enterz
    \W\XINT_mul_start  #2#3\W\X\Y\Z \Z
{%
    \XINT_mul_M {#1}#3\Z\Z\Z\Z
}%
\def\XINT_mul_start #1#2\W\X\Y\Z
{%
    \expandafter\XINT_mul_main\expandafter
    {\romannumeral0\XINT_mul_Mr {#1}#2\Z\Z\Z\Z}#2\W\X\Y\Z
}%
\def\XINT_mul_main #1#2\W\X\Y\Z #3#4#5#6%
{%
    \xint_gob_til_w
    #6\xint_mul_mainw
    #5\xint_mul_mainx
    #4\xint_mul_mainy
    #3\xint_mul_mainz
    \W\XINT_mul_compute {#1}{#3#4#5#6}#2\W\X\Y\Z
}%
\def\XINT_mul_compute #1#2#3\W\X\Y\Z
{%
    \expandafter\XINT_mul_main\expandafter
    {\romannumeral0\expandafter
     \XINT_mul_Ar\expandafter0\expandafter{\expandafter}%
     \romannumeral0\XINT_mul_Mr {#2}#3\Z\Z\Z\Z
     \W\X\Y\Z 0000#1\W\X\Y\Z }#3\W\X\Y\Z
}%
```

Ici, le deuxième nombre se termine. Fin du calcul. On utilise la variante \XINT_addm_A de l'addition car on sait que le deuxième terme est au moins aussi long que le premier. Lorsque le multiplicateur avait longueur 4n, la dernière addition a fourni le résultat à l'envers, il faut donc encore le renverser.

```
\def\xint_mul_mainw
    #1\xint_mul_mainx
```

```
    #2\xint_mul_mainy
    #3\xint_mul_mainz
    \W\XINT_mul_compute  #4#5#6\W\X\Y\Z \X\Y\Z
{%
    \expandafter\XINT_addm_A \expandafter0\expandafter{\expandafter}%
          \romannumeral0%
    \XINT_mul_Mr {#3#2#1}#6\Z\Z\Z\Z
                 \W\X\Y\Z 000#4\W\X\Y\Z
}%
\def\xint_mul_mainx
    #1\xint_mul_mainy
    #2\xint_mul_mainz
    \W\XINT_mul_compute  #3#4#5\W\X\Y\Z \Y\Z
{%
    \expandafter\XINT_addm_A\expandafter
    0\expandafter{\expandafter}%
    \romannumeral0\XINT_mul_Mr {#2#1}#5\Z\Z\Z\Z
    \W\X\Y\Z 00#3\W\X\Y\Z
}%
\def\xint_mul_mainy
    #1\xint_mul_mainz
    \W\XINT_mul_compute  #2#3#4\W\X\Y\Z \Z
{%
    \expandafter\XINT_addm_A\expandafter
    0\expandafter{\expandafter}%
    \romannumeral0\XINT_mul_Mr {#1}#4\Z\Z\Z\Z
     \W\X\Y\Z 0#2\W\X\Y\Z
}%
\def\xint_mul_mainz\W\XINT_mul_compute #1#2#3\W\X\Y\Z
{%
    \expandafter\xint_cleanupzeros_andstop\romannumeral0\XINT_rev{#1}%
}%
```

```
 Variante de la Multiplication
 \romannumeral0\XINT_mulr_enter <N1>\W\X\Y\Z <N2>\W\X\Y\Z
 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 et aussi par la division.
```

```
\def\XINT_mulr_enter #1\W\X\Y\Z #2#3#4#5%
{%
    \xint_gob_til_w
    #5\xint_mulr_enterw
    #4\xint_mulr_enterx
    #3\xint_mulr_entery
    #2\xint_mulr_enterz
    \W\XINT_mulr_start {#2#3#4#5}#1\W\X\Y\Z
}%
\def\xint_mulr_enterw
```

```
    #1\xint_mulr_enterx
    #2\xint_mulr_entery
    #3\xint_mulr_enterz
    \W\XINT_mulr_start  #4#5\W\X\Y\Z \X\Y\Z
{%
    \XINT_mul_Mr {#3#2#1}#5\Z\Z\Z\Z
}%
\def\xint_mulr_enterx
    #1\xint_mulr_entery
    #2\xint_mulr_enterz
    \W\XINT_mulr_start  #3#4\W\X\Y\Z \Y\Z
{%
    \XINT_mul_Mr {#2#1}#4\Z\Z\Z\Z
}%
\def\xint_mulr_entery
    #1\xint_mulr_enterz
    \W\XINT_mulr_start  #2#3\W\X\Y\Z \Z
{%
    \XINT_mul_Mr {#1}#3\Z\Z\Z\Z
}%
\def\XINT_mulr_start #1#2\W\X\Y\Z
{%
    \expandafter\XINT_mulr_main\expandafter
    {\romannumeral0\XINT_mul_Mr {#1}#2\Z\Z\Z\Z }#2\W\X\Y\Z
}%
\def\XINT_mulr_main #1#2\W\X\Y\Z #3#4#5#6%
{%
    \xint_gob_til_w
    #6\xint_mulr_mainw
    #5\xint_mulr_mainx
    #4\xint_mulr_mainy
    #3\xint_mulr_mainz
    \W\XINT_mulr_compute {#1}{#3#4#5#6}#2\W\X\Y\Z
}%
\def\XINT_mulr_compute #1#2#3\W\X\Y\Z
{%
    \expandafter\XINT_mulr_main\expandafter
    {\romannumeral0\expandafter
       \XINT_mul_Ar \expandafter0\expandafter{\expandafter}%
       \romannumeral0\XINT_mul_Mr {#2}#3\Z\Z\Z\Z \W\X\Y\Z 0000#1\W\X\Y\Z
    }#3\W\X\Y\Z
}%
\def\xint_mulr_mainw
    #1\xint_mulr_mainx
    #2\xint_mulr_mainy
    #3\xint_mulr_mainz
    \W\XINT_mulr_compute #4#5#6\W\X\Y\Z \X\Y\Z
{%
    \expandafter\XINT_addp_A
```

```
    \expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_mul_Mr {#3#2#1}#6\Z\Z\Z\Z
                  \W\X\Y\Z 000#4\W\X\Y\Z
}%
\def\xint_mulr_mainx
    #1\xint_mulr_mainy
    #2\xint_mulr_mainz
    \W\XINT_mulr_compute  #3#4#5\W\X\Y\Z \Y\Z
{%
    \expandafter\XINT_addp_A
    \expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_mul_Mr {#2#1}#5\Z\Z\Z\Z
                  \W\X\Y\Z 00#3\W\X\Y\Z
}%
\def\xint_mulr_mainy
    #1\xint_mulr_mainz
    \W\XINT_mulr_compute #2#3#4\W\X\Y\Z \Z
{%
    \expandafter\XINT_addp_A
    \expandafter0\expandafter{\expandafter}%
    \romannumeral0\XINT_mul_Mr {#1}#4\Z\Z\Z\Z
                  \W\X\Y\Z 0#2\W\X\Y\Z
}%
\def\xint_mulr_mainz\W\XINT_mulr_compute #1#2#3\W\X\Y\Z { #1}%
```

19.28 \xintSqr

```
\def\xintiSqr {\romannumeral0\xintisqr }%
\def\xintisqr #1%
{%
    \expandafter\XINT_sqr\expandafter {\romannumeral0\xintiabs{#1}}%
}%
\let\xintSqr\xintiSqr \let\xintsqr\xintisqr
\def\XINT_sqr #1%
{%
    \expandafter\XINT_mul_enter
          \romannumeral0%
          \XINT_RQ {}#1\R\R\R\R\R\R\R\Z
          \W\X\Y\Z #1\W\X\Y\Z
}%
```

19.29 \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 ver-
sion 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 \xintPrd-Expr which I should have used from the beginning.

```
\def\xintiPrd {\romannumeral0\xintiprd }%
\def\xintiprd #1{\xintiprdexpr #1\relax }%
\let\xintPrd\xintiPrd
\let\xintprd\xintiprd
\def\xintiPrdExpr {\romannumeral0\xintiprdexpr }%
\def\xintiprdexpr {\expandafter\XINT_prdexpr\romannumeral-`0}%
\let\xintPrdExpr\xintiPrdExpr
\let\xintprdexpr\xintiprdexpr
\def\XINT_prdexpr {\XINT_prod_loop_a 1\Z }%
\def\XINT_prod_loop_a #1\Z #2%
{%
    \expandafter\XINT_prod_loop_b \romannumeral-`0#2\Z #1\Z \Z
}%
\def\XINT_prod_loop_b #1%
{%
   \xint_gob_til_relax #1\XINT_prod_finished\relax
   \XINT_prod_loop_c #1%
}%
\def\XINT_prod_loop_c
{%
    \expandafter\XINT_prod_loop_a\romannumeral0\XINT_mul_fork
}%
\def\XINT_prod_finished #1\Z #2\Z \Z { #2}%
```

19.30 \xintFac

Modified with 1.02 and again in 1.03 for greater efficiency. I am tempted, here and elsewhere, to use \ifcase\XINT_Geq {#1}{1000000000} rather than \ifnum\XINT_Length {#1}>9 but for the time being I leave things as they stand. With release 1.05, rather than using \XINT_Length I opt finally for direct use of \numexpr (which will throw a suitable number too big message), and to raise the \xintError: FactorialOfTooBigNumber for argument larger than 1000000 (rather than 1000000000).

```
\def\xintFac {\romannumeral0\xintfac }%
\def\xintfac #1%
{%
    \expandafter\XINT_fac_fork\expandafter{\the\numexpr #1}%
}%
```

```
\def\XINT_fac_fork #1%
{%
    \ifcase\XINT_Sgn {#1}
       \xint_afterfi{\expandafter\space\expandafter 1\xint_gobble_i }%
    \or
       \expandafter\XINT_fac_checklength
    \else
       \xint_afterfi{\expandafter\xintError:FactorialOfNegativeNumber
                \expandafter\space\expandafter 1\xint_gobble_i }%
    \fi
    {#1}%
}%
\def\XINT_fac_checklength #1%
{%
    \ifnum #1>999999
         \xint_afterfi{\expandafter\xintError:FactorialOfTooBigNumber
                       \expandafter\space\expandafter 1\xint_gobble_i }%
    \else
         \xint_afterfi{\ifnum #1>9999
                          \expandafter\XINT_fac_big_loop
                       \else
                          \expandafter\XINT_fac_loop
                       \fi }%
    \fi
    {#1}%
}%
\def\XINT_fac_big_loop #1{\XINT_fac_big_loop_main {10000}{#1}{}}%
\def\XINT_fac_big_loop_main #1#2#3%
{%
    \ifnum #1<#2
        \expandafter
            \XINT_fac_big_loop_main
        \expandafter
           {\the\numexpr #1+1\expandafter }%
    \else
        \expandafter\XINT_fac_big_docomputation
    \fi
    {#2}{#3{#1}}%
}%
\def\XINT_fac_big_docomputation #1#2%
{%
    \expandafter \XINT_fac_bigcompute_loop \expandafter
    {\romannumeral0\XINT_fac_loop {9999}}#2\relax
}%
\def\XINT_fac_bigcompute_loop #1#2%
{%
    \xint_gob_til_relax #2\XINT_fac_bigcompute_end\relax
    \expandafter\XINT_fac_bigcompute_loop\expandafter
    {\expandafter\XINT_mul_enter
```

```
      \romannumeral0\XINT_RQ {}#2\R\R\R\R\R\R\R\Z
      \W\X\Y\Z #1\W\X\Y\Z }%
}%
\def\XINT_fac_bigcompute_end #1#2#3#4#5%
{%
    \XINT_fac_bigcompute_end_ #5%
}%
\def\XINT_fac_bigcompute_end_ #1\R #2\Z \W\X\Y\Z #3\W\X\Y\Z { #3}%
\def\XINT_fac_loop #1{\XINT_fac_loop_main 1{1000}{#1}}%
\def\XINT_fac_loop_main #1#2#3%
{%
    \ifnum #3>#1
    \else
        \expandafter\XINT_fac_loop_exit
    \fi
    \expandafter\XINT_fac_loop_main\expandafter
    {\the\numexpr #1+1\expandafter }\expandafter
    {\romannumeral0\XINT_mul_Mr {#1}#2\Z\Z\Z\Z }%
    {#3}%
}%
\def\XINT_fac_loop_exit #1#2#3#4#5#6#7%
{%
    \XINT_fac_loop_exit_ #6%
}%
\def\XINT_fac_loop_exit_ #1#2#3%
{%
    \XINT_mul_M
}%
```

19.31 \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.

```
\def\xintiPow {\romannumeral0\xintipow }%
\def\xintipow #1%
{%
    \expandafter\xint_pow\romannumeral-`0#1\Z%
}%
\let\xintPow\xintiPow \let\xintpow\xintipow
\def\xint_pow #1#2\Z
{%
    \xint_UDsignfork
      #1\dummy \XINT_pow_Aneg
       -\dummy \XINT_pow_Anonneg
    \krof
```

```
        #1{#2}%
}%
\def\XINT_pow_Aneg #1#2#3%
{%
   \expandafter\XINT_pow_Aneg_\expandafter{\the\numexpr #3}{#2}%
}%
\def\XINT_pow_Aneg_ #1%
{%
   \ifodd #1
       \expandafter\XINT_pow_Aneg_Bodd
   \fi
   \XINT_pow_Anonneg_ {#1}%
}%
\def\XINT_pow_Aneg_Bodd #1%
{%
    \expandafter\XINT_opp\romannumeral0\XINT_pow_Anonneg_
}%
```

```
B = #3, faire le xpxp. Modified with 1.06: use of \numexpr.
```

```
\def\XINT_pow_Anonneg #1#2#3%
{%
   \expandafter\XINT_pow_Anonneg_\expandafter {\the\numexpr #3}{#1#2}%
}%
```

```
#1 = B, #2 = |A|
```

```
\def\XINT_pow_Anonneg_ #1#2%
{%
    \ifcase\XINT_Cmp {#2}{1}
        \expandafter\XINT_pow_AisOne
    \or
        \expandafter\XINT_pow_AatleastTwo
    \else
        \expandafter\XINT_pow_AisZero
    \fi
    {#1}{#2}%
}%
\def\XINT_pow_AisOne #1#2{ 1}%
```

```
#1 = B
```

```
\def\XINT_pow_AisZero #1#2%
{%
     \ifcase\XINT_Sgn {#1}
         \xint_afterfi { 1}%
     \or
         \xint_afterfi { 0}%
     \else
         \xint_afterfi {\xintError:DivisionByZero\space 0}%
```

```
    \fi
}%
\def\XINT_pow_AatleastTwo #1%
{%
    \ifcase\XINT_Sgn {#1}
        \expandafter\XINT_pow_BisZero
    \or
        \expandafter\XINT_pow_checkBlength
    \else
        \expandafter\XINT_pow_BisNegative
    \fi
    {#1}%
}%
\def\XINT_pow_BisNegative #1#2{\xintError:FractionRoundedToZero\space 0}%
\def\XINT_pow_BisZero #1#2{ 1}%
```

B = #1 > 0, A = #2 > 1. With 1.05, I replace \xintiLen{#1}>9 by direct use of \numexpr [to generate an error message if the exponent is too large] 1.06: \numexpr was already used above.

```
\def\XINT_pow_checkBlength #1#2%
{%
    \ifnum #1>999999999
        \expandafter\XINT_pow_BtooBig
    \else
        \expandafter\XINT_pow_loop
    \fi
    {#1}{#2}\XINT_pow_posprod
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
}%
\def\XINT_pow_BtooBig #1\xint_relax #2\xint_relax
                          {\xintError:ExponentTooBig\space 0}%
\def\XINT_pow_loop #1#2%
{%
    \ifnum #1 = 1
        \expandafter\XINT_pow_loop_end
    \else
        \xint_afterfi{\expandafter\XINT_pow_loop_a
            \expandafter{\the\numexpr 2*(#1/2)-#1\expandafter }% b mod 2
            \expandafter{\the\numexpr #1-#1/2\expandafter }%     [b/2]
            \expandafter{\romannumeral0\xintisqr{#2}}}%
    \fi
    {{#2}}%
}%
\def\XINT_pow_loop_end {\romannumeral0\XINT_rord_main {}\relax }%
\def\XINT_pow_loop_a #1%
{%
```

```
    \ifnum #1 = 1
        \expandafter\XINT_pow_loop
    \else
        \expandafter\XINT_pow_loop_throwaway
    \fi
}%
\def\XINT_pow_loop_throwaway #1#2#3%
{%
   \XINT_pow_loop {#1}{#2}%
}%
```

Routine de produit servant pour le calcul des puissances. Chaque nouveau terme est plus grand que ce qui a déjà été calculé. Par conséquent on a intérêt à le conserver en second dans la routine de multiplication, donc le précédent calcul a intérêt à avoir été donné sur 4n, à l'envers. Il faut donc modifier la multiplication pour qu'elle fasse cela. Ce qui oblige à utiliser une version spéciale de l'addition également.

```
\def\XINT_pow_posprod #1%
{%
    \XINT_pow_pprod_checkifempty #1\Z
}%
\def\XINT_pow_pprod_checkifempty #1%
{%
    \xint_gob_til_relax #1\XINT_pow_pprod_emptyproduct\relax
    \XINT_pow_pprod_RQfirst #1%
}%
\def\XINT_pow_pprod_emptyproduct #1\Z { 1}%
\def\XINT_pow_pprod_RQfirst #1\Z
{%
    \expandafter\XINT_pow_pprod_getnext\expandafter
    {\romannumeral0\XINT_RQ {}#1\R\R\R\R\R\R\R\Z}%
}%
\def\XINT_pow_pprod_getnext #1#2%
{%
    \XINT_pow_pprod_checkiffinished #2\Z {#1}%
}%
\def\XINT_pow_pprod_checkiffinished #1%
{%
    \xint_gob_til_relax #1\XINT_pow_pprod_end\relax
    \XINT_pow_pprod_compute #1%
}%
\def\XINT_pow_pprod_compute #1\Z #2%
{%
    \expandafter\XINT_pow_pprod_getnext\expandafter
    {\romannumeral0\XINT_mulr_enter #2\W\X\Y\Z #1\W\X\Y\Z}%
}%
\def\XINT_pow_pprod_end\relax\XINT_pow_pprod_compute #1\Z #2%
{%
    \expandafter\xint_cleanupzeros_andstop
```

```
    \romannumeral0\XINT_rev {#2}%
}%
```

19.32 \xintDivision, \xintQuo, \xintRem

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

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

```
\def\xintiDivision {\romannumeral0\xintidivision }%
\def\xintidivision #1%
{%
    \expandafter\xint_division\expandafter {\romannumeral-`0#1}%
}%
\let\xintDivision\xintiDivision \let\xintdivision\xintidivision
\def\xint_division #1#2%
{%
    \expandafter\XINT_div_fork \romannumeral-`0#2\Z #1\Z
}%
\def\XINT_Division #1#2{\romannumeral0\XINT_div_fork #2\Z #1\Z }%
```

```
#1#2 = 2e input = diviseur = B. #3#4 = 1er input = divisé = A
```

```
\def\XINT_div_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\dummy \XINT_div_BisZero
      #3\dummy \XINT_div_AisZero
       0\dummy
        {\xint_UDsignfork
           #1\dummy \XINT_div_BisNegative  % B < 0
           #3\dummy \XINT_div_AisNegative  % A < 0, B > 0
            -\dummy \XINT_div_plusplus     % B > 0, A > 0
         \krof }%
    \krof
    {#2}{#4}#1#3% #1#2=B, #3#4=A
}%
\def\XINT_div_BisZero #1#2#3#4{\xintError:DivisionByZero\space {0}{0}}%
\def\XINT_div_AisZero #1#2#3#4{ {0}{0}}%
```

```
jusqu'à présent c'est facile.
minusplus signifie B < 0, A > 0
```

```
 plusminus signifie B > 0, A < 0
 Ici #3#1 correspond au diviseur B et #4#2 au divisé A.
   Cases with B<0 or especially A<0 are treated sub-optimally in terms of post-
 processing, things get reversed which could have been produced directly in the
 wanted order, but A,B>0 is given priority for optimization.
\def\XINT_div_plusplus #1#2#3#4%
{%
    \XINT_div_prepare {#3#1}{#4#2}%
}%

 B = #3#1 < 0, A non nul positif ou négatif
\def\XINT_div_BisNegative #1#2#3#4%
{%
    \expandafter\XINT_div_BisNegative_post
    \romannumeral0\XINT_div_fork #1\Z #4#2\Z
}%
\def\XINT_div_BisNegative_post #1%
{%
    \expandafter\space\expandafter {\romannumeral0\XINT_opp #1}%
}%

 B = #3#1 > 0, A =-#2< 0
\def\XINT_div_AisNegative #1#2#3#4%
{%
    \expandafter\XINT_div_AisNegative_post
    \romannumeral0\XINT_div_prepare {#3#1}{#2}{#3#1}%
}%
\def\XINT_div_AisNegative_post #1#2%
{%
    \ifcase\XINT_Sgn {#2}
       \expandafter \XINT_div_AisNegative_zerorem
    \or
       \expandafter \XINT_div_AisNegative_posrem
    \fi
    {#1}{#2}%
}%

 en #3 on a une copie de B (à l'endroit)
\def\XINT_div_AisNegative_zerorem #1#2#3%
{%
    \expandafter\space\expandafter {\romannumeral0\XINT_opp #1}{0}%
}%

 #1 = quotient, #2 = reste, #3 = diviseur initial (à l'endroit) remplace Reste
 par B - Reste, après avoir remplacé Q par -(Q+1) de sorte que la formule a =
 qb + r, 0<= r < |b| est valable
\def\XINT_div_AisNegative_posrem #1%
{%
```

```
    \expandafter \XINT_div_AisNegative_posrem_b \expandafter
       {\romannumeral0\xintiopp{\xintiAdd {#1}{1}}}%
}%
\def\XINT_div_AisNegative_posrem_b #1#2#3%
{%
    \expandafter \xint_exchangetwo_keepbraces_andstop \expandafter
    {\romannumeral0\XINT_sub {#3}{#2}}{#1}%
}%
```

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

```
\def\XINT_div_prepare #1%
{%
    \expandafter \XINT_div_prepareB_aa \expandafter
        {\romannumeral0\XINT_length {#1}}{#1}% B > 0 ici
}%
\def\XINT_div_prepareB_aa #1%
{%
    \ifnum #1=1
      \expandafter\XINT_div_prepareB_ab
    \else
      \expandafter\XINT_div_prepareB_a
    \fi
    {#1}%
}%
\def\XINT_div_prepareB_ab #1#2%
{%
    \ifnum #2=1
      \expandafter\XINT_div_prepareB_BisOne
    \else
      \expandafter\XINT_div_prepareB_e
    \fi {000}{3}{4}{#2}%
}%
\def\XINT_div_prepareB_BisOne #1#2#3#4#5{ {#5}{0}}%
\def\XINT_div_prepareB_a #1%
{%
  \expandafter\XINT_div_prepareB_c\expandafter{\the\numexpr 4*((#1+1)/4)}{#1}%
}%
```

```
#1 = K
```

```
\def\XINT_div_prepareB_c #1#2%
{%
    \ifcase \numexpr #1-#2\relax
       \expandafter\XINT_div_prepareB_d
    \or
       \expandafter\XINT_div_prepareB_di
    \or
       \expandafter\XINT_div_prepareB_dii
```

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

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

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

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

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

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

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

 A, x, K, B, c,

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

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

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

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

\def\XINT_div_prepareA_c #1#2%
{%
    \ifcase \numexpr #1-#2\relax
```

```
2442        \expandafter\XINT_div_prepareA_d
2443    \or
2444        \expandafter\XINT_div_prepareA_di
2445    \or
2446        \expandafter\XINT_div_prepareA_dii
2447    \or
2448        \expandafter\XINT_div_prepareA_diii
2449    \fi {#1}%
2450 }%
2451 \def\XINT_div_prepareA_d     {\XINT_div_prepareA_e {}}%
2452 \def\XINT_div_prepareA_di    {\XINT_div_prepareA_e {0}}%
2453 \def\XINT_div_prepareA_dii   {\XINT_div_prepareA_e {00}}%
2454 \def\XINT_div_prepareA_diii  {\XINT_div_prepareA_e {000}}%

 #1#3 = A préparé, #2 = longueur de ce A préparé,

2455 \def\XINT_div_prepareA_e #1#2#3%
2456 {%
2457     \XINT_div_startswitch {#1#3}{#2}%
2458 }%

 A, L, x, K, B, c

2459 \def\XINT_div_startswitch #1#2#3#4%
2460 {%
2461     \ifnum #2 > #4
2462       \expandafter\XINT_div_body_a
2463     \else
2464     \ifnum #2 = #4
2465       \expandafter\expandafter\expandafter\XINT_div_final_a
2466     \else
2467       \expandafter\expandafter\expandafter\XINT_div_finished_a
2468     \fi\fi {#1}{#4}{#3}{0000}{#2}%
2469 }%

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

2470 \def\XINT_div_finished_a #1#2#3%
2471 {%
2472   \expandafter\XINT_div_finished_b\expandafter {\romannumeral0\XINT_cuz {#1}}%
2473 }%

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

2474 \def\XINT_div_finished_b #1#2#3#4#5%
2475 {%
2476     \ifcase \XINT_Sgn {#1}
2477        \xint_afterfi {\XINT_div_finished_c {0}}%
2478     \or
2479        \xint_afterfi {\expandafter\XINT_div_finished_c\expandafter
2480                        {\romannumeral0\XINT_dsh_checksignx #5\Z {#1}}%
2481                       }%
```

```
    \fi
    {#2}%
}%
\def\XINT_div_finished_c #1#2%
{%
    \expandafter\space\expandafter {\romannumeral0\XINT_rev_andcuz {#2}}{#1}%
}%
```

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

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

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

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

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

```
\def\XINT_div_final_da #1%
{%
    \ifnum #1>9
       \expandafter\XINT_div_final_dP
    \else
       \xint_afterfi
       {\ifnum #1<0
        \expandafter\XINT_div_final_dN
        \else
        \expandafter\XINT_div_final_db
        \fi }%
    \fi
}%
\def\XINT_div_final_dN #1%
{%
    \expandafter\XINT_div_final_dP\the\numexpr #1-1\relax
}%
\def\XINT_div_final_dP #1#2#3#4#5% q,A,Q,L,B (puis c)
```

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

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

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

```
    \XINT_div_body_c {#1#2#3#4#5#6#7#8}%
}%
```

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

```
\def\XINT_div_body_c #1#2#3%
{%
    \XINT_div_body_d {#3}{}#2\Z {#1}{#3}%
}%
\def\XINT_div_body_d #1#2#3#4#5#6%
{%
    \ifnum #1 > 0
        \expandafter\XINT_div_body_d
        \expandafter{\the\numexpr #1-4\expandafter }%
    \else
        \expandafter\XINT_div_body_e
    \fi
    {#6#5#4#3#2}%
}%
\def\XINT_div_body_e #1#2\Z #3%
{%
    \XINT_div_body_f {#3}{#1}{#2}%
}%
```

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

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

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

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

```
    {\romannumeral0\XINT_mul_Mr {#1}0000#3\Z\Z\Z\Z }%
    {\romannumeral0\XINT_mul_Mr {#2}#3\Z\Z\Z\Z }%
    {#1#2}%
}%
\def\XINT_div_body_gggk #1#2#3#4%
{%
    \expandafter\XINT_div_body_h
    \romannumeral0\XINT_div_sub_xpxp
    {\romannumeral0\expandafter\XINT_mul_Ar
     \expandafter0\expandafter{\expandafter}#2\W\X\Y\Z #1\W\X\Y\Z }%
    {#4}\Z {#3}%
}%
```

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

```
\def\XINT_div_body_h #1#2#3#4#5#6#7#8#9\Z
{%
    \ifnum #1#2#3#4>0
        \xint_afterfi{\XINT_div_body_i {#1#2#3#4#5#6#7#8}}%
    \else
        \expandafter\XINT_div_body_k
    \fi
    {#1#2#3#4#5#6#7#8#9}%
}%
\def\XINT_div_body_k #1#2#3%
{%
    \XINT_div_body_l {#1}{#2}%
}%
```

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

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

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

```
\def\XINT_div_body_l #1#2#3#4#5#6#7%
{%
   \expandafter\XINT_div_body_m
   \the\numexpr 100000000+#2\relax {#6}{#3}{#7}{#1#5}{#4}%
}%
```

```
 chiffres de q, Q, K, L, A'=nouveau A, x, B, c
\def\XINT_div_body_m #1#2#3#4#5#6#7#8#9%
{%
    \ifnum #2#3#4#5>0
       \xint_afterfi {\XINT_div_body_n {#9#8#7#6#5#4#3#2}}%
    \else
       \xint_afterfi {\XINT_div_body_n {#9#8#7#6}}%
    \fi
}%

 q renversé, Q, K, L, A', x, B, c
\def\XINT_div_body_n #1#2%
{%
    \expandafter\XINT_div_body_o\expandafter
    {\romannumeral0\XINT_addr_A 0{}#1\W\X\Y\Z #2\W\X\Y\Z }%
}%

 q+Q, K, L, A', x, B, c
\def\XINT_div_body_o #1#2#3#4%
{%
    \XINT_div_body_p {#3}{#2}{}#4\Z {#1}%
}%

 L, K, {}, A'\Z, q+Q, x, B, c
\def\XINT_div_body_p #1#2#3#4#5#6#7%
{%
    \ifnum #1 > #2
        \xint_afterfi
        {\ifnum #4#5#6#7 > 0
           \expandafter\XINT_div_body_q
         \else
           \expandafter\XINT_div_body_repeatp
         \fi }%
    \else
        \expandafter\XINT_div_gotofinal_a
    \fi
    {#1}{#2}{#3}#4#5#6#7%
}%

 L, K, zeros, A' avec moins de zéros\Z, q+Q, x, B, c
\def\XINT_div_body_repeatp #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_div_body_p\expandafter{\the\numexpr #1-4}{#2}{0000#3}%
}%

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

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

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

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

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

```
\def\XINT_div_sub_xpxp #1%
{%
    \expandafter \XINT_div_sub_xpxp_  \expandafter{#1}%
}%
\def\XINT_div_sub_xpxp_ #1#2%
{%
    \expandafter\expandafter\expandafter\XINT_div_sub_xpxp__
     #2\W\X\Y\Z #1\W\X\Y\Z
}%
\def\XINT_div_sub_xpxp__
{%
    \XINT_div_sub_A 1{}%
}%
\def\XINT_div_sub_A #1#2#3#4#5#6%
{%
    \xint_gob_til_w #3\xint_div_sub_az\W
    \XINT_div_sub_B #1{#3#4#5#6}{#2}%
}%
\def\XINT_div_sub_B #1#2#3#4\W\X\Y\Z #5#6#7#8%
{%
    \xint_gob_til_w #5\xint_div_sub_bz\W
    \XINT_div_sub_onestep #1#2{#8#7#6#5}{#3}#4\W\X\Y\Z
}%
\def\XINT_div_sub_onestep #1#2#3#4#5#6%
{%
```

```
    \expandafter\XINT_div_sub_backtoA\the\numexpr 11#5#4#3#2-#6+#1-1\relax.%
}%
\def\XINT_div_sub_backtoA #1#2#3.#4%
{%
    \XINT_div_sub_A #2{#3#4}%
}%
\def\xint_div_sub_bz\W\XINT_div_sub_onestep #1#2#3#4#5#6#7%
{%
    \xint_UDzerofork
      #1\dummy  \XINT_div_sub_C   %
       0\dummy  \XINT_div_sub_D   % pas de retenue
    \krof
    {#7}#2#3#4#5%
}%
\def\XINT_div_sub_D #1#2\W\X\Y\Z
{%
    \expandafter\space
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
    #1%
}%
\def\XINT_div_sub_C #1#2#3#4#5%
{%
    \xint_gob_til_w #2\xint_div_sub_cz\W
    \XINT_div_sub_AC_onestep {#5#4#3#2}{#1}%
}%
\def\XINT_div_sub_AC_onestep #1%
{%
    \expandafter\XINT_div_sub_backtoC\the\numexpr 11#1-1\relax.%
}%
\def\XINT_div_sub_backtoC #1#2#3.#4%
{%
    \XINT_div_sub_AC_checkcarry #2{#3#4}% la retenue va \^etre examin\'ee
}%
\def\XINT_div_sub_AC_checkcarry #1%
{%
    \xint_gob_til_one #1\xint_div_sub_AC_nocarry 1\XINT_div_sub_C
}%
\def\xint_div_sub_AC_nocarry 1\XINT_div_sub_C #1#2\W\X\Y\Z
{%
    \expandafter\space
    \romannumeral0%
    \XINT_rord_main {}#2%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
```

```
2777         \xint_undef\xint_undef\xint_undef\xint_undef
2778       \xint_relax
2779     #1%
2780 }%
2781 \def\xint_div_sub_cz\W\XINT_div_sub_AC_onestep #1#2{ #2}%
2782 \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, QUO-
  TIENT BY TEN, QUOTIENT OR MULTIPLICATION BY POWER OF TEN, SPLIT OPERATION.
```

19.33 \xintFDg

```
  FIRST DIGIT. Code simplified in 1.05. And prepared for redefinition by xint-
  frac to parse through \xintNum
```

```
2783 \def\xintiFDg {\romannumeral0\xintifdg }%
2784 \def\xintifdg #1%
2785 {%
2786     \expandafter\XINT_fdg \romannumeral-‘0#1\W\Z
2787 }%
2788 \let\xintFDg\xintiFDg \let\xintfdg\xintifdg
2789 \def\XINT_FDg #1{\romannumeral0\XINT_fdg #1\W\Z }%
2790 \def\XINT_fdg #1#2#3\Z
2791 {%
2792     \xint_UDzerominusfork
2793       #1-\dummy  { 0}%   zero
2794       0#1\dummy  { #2}%  negative
2795        0-\dummy  { #1}%  positive
2796     \krof
2797 }%
```

19.34 \xintLDg

```
  LAST DIGIT. Simplified in 1.05. And prepared for extension by xintfrac to parse
  through \xintNum
```

```
2798 \def\xintiLDg {\romannumeral0\xintildg }%
2799 \def\xintildg #1%
2800 {%
2801     \expandafter\XINT_ldg\expandafter {\romannumeral-‘0#1}%
2802 }%
2803 \let\xintLDg\xintiLDg \let\xintldg\xintildg
2804 \def\XINT_LDg #1{\romannumeral0\XINT_ldg {#1}}%
2805 \def\XINT_ldg #1%
2806 {%
2807     \expandafter\XINT_ldg_\romannumeral0\XINT_rev {#1}\Z
2808 }%
2809 \def\XINT_ldg_ #1#2\Z{ #1}%
```

19.35 \xintMON

MINUS ONE TO THE POWER N

```
\def\xintiMON {\romannumeral0\xintimon }%
\def\xintimon #1%
{%
    \ifodd\xintiLDg {#1}
        \xint_afterfi{ -1}%
    \else
        \xint_afterfi{ 1}%
    \fi
}%
\def\xintiMMON {\romannumeral0\xintimmon }%
\def\xintimmon #1%
{%
    \ifodd\xintiLDg {#1}
        \xint_afterfi{ 1}%
    \else
        \xint_afterfi{ -1}%
    \fi
}%
\let\xintMON\xintiMON \let\xintmon\xintimon
\let\xintMMON\xintiMMON \let\xintmmon\xintimmon
```

19.36 \xintOdd

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

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

19.37 \xintDSL

```
DECIMAL SHIFT LEFT (=MULTIPLICATION PAR 10)
```

```
\def\xintDSL {\romannumeral0\xintdsl }%
\def\xintdsl #1%
{%
    \expandafter\XINT_dsl \romannumeral-`0#1\Z
}%
\def\XINT_DSL #1{\romannumeral0\XINT_dsl #1\Z }%
\def\XINT_dsl #1%
{%
    \xint_gob_til_zero #1\xint_dsl_zero 0\XINT_dsl_ #1%
}%
\def\xint_dsl_zero 0\XINT_dsl_ 0#1\Z { 0}%
\def\XINT_dsl_ #1\Z { #10}%
```

19.38 \xintDSR

```
DECIMAL SHIFT RIGHT (=DIVISION PAR 10)
```

```
\def\xintDSR {\romannumeral0\xintdsr }%
\def\xintdsr #1%
{%
    \expandafter\XINT_dsr_a\expandafter {\romannumeral-`0#1}\W\Z
}%
\def\XINT_DSR #1{\romannumeral0\XINT_dsr_a {#1}\W\Z }%
\def\XINT_dsr_a
{%
    \expandafter\XINT_dsr_b\romannumeral0\XINT_rev
}%
\def\XINT_dsr_b #1#2#3\Z
{%
    \xint_gob_til_w #2\xint_dsr_onedigit\W
    \xint_minus #2\xint_dsr_onedigit-%
    \expandafter\XINT_dsr_removew
    \romannumeral0\XINT_rev {#2#3}%
}%
\def\xint_dsr_onedigit #1\XINT_rev #2{ 0}%
\def\XINT_dsr_removew #1\W { }%
```

19.39 \xintDSH, \xintDSHr

```
DECIMAL SHIFTS \xintDSH {x}{A}
si x <= 0, fait A -> A.10^(|x|). v1.03 corrige l'oversight pour A=0.n si x >
0, et A >=0, fait A -> quo(A,10^(x))
si x > 0, et A < 0, fait A -> -quo(-A,10^(x))
(donc pour x > 0 c'est comme DSR itéré x fois)
\xintDSHr donne le 'reste' (si x<=0 donne zéro).
```

Release 1.06 now feeds x to a \numexpr first. I will revise the legacy code on another occasion.

```
\def\xintDSHr {\romannumeral0\xintdshr }%
\def\xintdshr #1%
{%
    \expandafter\XINT_dshr_checkxpositive \the\numexpr #1\relax\Z
}%
\def\XINT_dshr_checkxpositive #1%
{%
    \xint_UDzerominusfork
      0#1\dummy \XINT_dshr_xzeroorneg
      #1-\dummy \XINT_dshr_xzeroorneg
       0-\dummy \XINT_dshr_xpositive
    \krof #1%
}%
\def\XINT_dshr_xzeroorneg #1\Z #2{ 0}%
\def\XINT_dshr_xpositive #1\Z
{%
    \expandafter\xint_secondoftwo_andstop\romannumeral0\xintdsx {#1}%
}%
\def\xintDSH {\romannumeral0\xintdsh }%
\def\xintdsh #1#2%
{%
    \expandafter\xint_dsh\expandafter {\romannumeral-`0#2}{#1}%
}%
\def\xint_dsh #1#2%
{%
    \expandafter\XINT_dsh_checksignx \the\numexpr #2\relax\Z {#1}%
}%
\def\XINT_dsh_checksignx #1%
{%
    \xint_UDzerominusfork
      #1-\dummy  \XINT_dsh_xiszero
      0#1\dummy  \XINT_dsx_xisNeg_checkA     % on passe direct dans DSx
       0-\dummy  {\XINT_dsh_xisPos #1}%
    \krof
}%
\def\XINT_dsh_xiszero #1\Z #2{ #2}%
\def\XINT_dsh_xisPos #1\Z #2%
{%
    \expandafter\xint_firstoftwo_andstop
    \romannumeral0\XINT_dsx_checksignA #2\Z {#1}% via DSx
}%
```

19.40 \xintDSx

Je fais cette routine pour la version 1.01, après modification de \xintDecSplit. Dorénavant \xintDSx fera appel à \xintDecSplit et de même \xintDSH fera appel

```
à \xintDSx. J'ai donc supprimé entièrement l'ancien code de \xintDSH et re-écrit
entièrement celui de \xintDecSplit pour x positif.
  --> Attention le cas x=0 est traité dans la même catégorie que x > 0 <--
si x < 0, fait A -> A.10^(|x|)
si x >= 0, et A >=0, fait A -> {quo(A,10^(x))}{rem(A,10^(x))}
si x >= 0, et A < 0, d'abord on calcule {quo(-A,10^(x))}{rem(-A,10^(x))}
puis, si le premier n'est pas nul on lui donne le signe -
si le premier est nul on donne le signe - au second.
  On peut donc toujours reconstituer l'original A par 10^x Q \pm R où il faut
prendre le signe plus si Q est positif ou nul et le signe moins si Q est stricte-
ment 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.

2920 \def\xintDSx {\romannumeral0\xintdsx }%
2921 \def\xintdsx #1#2%
2922 {%
2923     \expandafter\xint_dsx\expandafter {\romannumeral-`0#2}{#1}%
2924 }%
2925 \def\xint_dsx #1#2%
2926 {%
2927     \expandafter\XINT_dsx_checksignx \the\numexpr #2\relax\Z {#1}%
2928 }%
2929 \def\XINT_DSx #1#2{\romannumeral0\XINT_dsx_checksignx #1\Z {#2}}%
2930 \def\XINT_dsx #1#2{\XINT_dsx_checksignx #1\Z {#2}}%
2931 \def\XINT_dsx_checksignx #1%
2932 {%
2933     \xint_UDzerominusfork
2934       #1-\dummy  \XINT_dsx_xisZero
2935       0#1\dummy  \XINT_dsx_xisNeg_checkA
2936        0-\dummy  {\XINT_dsx_xisPos #1}%
2937     \krof
2938 }%
2939 \def\XINT_dsx_xisZero #1\Z #2{ {#2}{0}}% attention comme x > 0
2940 \def\XINT_dsx_xisNeg_checkA #1\Z #2%
2941 {%
2942     \XINT_dsx_xisNeg_checkA_ #2\Z {#1}%
2943 }%
2944 \def\XINT_dsx_xisNeg_checkA_ #1#2\Z #3%
2945 {%
2946     \xint_gob_til_zero #1\XINT_dsx_xisNeg_Azero 0%
2947     \XINT_dsx_xisNeg_checkx {#3}{#3}\Z {#1#2}%
2948 }%
2949 \def\XINT_dsx_xisNeg_Azero #1\Z #2{ 0}%
2950 \def\XINT_dsx_xisNeg_checkx #1%
2951 {%
2952     \ifnum #1>999999999
2953        \xint_afterfi
```

```
2954        {\xintError:TooBigDecimalShift
2955         \expandafter\space\expandafter 0\xint_gobble_iii }%
2956     \else
2957        \expandafter \XINT_dsx_zeroloop
2958     \fi
2959 }%
2960 \def\XINT_dsx_zeroloop #1%
2961 {%
2962     \ifnum #1<9 \XINT_dsx_exita\fi
2963     \expandafter\XINT_dsx_zeroloop\expandafter
2964         {\the\numexpr #1-8}00000000%
2965 }%
2966 \def\XINT_dsx_exita\fi\expandafter\XINT_dsx_zeroloop
2967 {%
2968     \fi\expandafter\XINT_dsx_exitb
2969 }%
2970 \def\XINT_dsx_exitb #1%
2971 {%
2972     \expandafter\expandafter\expandafter
2973     \XINT_dsx_addzeros\csname xint_gobble_\romannumeral -#1\endcsname
2974 }%
2975 \def\XINT_dsx_addzeros #1\Z #2{ #2#1}%
2976 \def\XINT_dsx_xisPos #1\Z #2%
2977 {%
2978     \XINT_dsx_checksignA #2\Z {#1}%
2979 }%
2980 \def\XINT_dsx_checksignA #1%
2981 {%
2982     \xint_UDzerominusfork
2983       #1-\dummy  \XINT_dsx_AisZero
2984       0#1\dummy  \XINT_dsx_AisNeg
2985        0-\dummy  {\XINT_dsx_AisPos #1}%
2986     \krof
2987 }%
2988 \def\XINT_dsx_AisZero #1\Z #2{ {0}{0}}%
2989 \def\XINT_dsx_AisNeg #1\Z #2%
2990 {%
2991     \expandafter\XINT_dsx_AisNeg_dosplit_andcheckfirst
2992     \romannumeral0\XINT_split_checksizex {#2}{#1}%
2993 }%
2994 \def\XINT_dsx_AisNeg_dosplit_andcheckfirst #1%
2995 {%
2996     \XINT_dsx_AisNeg_checkiffirstempty #1\Z
2997 }%
2998 \def\XINT_dsx_AisNeg_checkiffirstempty #1%
2999 {%
3000     \xint_gob_til_z #1\XINT_dsx_AisNeg_finish_zero\Z
3001     \XINT_dsx_AisNeg_finish_notzero #1%
3002 }%
```

```
\def\XINT_dsx_AisNeg_finish_zero\Z
    \XINT_dsx_AisNeg_finish_notzero\Z #1%
{%
    \expandafter\XINT_dsx_end
    \expandafter {\romannumeral0\XINT_num {-#1}}{0}%
}%
\def\XINT_dsx_AisNeg_finish_notzero #1\Z #2%
{%
    \expandafter\XINT_dsx_end
    \expandafter {\romannumeral0\XINT_num {#2}}{-#1}%
}%
\def\XINT_dsx_AisPos #1\Z #2%
{%
    \expandafter\XINT_dsx_AisPos_finish
    \romannumeral0\XINT_split_checksizex {#2}{#1}%
}%
\def\XINT_dsx_AisPos_finish #1#2%
{%
    \expandafter\XINT_dsx_end
    \expandafter {\romannumeral0\XINT_num {#2}}%
                 {\romannumeral0\XINT_num {#1}}%
}%
\def\XINT_dsx_end #1#2%
{%
    \expandafter\space\expandafter{#2}{#1}%
}%
```

19.41 \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 imme-
diately a \numexpr with x. Some simplifications should probably be made to the
code, which is kept as is for the time being.
```

```
\def\xintDecSplitL {\romannumeral0\xintdecsplitl }%
\def\xintDecSplitR {\romannumeral0\xintdecsplitr }%
```

```
\def\xintdecsplitl
{%
    \expandafter\xint_firstoftwo_andstop
    \romannumeral0\xintdecsplit
}%
\def\xintdecsplitr
{%
    \expandafter\xint_secondoftwo_andstop
    \romannumeral0\xintdecsplit
}%
\def\xintDecSplit {\romannumeral0\xintdecsplit }%
\def\xintdecsplit #1#2%
{%
    \expandafter \xint_split \expandafter
    {\romannumeral0\xintiabs {#2}}{#1}%  fait expansion de A
}%
\def\xint_split #1#2%
{%
    \expandafter\XINT_split_checksizex\expandafter{\the\numexpr #2}{#1}%
}%
\def\XINT_split_checksizex #1% 999999999 is anyhow very big, could be reduced
{%
    \ifnum\numexpr\XINT_Abs{#1}>999999999
       \xint_afterfi {\xintError:TooBigDecimalSplit\XINT_split_bigx }%
    \else
       \expandafter\XINT_split_xfork
    \fi
    #1\Z
}%
\def\XINT_split_bigx  #1\Z #2%
{%
    \ifcase\XINT_Sgn {#1}
    \or \xint_afterfi { {}{#2}}% positive big x
    \else
        \xint_afterfi { {#2}{}}% negative big x
    \fi
}%
\def\XINT_split_xfork #1%
{%
    \xint_UDzerominusfork
      #1-\dummy  \XINT_split_zerosplit
      0#1\dummy  \XINT_split_fromleft
       0-\dummy  {\XINT_split_fromright #1}%
    \krof
}%
\def\XINT_split_zerosplit #1\Z #2{ {#2}{}}%
\def\XINT_split_fromleft  #1\Z #2%
{%
    \XINT_split_fromleft_loop {#1}{}#2\W\W\W\W\W\W\W\W\Z
```

```
}%
\def\XINT_split_fromleft_loop #1%
{%
    \ifnum #1<8 \XINT_split_fromleft_exita\fi
    \expandafter\XINT_split_fromleft_loop_perhaps\expandafter
    {\the\numexpr #1-8\expandafter}\XINT_split_fromleft_eight
}%
\def\XINT_split_fromleft_eight #1#2#3#4#5#6#7#8#9{#9{#1#2#3#4#5#6#7#8#9}}%
\def\XINT_split_fromleft_loop_perhaps #1#2%
{%
    \xint_gob_til_w #2\XINT_split_fromleft_toofar\W
    \XINT_split_fromleft_loop {#1}%
}%
\def\XINT_split_fromleft_toofar\W\XINT_split_fromleft_loop #1#2#3\Z
{%
    \XINT_split_fromleft_toofar_b #2\Z
}%
\def\XINT_split_fromleft_toofar_b #1\W #2\Z { {#1}{}}%
\def\XINT_split_fromleft_exita\fi
    \expandafter\XINT_split_fromleft_loop_perhaps\expandafter #1#2%
   {\fi \XINT_split_fromleft_exitb #1}%
\def\XINT_split_fromleft_exitb\the\numexpr #1-8\expandafter
{%
    \csname XINT_split_fromleft_endsplit_\romannumeral #1\endcsname
}%
\def\XINT_split_fromleft_endsplit_ #1#2\W #3\Z { {#1}{#2}}%
\def\XINT_split_fromleft_endsplit_i #1#2%
                {\XINT_split_fromleft_checkiftoofar #2{#1#2}}%
\def\XINT_split_fromleft_endsplit_ii #1#2#3%
                {\XINT_split_fromleft_checkiftoofar #3{#1#2#3}}%
\def\XINT_split_fromleft_endsplit_iii #1#2#3#4%
                {\XINT_split_fromleft_checkiftoofar #4{#1#2#3#4}}%
\def\XINT_split_fromleft_endsplit_iv #1#2#3#4#5%
                {\XINT_split_fromleft_checkiftoofar #5{#1#2#3#4#5}}%
\def\XINT_split_fromleft_endsplit_v #1#2#3#4#5#6%
                {\XINT_split_fromleft_checkiftoofar #6{#1#2#3#4#5#6}}%
\def\XINT_split_fromleft_endsplit_vi #1#2#3#4#5#6#7%
                {\XINT_split_fromleft_checkiftoofar #7{#1#2#3#4#5#6#7}}%
\def\XINT_split_fromleft_endsplit_vii #1#2#3#4#5#6#7#8%
                {\XINT_split_fromleft_checkiftoofar #8{#1#2#3#4#5#6#7#8}}%
\def\XINT_split_fromleft_checkiftoofar #1#2#3\W #4\Z
{%
    \xint_gob_til_w #1\XINT_split_fromleft_wenttoofar\W
    \space {#2}{#3}%
}%
\def\XINT_split_fromleft_wenttoofar\W\space #1%
{%
    \XINT_split_fromleft_wenttoofar_b #1\Z
}%
```

```
\def\XINT_split_fromleft_wenttoofar_b #1\W #2\Z { {#1}}%
\def\XINT_split_fromright #1\Z #2%
{%
    \expandafter \XINT_split_fromright_a \expandafter
    {\romannumeral0\XINT_rev {#2}}{#1}{#2}%
}%
\def\XINT_split_fromright_a #1#2%
{%
    \XINT_split_fromright_loop {#2}{}#1\W\W\W\W\W\W\W\W\Z
}%
\def\XINT_split_fromright_loop #1%
{%
    \ifnum #1<8 \XINT_split_fromright_exita\fi
    \expandafter\XINT_split_fromright_loop_perhaps\expandafter
    {\the\numexpr #1-8\expandafter }\XINT_split_fromright_eight
}%
\def\XINT_split_fromright_eight #1#2#3#4#5#6#7#8#9{#9{#9#8#7#6#5#4#3#2#1}}%
\def\XINT_split_fromright_loop_perhaps #1#2%
{%
    \xint_gob_til_w #2\XINT_split_fromright_toofar\W
    \XINT_split_fromright_loop {#1}%
}%
\def\XINT_split_fromright_toofar\W\XINT_split_fromright_loop #1#2#3\Z { {}}%
\def\XINT_split_fromright_exita\fi
    \expandafter\XINT_split_fromright_loop_perhaps\expandafter #1#2%
    {\fi \XINT_split_fromright_exitb #1}%
\def\XINT_split_fromright_exitb\the\numexpr #1-8\expandafter
{%
    \csname XINT_split_fromright_endsplit_\romannumeral #1\endcsname
}%
\def\XINT_split_fromright_endsplit_ #1#2\W #3\Z #4%
{%
    \expandafter\space\expandafter {\romannumeral0\XINT_rev{#2}}{#1}%
}%
\def\XINT_split_fromright_endsplit_i   #1#2%
            {\XINT_split_fromright_checkiftoofar #2{#2#1}}%
\def\XINT_split_fromright_endsplit_ii  #1#2#3%
            {\XINT_split_fromright_checkiftoofar #3{#3#2#1}}%
\def\XINT_split_fromright_endsplit_iii #1#2#3#4%
            {\XINT_split_fromright_checkiftoofar #4{#4#3#2#1}}%
\def\XINT_split_fromright_endsplit_iv  #1#2#3#4#5%
            {\XINT_split_fromright_checkiftoofar #5{#5#4#3#2#1}}%
\def\XINT_split_fromright_endsplit_v   #1#2#3#4#5#6%
            {\XINT_split_fromright_checkiftoofar #6{#6#5#4#3#2#1}}%
\def\XINT_split_fromright_endsplit_vi  #1#2#3#4#5#6#7%
            {\XINT_split_fromright_checkiftoofar #7{#7#6#5#4#3#2#1}}%
\def\XINT_split_fromright_endsplit_vii #1#2#3#4#5#6#7#8%
            {\XINT_split_fromright_checkiftoofar #8{#8#7#6#5#4#3#2#1}}%
\def\XINT_split_fromright_checkiftoofar #1%
```

```
{%
    \xint_gob_til_w #1\XINT_split_fromright_wenttoofar\W
    \XINT_split_fromright_endsplit_
}%
\def\XINT_split_fromright_wenttoofar\W\XINT_split_fromright_endsplit_ #1\Z #2%
    { {}{#2}}%
\XINT_restorecatcodes_endinput%
```

20 Package xintgcd implementation

The commenting is currently (2013/05/14) very sparse.

Contents

20.1 Catcodes, ε-TEX and reload detection

The code for reload detection is copied from Heiko Oberdiek's packages, and adapted here to check for previous loading of the master xint package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \def\space { }%
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintgcd.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
```

```
\expandafter
\ifx\csname numexpr\endcsname\relax
   \y{xintgcd}{\numexpr not available, aborting input}%
   \aftergroup\endinput
\else
  \ifx\x\relax   % plain-TeX, first loading of xintgcd.sty
    \ifx\w\relax % but xint.sty not yet loaded.
       \y{xintgcd}{Package xint is required}%
       \y{xintgcd}{Will try \string\input\space xint.sty}%
       \def\z{\endgroup\input xint.sty\relax}%
    \fi
  \else
    \def\empty {}%
    \ifx\x\empty % LaTeX, first loading,
    % variable is initialized, but \ProvidesPackage not yet seen
        \ifx\w\relax % xint.sty not yet loaded.
          \y{xintgcd}{Package xint is required}%
          \y{xintgcd}{Will try \string\RequirePackage{xint}}%
          \def\z{\endgroup\RequirePackage{xint}}%
        \fi
    \else
      \y{xintgcd}{I was already loaded, aborting input}%
      \aftergroup\endinput
    \fi
  \fi
\fi
\z%
```

20.2 Confirmation of xint loading

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \def\empty {}%
  \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
```

```
  \ifx\w\relax % Plain TeX, user gave a file name at the prompt
      \y{xintgcd}{Loading of package xint failed, aborting input}%
      \aftergroup\endinput
  \fi
  \ifx\w\empty % LaTeX, user gave a file name at the prompt
      \y{xintgcd}{Loading of package xint failed, aborting input}%
      \aftergroup\endinput
  \fi
\endgroup%
```

20.3 Catcodes

Perhaps catcodes have changed after the loading of **xint** and prior to the current loading of **xintgcd**, so we can not employ the `\XINT_restorecatcodes_endinput` in this style file. But there is no problem using `\XINT_setcatcodes`.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode95=11   % _
  \def\x
  {%
      \endgroup
      \edef\XINT_gcd_restorecatcodes_endinput
      {%
        \catcode36=\the\catcode36   % $
        \catcode96=\the\catcode96   % `
        \catcode47=\the\catcode47   % /
        \catcode41=\the\catcode41   % )
        \catcode40=\the\catcode40   % (
        \catcode42=\the\catcode42   % *
        \catcode43=\the\catcode43   % +
        \catcode62=\the\catcode62   % >
        \catcode60=\the\catcode60   % <
        \catcode58=\the\catcode58   % :
        \catcode46=\the\catcode46   % .
        \catcode45=\the\catcode45   % -
        \catcode44=\the\catcode44   % ,
        \catcode35=\the\catcode35   % #
        \catcode95=\the\catcode95   % _
        \catcode125=\the\catcode125 % }
        \catcode123=\the\catcode123 % {
        \endlinechar=\the\endlinechar
        \catcode13=\the\catcode13   % ^^M
        \catcode32=\the\catcode32   %
        \catcode61=\the\catcode61\relax   % =
        \noexpand\endinput
      }%
```

```
      \XINT_setcatcodes
      \catcode36=3  % $
  }%
\x
```

20.4 Package identification

```
\begingroup
  \catcode64=11 % @
  \catcode91=12 % [
  \catcode93=12 % ]
  \catcode58=12 % :
  \expandafter\ifx\csname ProvidesPackage\endcsname\relax
    \def\x#1#2#3[#4]{\endgroup
      \immediate\write-1{Package: #3 #4}%
      \xdef#1{#4}%
    }%
  \else
    \def\x#1#2[#3]{\endgroup
      #2[{#3}]%
      \ifx#1\@undefined
        \xdef#1{#3}%
      \fi
      \ifx#1\relax
        \xdef#1{#3}%
      \fi
    }%
  \fi
\expandafter\x\csname ver@xintgcd.sty\endcsname
\ProvidesPackage{xintgcd}%
  [2013/05/14 v1.06b Euclide algorithm with xint package (jfB)]%
```

20.5 \xintGCD

```
\def\xintGCD {\romannumeral0\xintgcd }%
\def\xintgcd #1%
{%
    \expandafter\XINT_gcd\expandafter{\romannumeral0\xintiabs {#1}}%
}%
\def\XINT_gcd #1#2%
{%
    \expandafter\XINT_gcd_fork\romannumeral0\xintiabs {#2}\Z #1\Z
}%
```

Ici #3#4=A, #1#2=B

```
\def\XINT_gcd_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\dummy \XINT_gcd_BisZero
      #3\dummy \XINT_gcd_AisZero
```

```
3337        0\dummy \XINT_gcd_loop
3338     \krof
3339     {#1#2}{#3#4}%
3340 }%
3341 \def\XINT_gcd_AisZero #1#2{ #1}%
3342 \def\XINT_gcd_BisZero #1#2{ #2}%
3343 \def\XINT_gcd_CheckRem #1#2\Z
3344 {%
3345     \xint_gob_til_zero #1\xint_gcd_end0\XINT_gcd_loop {#1#2}%
3346 }%
3347 \def\xint_gcd_end0\XINT_gcd_loop #1#2{ #2}%
```

```
#1=B, #2=A
```

```
3348 \def\XINT_gcd_loop #1#2%
3349 {%
3350     \expandafter\expandafter\expandafter
3351         \XINT_gcd_CheckRem
3352     \expandafter\xint_secondoftwo
3353     \romannumeral0\XINT_div_prepare {#1}{#2}\Z
3354     {#1}%
3355 }%
```

20.6 \xintBezout

```
3356 \def\xintBezout {\romannumeral0\xintbezout }%
3357 \def\xintbezout #1%
3358 {%
3359     \expandafter\xint_bezout\expandafter {\romannumeral-‘0#1}%
3360 }%
3361 \def\xint_bezout #1#2%
3362 {%
3363     \expandafter\XINT_bezout_fork \romannumeral-‘0#2\Z #1\Z
3364 }%
```

```
#3#4 = A, #1#2=B
```

```
3365 \def\XINT_bezout_fork #1#2\Z #3#4\Z
3366 {%
3367     \xint_UDzerosfork
3368      #1#3\dummy \XINT_bezout_botharezero
3369       #10\dummy \XINT_bezout_secondiszero
3370       #30\dummy \XINT_bezout_firstiszero
3371        00\dummy
3372         {\xint_UDsignsfork
3373           #1#3\dummy \XINT_bezout_minusminus % A < 0, B < 0
3374            #1-\dummy \XINT_bezout_minusplus  % A > 0, B < 0
3375            #3-\dummy \XINT_bezout_plusminus  % A < 0, B > 0
3376             --\dummy \XINT_bezout_plusplus   % A > 0, B > 0
3377          \krof }%
3378     \krof
```

```
    {#2}{#4}#1#3{#3#4}{#1#2}% #1#2=B, #3#4=A
}%
\def\XINT_bezout_botharezero #1#2#3#4#5#6%
{%
    \xintError:NoBezoutForZeros
    \space {0}{0}{0}{0}{0}%
}%

 attention première entrée doit être ici (-1)^n donc 1
 #4#2 = 0 = A, B = #3#1

\def\XINT_bezout_firstiszero #1#2#3#4#5#6%
{%
    \xint_UDsignfork
      #3\dummy { {0}{#3#1}{0}{1}{#1}}%
       -\dummy { {0}{#3#1}{0}{-1}{#1}}%
    \krof
}%

 #4#2 = A, B = #3#1 = 0

\def\XINT_bezout_secondiszero #1#2#3#4#5#6%
{%
    \xint_UDsignfork
       #4\dummy{ {#4#2}{0}{-1}{0}{#2}}%
        -\dummy{ {#4#2}{0}{1}{0}{#2}}%
    \krof
}%

 #4#2= A < 0, #3#1 = B < 0

\def\XINT_bezout_minusminus #1#2#3#4%
{%
    \expandafter\XINT_bezout_mm_post
    \romannumeral0\XINT_bezout_loop_a 1{#1}{#2}1001%
}%
\def\XINT_bezout_mm_post #1#2%
{%
    \expandafter\XINT_bezout_mm_postb\expandafter
    {\romannumeral0\xintiopp{#2}}{\romannumeral0\xintiopp{#1}}%
}%
\def\XINT_bezout_mm_postb #1#2%
{%
    \expandafter\XINT_bezout_mm_postc\expandafter {#2}{#1}%
}%
\def\XINT_bezout_mm_postc #1#2#3#4#5%
{%
    \space {#4}{#5}{#1}{#2}{#3}%
}%

 minusplus #4#2= A > 0, B < 0
```

```
\def\XINT_bezout_minusplus #1#2#3#4%
{%
    \expandafter\XINT_bezout_mp_post
    \romannumeral0\XINT_bezout_loop_a 1{#1}{#4#2}1001%
}%
\def\XINT_bezout_mp_post #1#2%
{%
    \expandafter\XINT_bezout_mp_postb\expandafter
      {\romannumeral0\xintiopp {#2}}{#1}%
}%
\def\XINT_bezout_mp_postb #1#2#3#4#5%
{%
    \space {#4}{#5}{#2}{#1}{#3}%
}%
```

plusminus A < 0, B > 0

```
\def\XINT_bezout_plusminus #1#2#3#4%
{%
    \expandafter\XINT_bezout_pm_post
    \romannumeral0\XINT_bezout_loop_a 1{#3#1}{#2}1001%
}%
\def\XINT_bezout_pm_post #1%
{%
    \expandafter \XINT_bezout_pm_postb \expandafter
        {\romannumeral0\xintiopp{#1}}%
}%
\def\XINT_bezout_pm_postb #1#2#3#4#5%
{%
    \space {#4}{#5}{#1}{#2}{#3}%
}%
```

plusplus

```
\def\XINT_bezout_plusplus #1#2#3#4%
{%
    \expandafter\XINT_bezout_pp_post
    \romannumeral0\XINT_bezout_loop_a 1{#3#1}{#4#2}1001%
}%
```

la parité (-1)^N est en #1, et on la jette ici.

```
\def\XINT_bezout_pp_post #1#2#3#4#5%
{%
    \space {#4}{#5}{#1}{#2}{#3}%
}%
```

n = 0: 1BAalpha(0)beta(0)alpha(-1)beta(-1)
n général: {(-1)^n}{r(n-1)}{r(n-2)}{alpha(n-1)}{beta(n-1)}{alpha(n-2)}{beta(n-2)}
#2 = B, #3 = A

```
\def\XINT_bezout_loop_a #1#2#3%
{%
    \expandafter\XINT_bezout_loop_b
    \expandafter{\the\numexpr -#1\expandafter }%
    \romannumeral0\XINT_div_prepare {#2}{#3}{#2}%
}%
```

Le q(n) a ici une existence éphémère, dans le version Bezout Algorithm il faudra le conserver. On voudra à la fin {{q(n)}{r(n)}{alpha(n)}{beta(n)}}. De plus ce n'est plus (-1)^n que l'on veut mais n. (ou dans un autre ordre) {-(-1)^n}{q(n)}{r(n)}{r(n-1)}{alpha(n-1)}{beta(n-1)}{alpha(n-2)}{beta(n-2)}

```
\def\XINT_bezout_loop_b #1#2#3#4#5#6#7#8%
{%
    \expandafter \XINT_bezout_loop_c \expandafter
        {\romannumeral0\xintiadd{\XINT_Mul{#5}{#2}}{#7}}%
        {\romannumeral0\xintiadd{\XINT_Mul{#6}{#2}}{#8}}%
    {#1}{#3}{#4}{#5}{#6}%
}%
```

{alpha(n)}{->beta(n)}{-(-1)^n}{r(n)}{r(n-1)}{alpha(n-1)}{beta(n-1)}

```
\def\XINT_bezout_loop_c #1#2%
{%
    \expandafter \XINT_bezout_loop_d \expandafter
        {#2}{#1}%
}%
```

{beta(n)}{alpha(n)}{(-1)^(n+1)}{r(n)}{r(n-1)}{alpha(n-1)}{beta(n-1)}

```
\def\XINT_bezout_loop_d #1#2#3#4#5%
{%
    \XINT_bezout_loop_e #4\Z {#3}{#5}{#2}{#1}%
}%
```

r(n)\Z {(-1)^(n+1)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}

```
\def\XINT_bezout_loop_e #1#2\Z
{%
    \xint_gob_til_zero #1\xint_bezout_loop_exit0\XINT_bezout_loop_f
    {#1#2}%
}%
```

{r(n)}{(-1)^(n+1)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}

```
\def\XINT_bezout_loop_f #1#2%
{%
    \XINT_bezout_loop_a {#2}{#1}%
}%
```

{(-1)^(n+1)}{r(n)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)} et itération

```
\def\xint_bezout_loop_exit0\XINT_bezout_loop_f #1#2%
```

```
{%
    \ifcase #2
    \or  \expandafter\XINT_bezout_exiteven
    \else\expandafter\XINT_bezout_exitodd
    \fi
}%
\def\XINT_bezout_exiteven #1#2#3#4#5%
{%
    \space {#5}{#4}{#1}%
}%
\def\XINT_bezout_exitodd #1#2#3#4#5%
{%
    \space {-#5}{-#4}{#1}%
}%
```

20.7 \xintEuclideAlgorithm

```
Pour Euclide: {N}{A}{D=r(n)}{B}{q1}{r1}{q2}{r2}{q3}{r3}....{qN}{rN=0}
u<2n> = u<2n+3>u<2n+2> + u<2n+4> à la n ième étape
```

```
\def\xintEuclideAlgorithm {\romannumeral0\xinteuclidealgorithm }%
\def\xinteuclidealgorithm #1%
{%
    \expandafter \XINT_euc \expandafter{\romannumeral0\xintiabs {#1}}%
}%
\def\XINT_euc #1#2%
{%
    \expandafter\XINT_euc_fork \romannumeral0\xintiabs {#2}\Z #1\Z
}%
```

```
Ici #3#4=A, #1#2=B
```

```
\def\XINT_euc_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\dummy \XINT_euc_BisZero
      #3\dummy \XINT_euc_AisZero
       0\dummy \XINT_euc_a
    \krof
    {0}{#1#2}{#3#4}{{#3#4}{#1#2}}{}\Z
}%
```

```
Le {} pour protéger {{A}{B}} si on s'arrête après une étape (B divise A). On
va renvoyer:
{N}{A}{D=r(n)}{B}{q1}{r1}{q2}{r2}{q3}{r3}....{qN}{rN=0}
```

```
\def\XINT_euc_AisZero #1#2#3#4#5#6{ {1}{0}{#2}{#2}{0}{0}}%
\def\XINT_euc_BisZero #1#2#3#4#5#6{ {1}{0}{#3}{#3}{0}{0}}%
```

```
 {n}{rn}{an}{{qn}{rn}}...{{A}{B}}{}\Z
 a(n) = r(n-1). Pour n=0 on a juste {0}{B}{A}{{A}{B}}{}\Z
 \XINT_div_prepare {u}{v} divise v par u
3521 \def\XINT_euc_a #1#2#3%
3522 {%
3523     \expandafter\XINT_euc_b
3524     \expandafter {\the\numexpr #1+1\expandafter }%
3525     \romannumeral0\XINT_div_prepare {#2}{#3}{#2}%
3526 }%
 {n+1}{q(n+1)}{r(n+1)}{rn}{{qn}{rn}}...
3527 \def\XINT_euc_b #1#2#3#4%
3528 {%
3529     \XINT_euc_c #3\Z {#1}{#3}{#4}{{#2}{#3}}%
3530 }%
 r(n+1)\Z {n+1}{r(n+1)}{r(n)}{{q(n+1)}{r(n+1)}}{{qn}{rn}}...
 Test si r(n+1) est nul.
3531 \def\XINT_euc_c #1#2\Z
3532 {%
3533     \xint_gob_til_zero #1\xint_euc_end0\XINT_euc_a
3534 }%
 {n+1}{r(n+1)}{r(n)}{{q(n+1)}{r(n+1)}}...{}\Z Ici r(n+1) = 0. On arrête on se
 prépare à inverser {n+1}{0}{r(n)}{{q(n+1)}{r(n+1)}}.....{{q1}{r1}}{{A}{B}}{}\Z
 On veut renvoyer: {N=n+1}{A}{D=r(n)}{B}{q1}{r1}{q2}{r2}{q3}{r3}....{qN}{rN=0}
3535 \def\xint_euc_end0\XINT_euc_a #1#2#3#4\Z%
3536 {%
3537     \expandafter\xint_euc_end_
3538     \romannumeral0%
3539     \XINT_rord_main {}#4{{#1}{#3}}%
3540     \xint_relax
3541       \xint_undef\xint_undef\xint_undef\xint_undef
3542       \xint_undef\xint_undef\xint_undef\xint_undef
3543     \xint_relax
3544 }%
3545 \def\xint_euc_end_ #1#2#3%
3546 {%
3547     \space {#1}{#3}{#2}%
3548 }%
```

20.8 \xintBezoutAlgorithm

```
 Pour Bezout: objectif, renvoyer
 {N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
```

```
 {q2}{r2}{alpha2}{beta2}....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}
 alpha0=1, beta0=0, alpha(-1)=0, beta(-1)=1

\def\xintBezoutAlgorithm {\romannumeral0\xintbezoutalgorithm }%
\def\xintbezoutalgorithm #1%
{%
    \expandafter \XINT_bezalg \expandafter{\romannumeral0\xintiabs {#1}}%
}%
\def\XINT_bezalg #1#2%
{%
    \expandafter\XINT_bezalg_fork \romannumeral0\xintiabs {#2}\Z #1\Z
}%

 Ici #3#4=A, #1#2=B

\def\XINT_bezalg_fork #1#2\Z #3#4\Z
{%
    \xint_UDzerofork
      #1\dummy \XINT_bezalg_BisZero
      #3\dummy \XINT_bezalg_AisZero
       0\dummy \XINT_bezalg_a
    \krof
    0{#1#2}{#3#4}1001{{#3#4}{#1#2}}{}\Z
}%
\def\XINT_bezalg_AisZero #1#2#3\Z{ {1}{0}{0}{1}{#2}{#2}{1}{0}{0}{0}{0}{1}}%
\def\XINT_bezalg_BisZero #1#2#3#4\Z{ {1}{0}{0}{1}{#3}{#3}{1}{0}{0}{0}{0}{1}}%

 pour préparer l'étape n+1 il faut {n}{r(n)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-
 1)}{beta(n-1)}{{q(n)}{r(n)}{alpha(n)}{beta(n)}}... division de #3 par #2

\def\XINT_bezalg_a #1#2#3%
{%
    \expandafter\XINT_bezalg_b
    \expandafter {\the\numexpr #1+1\expandafter }%
    \romannumeral0\XINT_div_prepare {#2}{#3}{#2}%
}%

 {n+1}{q(n+1)}{r(n+1)}{r(n)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}...

\def\XINT_bezalg_b #1#2#3#4#5#6#7#8%
{%
    \expandafter\XINT_bezalg_c\expandafter
     {\romannumeral0\xintiadd {\xintiMul {#6}{#2}}{#8}}%
     {\romannumeral0\xintiadd {\xintiMul {#5}{#2}}{#7}}%
     {#1}{#2}{#3}{#4}{#5}{#6}%
}%

 {beta(n+1)}{alpha(n+1)}{n+1}{q(n+1)}{r(n+1)}{r(n)}{alpha(n)}{beta(n)}

\def\XINT_bezalg_c #1#2#3#4#5#6%
```

```
{%
    \expandafter\XINT_bezalg_d\expandafter {#2}{#3}{#4}{#5}{#6}{#1}%
}%

 {alpha(n+1)}{n+1}{q(n+1)}{r(n+1)}{r(n)}{beta(n+1)}

\def\XINT_bezalg_d #1#2#3#4#5#6#7#8%
{%
    \XINT_bezalg_e #4\Z {#2}{#4}{#5}{#1}{#6}{#7}{#8}{{#3}{#4}{#1}{#6}}%
}%

 r(n+1)\Z {n+1}{r(n+1)}{r(n)}{alpha(n+1)}{beta(n+1)}
 {alpha(n)}{beta(n)}{q,r,alpha,beta(n+1)}
 Test si r(n+1) est nul.

\def\XINT_bezalg_e #1#2\Z
{%
    \xint_gob_til_zero #1\xint_bezalg_end0\XINT_bezalg_a
}%

 Ici r(n+1) = 0. On arrête on se prépare à inverser.
 {n+1}{r(n+1)}{r(n)}{alpha(n+1)}{beta(n+1)}{alpha(n)}{beta(n)}
 {q,r,alpha,beta(n+1)}...{{A}{B}}{}\Z
 On veut renvoyer
 {N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
 {q2}{r2}{alpha2}{beta2}....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}

\def\xint_bezalg_end0\XINT_bezalg_a #1#2#3#4#5#6#7#8\Z
{%
    \expandafter\xint_bezalg_end_
    \romannumeral0%
    \XINT_rord_main {}#8{{#1}{#3}}%
    \xint_relax
      \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_undef\xint_undef\xint_undef\xint_undef
    \xint_relax
}%

 {N}{D}{A}{B}{q1}{r1}{alpha1=q1}{beta1=1}{q2}{r2}{alpha2}{beta2}
 ....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}
 On veut renvoyer
 {N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
 {q2}{r2}{alpha2}{beta2}....{qN}{rN=0}{alphaN=A/D}{betaN=B/D}

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

20.9 \xintTypesetEuclideAlgorithm

```
TYPESETTING
  Organisation:
  {N}{A}{D}{B}{q1}{r1}{q2}{r2}{q3}{r3}....{qN}{rN=0}
\U1 = N = nombre d'étapes, \U3 = PGCD, \U2 = A, \U4=B q1 = \U5, q2 = \U7 -->
qn = \U<2n+3>, rn = \U<2n+4> bn = rn. B = r0. A=r(-1)
  r(n-2) = q(n)r(n-1)+r(n) (n e étape)
  \U{2n} = \U{2n+3} \times \U{2n+2} + \U{2n+4}, n e étape. (avec n entre 1 et
N)
```

```
\def\xintTypesetEuclideAlgorithm #1#2%
{% l'algo remplace #1 et #2 par |#1| et |#2|
  \par
  \begingroup
    \xintAssignArray\xintEuclideAlgorithm {#1}{#2}\to\U
    \edef\A{\U2}\edef\B{\U4}\edef\N{\U1}%
    \setbox 0 \vbox{\halign {$##$\cr \A\cr \B \cr}}%
    \noindent
    \count 255 1
    \loop
      \hbox to \wd 0 {\hfil$\U{\numexpr 2*\count 255\relax}$}%
      ${} =  \U{\numexpr 2*\count 255 + 3\relax}
      \times \U{\numexpr 2*\count 255 + 2\relax}
          +  \U{\numexpr 2*\count 255 + 4\relax}$%
    \ifnum \count 255 < \N
      \hfill\break
      \advance \count 255 1
    \repeat
  \par
  \endgroup
}%
```

20.10 \xintTypesetBezoutAlgorithm

```
Pour Bezout on a: {N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
{q2}{r2}{alpha2}{beta2}....{qN}{rN=0}{alphaN=A/D}{betaN=B/D} Donc 4N+8 termes:
U1 = N, U2= A, U5=D, U6=B, q1 = U9, qn = U{4n+5}, n au moins 1
rn = U{4n+6}, n au moins -1
alpha(n) = U{4n+7}, n au moins -1
beta(n) = U{4n+8}, n au moins -1
```

```
\def\xintTypesetBezoutAlgorithm #1#2%
{%
  \par
  \begingroup
    \parindent0pt
    \xintAssignArray\xintBezoutAlgorithm {#1}{#2}\to\BEZ
    \edef\A{\BEZ2}\edef\B{\BEZ6}\edef\N{\BEZ1}% A = |#1|, B = |#2|
```

```
    \setbox 0 \vbox{\halign {$##$\cr \A\cr \B \cr}}%
    \count 255 1
    \loop
      \noindent
      \hbox to \wd 0 {\hfil$\BEZ{4*\count 255 - 2}$}%
      ${} =  \BEZ{4*\count 255 + 5}
      \times \BEZ{4*\count 255 + 2}
          +  \BEZ{4*\count 255 + 6}$\hfill\break
      \hbox to \wd 0 {\hfil$\BEZ{4*\count 255 +7}$}%
      ${} = \BEZ{4*\count 255 + 5}
      \times \BEZ{4*\count 255 + 3}
          +  \BEZ{4*\count 255 - 1}$\hfill\break
      \hbox to \wd 0 {\hfil$\BEZ{4*\count 255 +8}$}%
      ${} =  \BEZ{4*\count 255 + 5}
      \times \BEZ{4*\count 255 + 4}
          +  \BEZ{4*\count 255 }$
      \endgraf
    \ifnum \count 255 < \N
    \advance \count 255 1
  \repeat
  \par
    \edef\U{\BEZ{4*\N + 4}}%
    \edef\V{\BEZ{4*\N + 3}}%
    \edef\D{\BEZ5}%
    \ifodd\N
       $\U\times\A  - \V\times \B = -\D$%
    \else
       $\U\times\A - \V\times\B = \D$%
    \fi
  \par
  \endgroup
}%
\XINT_gcd_restorecatcodes_endinput%
```

21 Package xintfrac implementation

The commenting is currently (2013/05/14) very sparse.

Contents

21.1 Catcodes, ε-TEX and reload detection

The code for reload detection is copied from Heiko Oberdiek's packages, and adapted here to check for previous loading of the master **xint** package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \def\space { }%
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintfrac.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintfrac}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintfrac.sty
```

```
      \ifx\w\relax % but xint.sty not yet loaded.
         \y{xintfrac}{Package xint is required}%
         \y{xintfrac}{Will try \string\input\space xint.sty}%
         \def\z{\endgroup\input xint.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xint.sty not yet loaded.
            \y{xintfrac}{Package xint is required}%
            \y{xintfrac}{Will try \string\RequirePackage{xint}}%
            \def\z{\endgroup\RequirePackage{xint}}%
          \fi
      \else
        \y{xintfrac}{I was already loaded, aborting input}%
        \aftergroup\endinput
      \fi
    \fi
  \fi
\z%
```

21.2 Confirmation of xint loading

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \def\empty {}%
  \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
  \ifx\w\relax % Plain TeX, user gave a file name at the prompt
      \y{xintfrac}{Loading of package xint failed, aborting input}%
      \aftergroup\endinput
  \fi
  \ifx\w\empty % LaTeX, user gave a file name at the prompt
      \y{xintfrac}{Loading of package xint failed, aborting input}%
```

```
      \aftergroup\endinput
  \fi
\endgroup%
```

21.3 Catcodes

Perhaps catcodes have changed after the loading of xint and prior to the current loading of xintfrac, so we can not employ the \XINT_restorecatcodes_endinput in this style file. But there is no problem using \XINT_setcatcodes.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode95=11   % _
  \def\x
  {%
      \endgroup
      \edef\XINT_frac_restorecatcodes_endinput
      {%
        \catcode94=\the\catcode94   % ^
        \catcode93=\the\catcode93   % ]
        \catcode91=\the\catcode91   % [
        \catcode96=\the\catcode96   % `
        \catcode47=\the\catcode47   % /
        \catcode41=\the\catcode41   % )
        \catcode40=\the\catcode40   % (
        \catcode42=\the\catcode42   % *
        \catcode43=\the\catcode43   % +
        \catcode62=\the\catcode62   % >
        \catcode60=\the\catcode60   % <
        \catcode58=\the\catcode58   % :
        \catcode46=\the\catcode46   % .
        \catcode45=\the\catcode45   % -
        \catcode44=\the\catcode44   % ,
        \catcode35=\the\catcode35   % #
        \catcode95=\the\catcode95   % _
        \catcode125=\the\catcode125 % }
        \catcode123=\the\catcode123 % {
        \endlinechar=\the\endlinechar
        \catcode13=\the\catcode13   % ^^M
        \catcode32=\the\catcode32   %
        \catcode61=\the\catcode61\relax   % =
        \noexpand\endinput
      }%
      \XINT_setcatcodes
      \catcode91=12 % [
      \catcode93=12 % ]
      \catcode94=7  % ^
```

```
  }%
\x
```

21.4 Package identification

```
\begingroup
  \catcode64=11 % @
  \catcode58=12 % :
  \expandafter\ifx\csname ProvidesPackage\endcsname\relax
    \def\x#1#2#3[#4]{\endgroup
      \immediate\write-1{Package: #3 #4}%
      \xdef#1{#4}%
    }%
  \else
    \def\x#1#2[#3]{\endgroup
      #2[{#3}]%
      \ifx#1\@undefined
        \xdef#1{#3}%
      \fi
      \ifx#1\relax
        \xdef#1{#3}%
      \fi
    }%
  \fi
\expandafter\x\csname ver@xintfrac.sty\endcsname
\ProvidesPackage{xintfrac}%
  [2013/05/14 v1.06b Expandable operations on fractions (jfB)]%
```

21.5 \xintLen

```
\def\xintLen {\romannumeral0\xintlen }%
\def\xintlen #1%
{%
    \expandafter\XINT_flen\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_flen #1#2#3%
{%
    \expandafter\space
    \the\numexpr -1+\XINT_Abs {#1}+\XINT_Len {#2}+\XINT_Len {#3}\relax
}%
```

21.6 \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 of course \xintIrr, \xintJrr, \xintRaw)

```
\def\XINT_outfrac #1#2#3%
{%
```

```
    \ifcase\XINT_Sgn{#3}
        \expandafter \XINT_outfrac_divisionbyzero
    \or
        \expandafter \XINT_outfrac_P
    \else
        \expandafter \XINT_outfrac_N
    \fi
    {#2}{#3}[#1]%
}%
\def\XINT_outfrac_divisionbyzero #1#2{\xintError:DivisionByZero\space #1/0}%
\def\XINT_outfrac_P #1#2%
{%
    \ifcase\XINT_Sgn{#1}
        \expandafter\XINT_outfrac_Zero
    \fi
    \space #1/#2%
}%
\def\XINT_outfrac_Zero #1[#2]{ 0/1[0]}%
\def\XINT_outfrac_N #1#2%
{%
    \expandafter\XINT_outfrac_N_a\expandafter
    {\romannumeral0\XINT_opp #2}{\romannumeral0\XINT_opp #1}%
}%
\def\XINT_outfrac_N_a #1#2%
{%
    \expandafter\XINT_outfrac_P\expandafter {#2}{#1}%
}%
```

21.7 \XINT_inFrac

```
\def\XINT_inFrac {\romannumeral0\XINT_infrac }%
\def\XINT_infrac #1%
{%
    \expandafter\XINT_infrac_ \romannumeral-`0#1[\W]\Z\T
}%
\def\XINT_infrac_ #1[#2#3]#4\Z
{%
    \xint_UDwfork
      #2\dummy \XINT_infrac_A
      \W\dummy \XINT_infrac_B
    \krof
    #1[#2#3]#4%
}%
\def\XINT_infrac_A #1[\W]\T
{%
    \XINT_frac #1/\W\Z
}%
\def\XINT_infrac_B #1%
{%
```

```
    \xint_gob_til_zero #1\XINT_infrac_Zero0\XINT_infrac_BB #1%
}%
\def\XINT_infrac_BB #1[\W]\T {\XINT_infrac_BC #1/\W\Z }%
\def\XINT_infrac_BC #1/#2#3\Z
{%
    \xint_UDwfork
     #2\dummy \XINT_infrac_BCa
     \W\dummy {\expandafter\XINT_infrac_BCb \romannumeral-`0#2}%
    \krof
    #3\Z #1\Z
}%
\def\XINT_infrac_BCa \Z #1[#2]#3\Z { {#2}{#1}{1}}%
\def\XINT_infrac_BCb #1[#2]/\W\Z #3\Z { {#2}{#3}{#1}}%
\def\XINT_infrac_Zero #1\T { {0}{0}{1}}%
```

21.8 \XINT_frac

```
\def\XINT_frac #1/#2#3\Z
{%
    \xint_UDwfork
     #2\dummy \XINT_frac_A
     \W\dummy {\expandafter\XINT_frac_B \romannumeral-`0#2}%
    \krof
    #3.\W\Z #1.\W\Z
}%
\def\XINT_frac_B #1.#2#3\Z
{%
    \xint_UDwfork
      #2\dummy \XINT_frac_Ba
      \W\dummy {\XINT_frac_Bb #2}%
    \krof
    #3\Z #1\Z
}%
\def\XINT_frac_Bb #1/\W.\W\Z #2\Z
{%
    \expandafter \XINT_frac_C \expandafter
    {\romannumeral0\XINT_length {#1}}{#2#1}%
}%
\def\XINT_frac_Ba \Z #1/\W\Z {\XINT_frac_C {0}{#1}}%
\def\XINT_frac_A .\W\Z {\XINT_frac_C {0}{1}}%
\def\XINT_frac_C #1#2#3.#4#5\Z
{%
    \xint_UDwfork
      #4\dummy \XINT_frac_Ca
      \W\dummy {\XINT_frac_Cb #4}%
    \krof
    #5\Z #3\Z {#1}{#2}%
}%
\def\XINT_frac_Ca \Z #1\Z {\XINT_frac_D {0}{#1}}%
\def\XINT_frac_Cb #1.\W\Z #2\Z
```

```
{%
    \expandafter\XINT_frac_D\expandafter
    {\romannumeral0\XINT_length {#1}}{#2#1}%
}%
\def\XINT_frac_D #1#2#3#4%
{%
    \expandafter \XINT_frac_E \expandafter
    {\the\numexpr -#1+#3\expandafter}\expandafter
    {\romannumeral0\XINT_num_loop #2%
     \xint_relax\xint_relax\xint_relax\xint_relax
     \xint_relax\xint_relax\xint_relax\xint_relax\Z }%
    {\romannumeral0\XINT_num_loop #4%
     \xint_relax\xint_relax\xint_relax\xint_relax
     \xint_relax\xint_relax\xint_relax\xint_relax\Z }%
}%
\def\XINT_frac_E #1#2#3%
{%
   \expandafter \XINT_frac_F  #3\Z {#2}{#1}%
}%
\def\XINT_frac_F #1%
{%
    \xint_UDzerominusfork
      #1-\dummy  \XINT_frac_Gdivisionbyzero
      0#1\dummy  \XINT_frac_Gneg
       0-\dummy  {\XINT_frac_Gpos #1}%
    \krof
}%
\def\XINT_frac_Gdivisionbyzero #1\Z #2#3%
{%
   \xintError:DivisionByZero\space {0}{#2}{0}%
}%
\def\XINT_frac_Gneg #1\Z #2#3%
{%
    \expandafter\XINT_frac_H \expandafter{\romannumeral0\XINT_opp #2}{#3}{#1}%
}%
\def\XINT_frac_H #1#2{ {#2}{#1}}%
\def\XINT_frac_Gpos #1\Z #2#3{ {#3}{#2}{#1}}%
```

21.9 \XINT_factortens, \XINT_cuz_cnt

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

```
\def\XINT_cuz_cnt #1%
{%
    \XINT_cuz_cnt_loop {}#1\R\R\R\R\R\R\R\Z
}%
\def\XINT_cuz_cnt_loop #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_r #9\XINT_cuz_cnt_toofara \R
    \expandafter\XINT_cuz_cnt_checka\expandafter
    {\the\numexpr #1+8\relax}{#2#3#4#5#6#7#8#9}%
}%
\def\XINT_cuz_cnt_toofara\R
    \expandafter\XINT_cuz_cnt_checka\expandafter #1#2%
{%
    \XINT_cuz_cnt_toofarb {#1}#2%
}%
\def\XINT_cuz_cnt_toofarb #1#2\Z {\XINT_cuz_cnt_toofarc #2\Z {#1}}%
\def\XINT_cuz_cnt_toofarc #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_r #2\XINT_cuz_cnt_toofard 7%
            #3\XINT_cuz_cnt_toofard 6%
            #4\XINT_cuz_cnt_toofard 5%
            #5\XINT_cuz_cnt_toofard 4%
            #6\XINT_cuz_cnt_toofard 3%
            #7\XINT_cuz_cnt_toofard 2%
            #8\XINT_cuz_cnt_toofard 1%
            \Z #1#2#3#4#5#6#7#8%
}%
\def\XINT_cuz_cnt_toofard #1#2\Z #3\R #4\Z #5%
{%
    \expandafter\XINT_cuz_cnt_toofare
    \the\numexpr #3\relax \R\R\R\R\R\R\R\Z
    {\the\numexpr #5-#1\relax}\R\Z
}%
\def\XINT_cuz_cnt_toofare #1#2#3#4#5#6#7#8%
{%
    \xint_gob_til_r #2\XINT_cuz_cnt_stopc 1%
            #3\XINT_cuz_cnt_stopc 2%
            #4\XINT_cuz_cnt_stopc 3%
            #5\XINT_cuz_cnt_stopc 4%
            #6\XINT_cuz_cnt_stopc 5%
            #7\XINT_cuz_cnt_stopc 6%
            #8\XINT_cuz_cnt_stopc 7%
            \Z #1#2#3#4#5#6#7#8%
}%
\def\XINT_cuz_cnt_checka #1#2%
{%
    \expandafter\XINT_cuz_cnt_checkb\the\numexpr #2\relax \Z {#1}%
}%
\def\XINT_cuz_cnt_checkb #1%
```

```
{%
    \xint_gob_til_zero #1\expandafter\XINT_cuz_cnt_loop\xint_gob_til_z
    0\XINT_cuz_cnt_stopa #1%
}%
\def\XINT_cuz_cnt_stopa #1\Z
{%
    \XINT_cuz_cnt_stopb #1\R\R\R\R\R\R\R\Z %
}%
\def\XINT_cuz_cnt_stopb #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_r #2\XINT_cuz_cnt_stopc 1%
            #3\XINT_cuz_cnt_stopc 2%
            #4\XINT_cuz_cnt_stopc 3%
            #5\XINT_cuz_cnt_stopc 4%
            #6\XINT_cuz_cnt_stopc 5%
            #7\XINT_cuz_cnt_stopc 6%
            #8\XINT_cuz_cnt_stopc 7%
            #9\XINT_cuz_cnt_stopc 8%
            \Z #1#2#3#4#5#6#7#8#9%
}%
\def\XINT_cuz_cnt_stopc #1#2\Z #3\R #4\Z #5%
{%
    \expandafter\XINT_cuz_cnt_stopd\expandafter
    {\the\numexpr #5-#1}#3%
}%
\def\XINT_cuz_cnt_stopd #1#2\R #3\Z
{%
    \expandafter\space\expandafter
     {\romannumeral0\XINT_rord_main {}#2%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax }{#1}%
}%
```

21.10 \xintRaw

```
\def\xintRaw {\romannumeral0\xintraw }%
\def\xintraw
{%
    \expandafter\XINT_raw\romannumeral0\XINT_infrac
}%
\def\XINT_raw #1%
{%
    \ifcase\XINT_Sgn {#1}
      \expandafter\XINT_raw_Ba
    \or
      \expandafter\XINT_raw_A
    \else
      \expandafter\XINT_raw_Ba
```

```
    \fi
    {#1}%
}%
\def\XINT_raw_A  #1#2#3{\xint_dsh {#2}{-#1}/#3}%
\def\XINT_raw_Ba #1#2#3{\expandafter\XINT_raw_Bb
                        \expandafter{\romannumeral0\xint_dsh {#3}{#1}}{#2}}%
\def\XINT_raw_Bb #1#2{ #2/#1}%
```

21.11 \xintNumerator

```
\def\xintNumerator {\romannumeral0\xintnumerator }%
\def\xintnumerator
{%
    \expandafter\XINT_numer\romannumeral0\XINT_infrac
}%
\def\XINT_numer #1%
{%
    \ifcase\XINT_Sgn {#1}
      \expandafter\XINT_numer_B
    \or
      \expandafter\XINT_numer_A
    \else
      \expandafter\XINT_numer_B
    \fi
    {#1}%
}%
\def\XINT_numer_A #1#2#3{\xint_dsh {#2}{-#1}}%
\def\XINT_numer_B #1#2#3{ #2}%
```

21.12 \xintDenominator

```
\def\xintDenominator {\romannumeral0\xintdenominator }%
\def\xintdenominator
{%
    \expandafter\XINT_denom\romannumeral0\XINT_infrac
}%
\def\XINT_denom #1%
{%
    \ifcase\XINT_Sgn {#1}
      \expandafter\XINT_denom_B
    \or
      \expandafter\XINT_denom_A
    \else
      \expandafter\XINT_denom_B
    \fi
    {#1}%
}%
\def\XINT_denom_A #1#2#3{ #3}%
\def\XINT_denom_B #1#2#3{\xint_dsh {#3}{#1}}%
```

21.13 \xintFrac

```
\def\xintFrac {\romannumeral0\xintfrac }%
\def\xintfrac #1%
{%
    \expandafter\XINT__frac_A\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT__frac_A #1{\XINT__frac_B #1\Z }%
\def\XINT__frac_B #1#2\Z
{%
    \xint_gob_til_zero #1\XINT__frac_C 0\XINT__frac_D {10^{#1#2}}%
}%
\def\XINT__frac_C #1#2#3#4#5%
{%
    \ifcase\XINT_isOne {#5}
    \or \xint_afterfi {\expandafter\xint_firstoftwo_andstop\xint_gobble_ii }%
    \fi
    \space
    \frac {#4}{#5}%
}%
\def\XINT__frac_D #1#2#3%
{%
    \ifcase\XINT_isOne {#3}
    \or \XINT__frac_E
    \fi
    \space
    \frac {#2}{#3}#1%
}%
\def\XINT__frac_E \fi #1#2#3#4{\fi \space #3\cdot }%
```

21.14 \xintSignedFrac

```
\def\xintSignedFrac {\romannumeral0\xintsignedfrac }%
\def\xintsignedfrac #1%
{%
    \expandafter\XINT_sgnfrac_a\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_sgnfrac_a #1#2%
{%
    \XINT_sgnfrac_b #2\Z {#1}%
}%
\def\XINT_sgnfrac_b #1%
{%
    \xint_UDsignfork
      #1\dummy \XINT_sgnfrac_N
       -\dummy {\XINT_sgnfrac_P #1}%
    \krof
}%
\def\XINT_sgnfrac_P #1\Z #2%
{%
```

```
    \XINT__frac_A {#2}{#1}%
}%
\def\XINT_sgnfrac_N
{%
    \expandafter\xint_minus_andstop\romannumeral0\XINT_sgnfrac_P
}%
```

21.15 \xintFwOver

```
\def\xintFwOver {\romannumeral0\xintfwover }%
\def\xintfwover #1%
{%
    \expandafter\XINT_fwover_A\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_fwover_A #1{\XINT_fwover_B #1\Z }%
\def\XINT_fwover_B #1#2\Z
{%
    \xint_gob_til_zero #1\XINT_fwover_C 0\XINT_fwover_D {10^{#1#2}}%
}%
\def\XINT_fwover_C #1#2#3#4#5%
{%
    \ifcase\XINT_isOne {#5}
       \xint_afterfi { {#4\over #5}}%
    \or
       \xint_afterfi { #4}%
    \fi
}%
\def\XINT_fwover_D #1#2#3%
{%
    \ifcase\XINT_isOne {#3}
      \xint_afterfi { {#2\over #3}}%
    \or
      \xint_afterfi { #2\cdot }%
    \fi
    #1%
}%
```

21.16 \xintSignedFwOver

```
\def\xintSignedFwOver {\romannumeral0\xintsignedfwover }%
\def\xintsignedfwover #1%
{%
    \expandafter\XINT_sgnfwover_a\romannumeral0\XINT_infrac {#1}%
}%
\def\XINT_sgnfwover_a #1#2%
{%
    \XINT_sgnfwover_b #2\Z {#1}%
}%
\def\XINT_sgnfwover_b #1%
{%
    \xint_UDsignfork
```

```
4190       #1\dummy \XINT_sgnfwover_N
4191         -\dummy {\XINT_sgnfwover_P #1}%
4192     \krof
4193 }%
4194 \def\XINT_sgnfwover_P #1\Z #2%
4195 {%
4196     \XINT_fwover_A {#2}{#1}%
4197 }%
4198 \def\XINT_sgnfwover_N
4199 {%
4200     \expandafter\xint_minus_andstop\romannumeral0\XINT_sgnfwover_P
4201 }%
```

21.17 \xintREZ

```
4202 \def\xintREZ {\romannumeral0\xintrez }%
4203 \def\xintrez
4204 {%
4205     \expandafter\XINT_rez_A\romannumeral0\XINT_infrac
4206 }%
4207 \def\XINT_rez_A #1#2%
4208 {%
4209     \XINT_rez_AB #2\Z {#1}%
4210 }%
4211 \def\XINT_rez_AB #1%
4212 {%
4213     \xint_UDzerominusfork
4214       #1-\dummy \XINT_rez_zero
4215       0#1\dummy \XINT_rez_neg
4216         0-\dummy {\XINT_rez_B #1}%
4217     \krof
4218 }%
4219 \def\XINT_rez_zero #1\Z #2#3{ 0/1[0]}%
4220 \def\XINT_rez_neg {\expandafter\xint_minus_andstop\romannumeral0\XINT_rez_B }%
4221 \def\XINT_rez_B #1\Z
4222 {%
4223     \expandafter\XINT_rez_C\romannumeral0\XINT_factortens {#1}%
4224 }%
4225 \def\XINT_rez_C #1#2#3#4%
4226 {%
4227     \expandafter\XINT_rez_D\romannumeral0\XINT_factortens {#4}{#3}{#2}{#1}%
4228 }%
4229 \def\XINT_rez_D #1#2#3#4#5%
4230 {%
4231     \expandafter\XINT_rez_E\expandafter
4232     {\the\numexpr #3+#4-#2}{#1}{#5}%
4233 }%
4234 \def\XINT_rez_E #1#2#3{ #3/#2[#1]}%
```

21.18 \xintIrr

1.04 fixes a buggy \xintIrr {0}. 1.05 modifies the initial parsing and post-processing to use \xintraw and to more quickly deal with an input denominator equal to 1.

```
\def\xintIrr {\romannumeral0\xintirr }%
\def\xintirr #1%
{%
    \expandafter\XINT_irr_start\romannumeral0\xintraw {#1}\Z
}%
\def\XINT_irr_start #1#2/#3\Z
{%
    \ifcase\XINT_isOne {#3}
      \xint_afterfi
          {\xint_UDsignfork
               #1\dummy \XINT_irr_negative
                -\dummy {\XINT_irr_nonneg #1}%
           \krof}%
    \or
      \xint_afterfi{\XINT_irr_denomisone #1}%
    \fi
    #2\Z {#3}%
}%
\def\XINT_irr_denomisone #1\Z #2{ #1}%
\def\XINT_irr_negative   #1\Z #2{\XINT_irr_D #1\Z #2\Z \xint_minus_andstop}%
\def\XINT_irr_nonneg     #1\Z #2{\XINT_irr_D #1\Z #2\Z \space}%
\def\XINT_irr_D #1#2\Z #3#4\Z
{%
    \xint_UDzerosfork
       #3#1\dummy \XINT_irr_indeterminate
       #30\dummy  \XINT_irr_divisionbyzero
       #10\dummy  \XINT_irr_zero
        00\dummy  \XINT_irr_loop_a
    \krof
    {#3#4}{#1#2}{#3#4}{#1#2}%
}%
\def\XINT_irr_indeterminate #1#2#3#4#5{\xintError:NaN\space 0/0}%
\def\XINT_irr_divisionbyzero #1#2#3#4#5{\xintError:DivisionByZero #5#2/0}%
\def\XINT_irr_zero #1#2#3#4#5{ 0}%
\def\XINT_irr_loop_a #1#2%
{%
    \expandafter\XINT_irr_loop_d
    \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
}%
\def\XINT_irr_loop_d #1#2%
{%
    \XINT_irr_loop_e #2\Z
}%
```

```
\def\XINT_irr_loop_e #1#2\Z
{%
    \xint_gob_til_zero #1\xint_irr_loop_exit0\XINT_irr_loop_a {#1#2}%
}%
\def\xint_irr_loop_exit0\XINT_irr_loop_a #1#2#3#4%
{%
    \expandafter\XINT_irr_loop_exitb\expandafter
    {\romannumeral0\xintiquo {#3}{#2}}%
    {\romannumeral0\xintiquo {#4}{#2}}%
}%
\def\XINT_irr_loop_exitb #1#2%
{%
   \expandafter\XINT_irr_finish\expandafter {#2}{#1}%
}%
\def\XINT_irr_finish #1#2#3%
{%
    \ifcase\XINT_isOne {#2}
      \xint_afterfi {#3#1/#2}%
    \or
      \xint_afterfi {#3#1}%
    \fi
}%
```

21.19 \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 \xintraw). This way, macros such as \xintQuo can be modified with minimal
overhead to accept fractional input as long as it evaluates to an integer.
```

```
\def\xintNum {\romannumeral0\xintnum }%
\def\xintnum #1{\expandafter\XINT_intcheck\romannumeral0\xintirr {#1}/\W\Z }%
\def\XINT_intcheck #1/#2#3\Z
{%
    \xint_gob_til_w #2\xint_gobble_ii\W
    \xintError:NotAnInteger
    \space #1%
}%
```

21.20 \xintJrr

```
Modified similarly as \xintIrr in release 1.05
```

```
\def\xintJrr {\romannumeral0\xintjrr }%
\def\xintjrr #1%
{%
```

```
    \expandafter\XINT_jrr_start\romannumeral0\xintraw {#1}\Z
}%
\def\XINT_jrr_start #1#2/#3\Z
{%
    \ifcase\XINT_isOne {#3}
      \xint_afterfi
          {\xint_UDsignfork
               #1\dummy \XINT_jrr_negative
                -\dummy {\XINT_jrr_nonneg #1}%
           \krof}%
    \or
      \xint_afterfi{\XINT_jrr_denomisone #1}%
    \fi
    #2\Z {#3}%
}%
\def\XINT_jrr_denomisone #1\Z #2{ #1}%
\def\XINT_jrr_negative   #1\Z #2{\XINT_jrr_D #1\Z #2\Z \xint_minus_andstop }%
\def\XINT_jrr_nonneg     #1\Z #2{\XINT_jrr_D #1\Z #2\Z \space}%
\def\XINT_jrr_D #1#2\Z #3#4\Z
{%
    \xint_UDzerosfork
       #3#1\dummy \XINT_jrr_indeterminate
       #30\dummy  \XINT_jrr_divisionbyzero
       #10\dummy  \XINT_jrr_zero
        00\dummy  \XINT_jrr_loop_a
    \krof
    {#3#4}{#1#2}1001%
}%
\def\XINT_jrr_indeterminate #1#2#3#4#5#6#7{\xintError:NaN\space 0/0}%
\def\XINT_jrr_divisionbyzero #1#2#3#4#5#6#7{\xintError:DivisionByZero #7#2/0}%
\def\XINT_jrr_zero #1#2#3#4#5#6#7{ 0}%
\def\XINT_jrr_loop_a #1#2%
{%
    \expandafter\XINT_jrr_loop_b
    \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
}%
\def\XINT_jrr_loop_b #1#2#3#4#5#6#7%
{%
    \expandafter \XINT_jrr_loop_c \expandafter
        {\romannumeral0\xintiadd{\XINT_Mul{#4}{#1}}{#6}}%
        {\romannumeral0\xintiadd{\XINT_Mul{#5}{#1}}{#7}}%
    {#2}{#3}{#4}{#5}%
}%
\def\XINT_jrr_loop_c #1#2%
{%
    \expandafter \XINT_jrr_loop_d \expandafter{#2}{#1}%
}%
\def\XINT_jrr_loop_d #1#2#3#4%
{%
```

```
    \XINT_jrr_loop_e #3\Z {#4}{#2}{#1}%
}%
\def\XINT_jrr_loop_e #1#2\Z
{%
    \xint_gob_til_zero #1\xint_jrr_loop_exit0\XINT_jrr_loop_a {#1#2}%
}%
\def\xint_jrr_loop_exit0\XINT_jrr_loop_a #1#2#3#4#5#6%
{%
    \XINT_irr_finish {#3}{#4}%
}%
```

21.21 \xintTrunc, \xintiTrunc

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

```
\def\xintTrunc {\romannumeral0\xinttrunc }%
\def\xintiTrunc {\romannumeral0\xintitrunc }%
\def\xinttrunc #1%
{%
    \expandafter\XINT_trunc\expandafter {\the\numexpr #1}%
}%
\def\XINT_trunc #1#2%
{%
    \expandafter\XINT_trunc_G
    \romannumeral0\expandafter\XINT_trunc_A
    \romannumeral0\XINT_infrac {#2}{#1}{#1}%
}%
\def\xintitrunc #1%
{%
    \expandafter\XINT_itrunc\expandafter {\the\numexpr #1}%
}%
\def\XINT_itrunc #1#2%
{%
    \expandafter\XINT_itrunc_G
    \romannumeral0\expandafter\XINT_trunc_A
    \romannumeral0\XINT_infrac {#2}{#1}{#1}%
}%
\def\XINT_trunc_A #1#2#3#4%
{%
    \expandafter\XINT_trunc_checkifzero
    \expandafter{\the\numexpr #1+#4}#2\Z {#3}%
}%
\def\XINT_trunc_checkifzero #1#2#3\Z
{%
    \xint_gob_til_zero #2\XINT_trunc_iszero0\XINT_trunc_B {#1}{#2#3}%
}%
\def\XINT_trunc_iszero #1#2#3#4#5{ 0\Z 0}%
\def\XINT_trunc_B #1%
{%
```

```
    \ifcase\XINT_Sgn {#1}
      \expandafter\XINT_trunc_D
    \or
      \expandafter\XINT_trunc_D
    \else
      \expandafter\XINT_trunc_C
    \fi
    {#1}%
}%
\def\XINT_trunc_C #1#2#3%
{%
    \expandafter \XINT_trunc_E
    \romannumeral0\xint_dsh {#3}{#1}\Z #2\Z
}%
\def\XINT_trunc_D #1#2%
{%
    \expandafter \XINT_trunc_DE \expandafter
    {\romannumeral0\xint_dsh {#2}{-#1}}%
}%
\def\XINT_trunc_DE #1#2{\XINT_trunc_E #2\Z #1\Z }%
\def\XINT_trunc_E #1#2\Z #3#4\Z
{%
    \xint_UDsignsfork
          #1#3\dummy \XINT_trunc_minusminus
           #1-\dummy {\XINT_trunc_minusplus #3}%
           #3-\dummy {\XINT_trunc_plusminus #1}%
            --\dummy {\XINT_trunc_plusplus  #3#1}%
    \krof
    {#4}{#2}%
}%
\def\XINT_trunc_minusminus #1#2{\xintiquo {#1}{#2}\Z \space}%
\def\XINT_trunc_minusplus #1#2#3{\xintiquo {#1#2}{#3}\Z \xint_minus_andstop}%
\def\XINT_trunc_plusminus #1#2#3{\xintiquo {#2}{#1#3}\Z \xint_minus_andstop}%
\def\XINT_trunc_plusplus  #1#2#3#4{\xintiquo {#1#3}{#2#4}\Z \space}%
\def\XINT_itrunc_G #1#2\Z #3#4%
{%
    \xint_gob_til_zero #1\XINT_trunc_zero 0\xint_firstoftwo {#3#1#2}0%
}%
\def\XINT_trunc_G #1\Z #2#3%
{%
    \xint_gob_til_zero #2\XINT_trunc_zero 0%
    \expandafter\XINT_trunc_H\expandafter
    {\the\numexpr\romannumeral0\XINT_length {#1}-#3}{#3}{#1}#2%
}%
\def\XINT_trunc_zero 0#10{ 0}%
\def\XINT_trunc_H #1#2%
{%
    \ifnum #1 > 0
        \xint_afterfi {\XINT_trunc_Ha {#2}}%
```

```
    \else
        \xint_afterfi {\XINT_trunc_Hb {-#1}}% -0,--1,--2, ....
    \fi
}%
\def\XINT_trunc_Ha
{%
  \expandafter\XINT_trunc_Haa\romannumeral0\xintdecsplit
}%
\def\XINT_trunc_Haa #1#2#3%
{%
    #3#1.#2%
}%
\def\XINT_trunc_Hb #1#2#3%
{%
    \expandafter #3\expandafter0\expandafter.%
    \romannumeral0\XINT_dsx_zeroloop {#1}\Z {}#2% #1=-0 possible!
}%
```

21.22 \xintRound, \xintiRound

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

```
\def\xintRound {\romannumeral0\xintround }%
\def\xintiRound {\romannumeral0\xintiround }%
\def\xintround #1%
{%
    \expandafter\XINT_round\expandafter {\the\numexpr #1}%
}%
\def\XINT_round
{%
    \expandafter\XINT_trunc_G\romannumeral0\XINT_round_A
}%
\def\xintiround #1%
{%
    \expandafter\XINT_iround\expandafter {\the\numexpr #1}%
}%
\def\XINT_iround
{%
    \expandafter\XINT_itrunc_G\romannumeral0\XINT_round_A
}%
\def\XINT_round_A #1#2%
{%
    \expandafter\XINT_round_B
    \romannumeral0\expandafter\XINT_trunc_A
    \romannumeral0\XINT_infrac {#2}{\the\numexpr #1+1\relax}{#1}%
}%
\def\XINT_round_B #1\Z
{%
    \expandafter\XINT_round_C
```

```
    \romannumeral0\XINT_rord_main {}#1%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax
    \Z
}%
\def\XINT_round_C #1%
{%
    \ifnum #1<5
        \expandafter\XINT_round_Daa
    \else
        \expandafter\XINT_round_Dba
    \fi
}%
\def\XINT_round_Daa #1%
{%
    \xint_gob_til_z #1\XINT_round_Daz\Z \XINT_round_Da #1%
}%
\def\XINT_round_Daz\Z \XINT_round_Da \Z { 0\Z }%
\def\XINT_round_Da #1\Z
{%
    \XINT_rord_main {}#1%
      \xint_relax
        \xint_undef\xint_undef\xint_undef\xint_undef
        \xint_undef\xint_undef\xint_undef\xint_undef
      \xint_relax  \Z
}%
\def\XINT_round_Dba #1%
{%
    \xint_gob_til_z #1\XINT_round_Dbz\Z \XINT_round_Db #1%
}%
\def\XINT_round_Dbz\Z \XINT_round_Db \Z { 1\Z }%
\def\XINT_round_Db #1\Z
{%
    \XINT_addm_A 0{}1000\W\X\Y\Z #1000\W\X\Y\Z \Z
}%
```

21.23 \xintAdd

```
\def\xintAdd {\romannumeral0\xintadd }%
\def\xintadd #1%
{%
    \expandafter\xint_fadd\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fadd #1#2{\expandafter\XINT_fadd_A\romannumeral0\XINT_infrac{#2}#1}%
\def\XINT_fadd_A #1#2#3#4%
{%
    \ifnum #4 > #1
```

```
4543        \xint_afterfi {\XINT_fadd_B {#1}}%
4544    \else
4545        \xint_afterfi {\XINT_fadd_B {#4}}%
4546    \fi
4547    {#1}{#4}{#2}{#3}%
4548 }%
4549 \def\XINT_fadd_B #1#2#3#4#5#6#7%
4550 {%
4551     \expandafter\XINT_fadd_C\expandafter
4552     {\romannumeral0\xintimul {#7}{#5}}%
4553     {\romannumeral0\xintiadd
4554     {\romannumeral0\xintimul {\xintDSH {\the\numexpr -#3+#1\relax}{#6}}{#5}}%
4555     {\romannumeral0\xintimul {#7}{\xintDSH {\the\numexpr -#2+#1\relax}{#4}}}%
4556     }%
4557     {#1}%
4558 }%
4559 \def\XINT_fadd_C #1#2#3%
4560 {%
4561     \expandafter\XINT_fadd_D\expandafter {#2}{#3}{#1}%
4562 }%
4563 \def\XINT_fadd_D #1#2{\XINT_outfrac {#2}{#1}}%
```

21.24 \xintSub

```
4564 \def\xintSub {\romannumeral0\xintsub }%
4565 \def\xintsub #1%
4566 {%
4567     \expandafter\xint_fsub\expandafter {\romannumeral0\XINT_infrac {#1}}%
4568 }%
4569 \def\xint_fsub #1#2%
4570    {\expandafter\XINT_fsub_A\romannumeral0\XINT_infrac {#2}#1}%
4571 \def\XINT_fsub_A #1#2#3#4%
4572 {%
4573     \ifnum #4 > #1
4574        \xint_afterfi {\XINT_fsub_B {#1}}%
4575     \else
4576        \xint_afterfi {\XINT_fsub_B {#4}}%
4577     \fi
4578     {#1}{#4}{#2}{#3}%
4579 }%
4580 \def\XINT_fsub_B #1#2#3#4#5#6#7%
4581 {%
4582     \expandafter\XINT_fsub_C\expandafter
4583     {\romannumeral0\xintimul {#7}{#5}}%
4584     {\romannumeral0\xintisub
4585     {\romannumeral0\xintimul {\xintDSH {\the\numexpr -#3+#1\relax}{#6}}{#5}}%
4586     {\romannumeral0\xintimul {#7}{\xintDSH {\the\numexpr -#2+#1\relax}{#4}}}%
4587     }%
4588     {#1}%
4589 }%
```

```
\def\XINT_fsub_C #1#2#3%
{%
    \expandafter\XINT_fsub_D\expandafter {#2}{#3}{#1}%
}%
\def\XINT_fsub_D #1#2{\XINT_outfrac {#2}{#1}}%
```

21.25 \xintSum, \xintSumExpr

```
\def\xintSum {\romannumeral0\xintsum }%
\def\xintsum #1{\xintsumexpr #1\relax }%
\def\xintSumExpr {\romannumeral0\xintsumexpr }%
\def\xintsumexpr {\expandafter\XINT_fsumexpr\romannumeral-`0}%
\def\XINT_fsumexpr {\XINT_fsum_loop_a {0/1[0]}}%
\def\XINT_fsum_loop_a #1#2%
{%
    \expandafter\XINT_fsum_loop_b \romannumeral-`0#2\Z {#1}%
}%
\def\XINT_fsum_loop_b #1%
{%
    \xint_gob_til_relax #1\XINT_fsum_finished\relax
    \XINT_fsum_loop_c #1%
}%
\def\XINT_fsum_loop_c #1\Z #2%
{%
    \expandafter\XINT_fsum_loop_a\expandafter{\romannumeral0\xintadd {#2}{#1}}%
}%
\def\XINT_fsum_finished #1\Z #2{ #2}%
```

21.26 \xintMul

```
\def\xintMul {\romannumeral0\xintmul }%
\def\xintmul #1%
{%
    \expandafter\xint_fmul\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fmul #1#2%
   {\expandafter\XINT_fmul_A\romannumeral0\XINT_infrac {#2}#1}%
\def\XINT_fmul_A #1#2#3#4#5#6%
{%
    \expandafter\XINT_fmul_B
    \expandafter{\the\numexpr #1+#4\expandafter}%
    \expandafter{\romannumeral0\xintimul {#6}{#3}}%
    {\romannumeral0\xintimul {#5}{#2}}%
}%
\def\XINT_fmul_B #1#2#3%
{%
    \expandafter \XINT_fmul_C \expandafter{#3}{#1}{#2}%
}%
\def\XINT_fmul_C #1#2{\XINT_outfrac {#2}{#1}}%
```

21.27 \xintSqr

```
\def\xintSqr {\romannumeral0\xintsqr }%
\def\xintsqr #1%
{%
    \expandafter\xint_fsqr\expandafter{\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fsqr #1{\XINT_fmul_A #1#1}%
```

21.28 \xintPow

```
Modified in 1.06 to give the exponent to a \numexpr
```

```
\def\xintPow {\romannumeral0\xintpow }%
\def\xintpow #1%
{%
    \expandafter\xint_fpow\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fpow #1#2%
{%
    \expandafter\XINT_fpow_fork\the\numexpr #2\relax\Z #1%
}%
\def\XINT_fpow_fork #1#2\Z
{%
    \xint_UDzerominusfork
      #1-\dummy  \XINT_fpow_zero
      0#1\dummy  \XINT_fpow_neg
       0-\dummy  {\XINT_fpow_pos #1}%
    \krof
    {#2}%
}%
\def\XINT_fpow_zero #1#2#3#4%
{%
    \space 1/1[0]%
}%
\def\XINT_fpow_pos #1#2#3#4#5%
{%
    \expandafter\XINT_fpow_pos_A\expandafter
    {\the\numexpr #1#2*#3\expandafter}\expandafter
    {\romannumeral0\xintipow {#5}{#1#2}}%
    {\romannumeral0\xintipow {#4}{#1#2}}%
}%
\def\XINT_fpow_neg #1#2#3#4%
{%
    \expandafter\XINT_fpow_pos_A\expandafter
    {\the\numexpr -#1*#2\expandafter}\expandafter
    {\romannumeral0\xintipow {#3}{#1}}%
    {\romannumeral0\xintipow {#4}{#1}}%
}%
\def\XINT_fpow_pos_A #1#2#3%
```

```
{%
    \expandafter\XINT_fpow_pos_B\expandafter {#3}{#1}{#2}%
}%
\def\XINT_fpow_pos_B #1#2{\XINT_outfrac {#2}{#1}}%
```

21.29 \xintPrd, \xintPrdExpr

```
\def\xintPrd {\romannumeral0\xintprd }%
\def\xintprd #1{\xintprdexpr #1\relax }%
\def\xintPrdExpr {\romannumeral0\xintprdexpr }%
\def\xintprdexpr {\expandafter\XINT_fprdexpr \romannumeral-`0}%
\def\XINT_fprdexpr {\XINT_fprod_loop_a {1/1[0]}}%
\def\XINT_fprod_loop_a #1#2%
{%
    \expandafter\XINT_fprod_loop_b \romannumeral-`0#2\Z {#1}%
}%
\def\XINT_fprod_loop_b #1%
{%
    \xint_gob_til_relax #1\XINT_fprod_finished\relax
    \XINT_fprod_loop_c #1%
}%
\def\XINT_fprod_loop_c #1\Z #2%
{%
  \expandafter\XINT_fprod_loop_a\expandafter{\romannumeral0\xintmul {#1}{#2}}%
}%
\def\XINT_fprod_finished #1\Z #2{ #2}%
```

21.30 \xintDiv

```
\def\xintDiv {\romannumeral0\xintdiv }%
\def\xintdiv #1%
{%
    \expandafter\xint_fdiv\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fdiv #1#2%
   {\expandafter\XINT_fdiv_A\romannumeral0\XINT_infrac {#2}#1}%
\def\XINT_fdiv_A #1#2#3#4#5#6%
{%
    \expandafter\XINT_fdiv_B
    \expandafter{\the\numexpr #4-#1\expandafter}%
    \expandafter{\romannumeral0\xintimul {#2}{#6}}%
    {\romannumeral0\xintimul {#3}{#5}}%
}%
\def\XINT_fdiv_B #1#2#3%
{%
    \expandafter\XINT_fdiv_C
    \expandafter{#3}{#1}{#2}%
}%
\def\XINT_fdiv_C #1#2{\XINT_outfrac {#2}{#1}}%
```

21.31 \xintCmp

```
\def\xintCmp {\romannumeral0\xintcmp }%
\def\xintcmp #1%
{%
    \expandafter\xint_fcmp\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fcmp #1#2{\expandafter\XINT_fcmp_A\romannumeral0\XINT_infrac {#2}#1}%
\def\XINT_fcmp_A #1#2#3#4%
{%
    \ifnum #4 > #1
       \xint_afterfi {\XINT_fcmp_B {#1}}%
    \else
       \xint_afterfi {\XINT_fcmp_B {#4}}%
    \fi
    {#1}{#4}{#2}{#3}%
}%
\def\XINT_fcmp_B #1#2#3#4#5#6#7%
{%
    \xinticmp
    {\romannumeral0\xintimul {\xintDSH {\the\numexpr -#3+#1\relax}{#6}}{#5}}%
    {\romannumeral0\xintimul {#7}{\xintDSH {\the\numexpr -#2+#1\relax}{#4}}}%
}%
```

21.32 \xintMax

```
\def\xintMax {\romannumeral0\xintmax }%
\def\xintmax #1%
{%
    \expandafter\xint_fmax\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fmax #1#2{\expandafter\XINT_outfrac
                    \romannumeral0\expandafter\XINT_fmax_A
                    \romannumeral0\XINT_infrac {#2}#1}%
\def\XINT_fmax_A #1#2#3#4#5#6%
{%
    \ifnum #4 > #1
       \xint_afterfi {\XINT_fmax_B {#1}}%
    \else
       \xint_afterfi {\XINT_fmax_B {#4}}%
    \fi
    {#1}{#4}{#2}{#3}{#5}{#6}{{#4}{#5}{#6}}{{#1}{#2}{#3}}%
}%
\def\XINT_fmax_B #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_fmax_C\expandafter
    {\romannumeral0\xintimul {\xintDSH {\the\numexpr -#3+#1\relax}{#6}}{#5}}%
    {\romannumeral0\xintimul {#7}{\xintDSH {\the\numexpr -#2+#1\relax}{#4}}}%
}%
\def\XINT_fmax_C #1#2%
```

```
{%
    \expandafter\XINT_max_fork #2\Z #1\Z
}%
```

21.33 \xintMin

```
\def\xintMin {\romannumeral0\xintmin }%
\def\xintmin #1%
{%
    \expandafter\xint_fmin\expandafter {\romannumeral0\XINT_infrac {#1}}%
}%
\def\xint_fmin #1#2%
{%
    \expandafter\XINT_outfrac
    \romannumeral0\expandafter\XINT_fmin_A
    \romannumeral0\XINT_infrac {#2}#1%
}%
\def\XINT_fmin_A #1#2#3#4#5#6%
{%
    \ifnum #4 > #1
       \xint_afterfi {\XINT_fmin_B {#1}}%
    \else
       \xint_afterfi {\XINT_fmin_B {#4}}%
    \fi
    {#1}{#4}{#2}{#3}{#5}{#6}{{#4}{#5}{#6}}{{#1}{#2}{#3}}%
}%
\def\XINT_fmin_B #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_fmin_C\expandafter
    {\romannumeral0\xintimul {\xintDSH {\the\numexpr -#3+#1\relax}{#6}}{#5}}%
    {\romannumeral0\xintimul {#7}{\xintDSH {\the\numexpr -#2+#1\relax}{#4}}}%
}%
\def\XINT_fmin_C #1#2%
{%
    \expandafter\XINT_min_fork #2\Z #1\Z
}%
```

21.34 \xintAbs

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

21.35 \xintOpp

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

21.36 \xintSgn

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

21.37 \xintGeq

```
\def\xintGeq {\romannumeral0\xintgeq }%
\def\xintgeq #1%
{%
    \expandafter\xint_xgeq\expandafter{\romannumeral0\xintnum {#1}}%
}%
\def\xint_xgeq #1#2%
{%
    \expandafter\XINT_geq_fork\romannumeral0\xintnum {#2}\Z #1\Z
}%
```

21.38 \xintDivision, \xintQuo, \xintRem

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

21.39 \xintFDg, \xintLDg, \xintMON, \xintMMON, \xintOdd

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

22 Package xintseries implementation

The commenting is currently (2013/05/14) very sparse.

Contents

22.1 Catcodes, ε-TEX and reload detection

The code for reload detection is copied from Heiko Oberdiek's packages, and adapted here to check for previous loading of the xintfrac package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \def\space { }%
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintseries.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintseries}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintseries.sty
      \ifx\w\relax % but xintfrac.sty not yet loaded.
         \y{xintseries}{Package xintfrac is required}%
         \y{xintseries}{Will try \string\input\space xintfrac.sty}%
         \def\z{\endgroup\input xintfrac.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
```

```
          \ifx\w\relax % xintfrac.sty not yet loaded.
            \y{xintseries}{Package xintfrac is required}%
            \y{xintseries}{Will try \string\RequirePackage{xintfrac}}%
            \def\z{\endgroup\RequirePackage{xintfrac}}%
          \fi
      \else
        \y{xintseries}{I was already loaded, aborting input}%
        \aftergroup\endinput
      \fi
    \fi
  \fi
\z%
```

22.2 Confirmation of xintfrac loading

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \def\empty {}%
  \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
  \ifx\w\relax % Plain TeX, user gave a file name at the prompt
      \y{xintseries}{Loading of package xintfrac failed, aborting input}%
      \aftergroup\endinput
  \fi
  \ifx\w\empty % LaTeX, user gave a file name at the prompt
      \y{xintseries}{Loading of package xintfrac failed, aborting input}%
      \aftergroup\endinput
  \fi
\endgroup%
```

22.3 Catcodes

Perhaps catcodes have changed after the loading of xint and xintfrac and prior to the current loading of xintseries, so we can not employ the `\XINT_restorecatcodes_endinput` in this style file. But there is no problem using `\XINT_setcatcodes`.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode95=11   % _
  \def\x
  {%
      \endgroup
      \edef\XINT_series_restorecatcodes_endinput
      {%
        \catcode93=\the\catcode93   % ]
        \catcode91=\the\catcode91   % [
        \catcode96=\the\catcode96   % `
        \catcode47=\the\catcode47   % /
        \catcode41=\the\catcode41   % )
        \catcode40=\the\catcode40   % (
        \catcode42=\the\catcode42   % *
        \catcode43=\the\catcode43   % +
        \catcode62=\the\catcode62   % >
        \catcode60=\the\catcode60   % <
        \catcode58=\the\catcode58   % :
        \catcode46=\the\catcode46   % .
        \catcode45=\the\catcode45   % -
        \catcode44=\the\catcode44   % ,
        \catcode35=\the\catcode35   % #
        \catcode95=\the\catcode95   % _
        \catcode125=\the\catcode125 % }
        \catcode123=\the\catcode123 % {
        \endlinechar=\the\endlinechar
        \catcode13=\the\catcode13   % ^^M
        \catcode32=\the\catcode32   %
        \catcode61=\the\catcode61\relax   % =
        \noexpand\endinput
      }%
      \XINT_setcatcodes
      \catcode91=12 % [
      \catcode93=12 % ]
  }%
\x
```

22.4 Package identification

```
\begingroup
  \catcode64=11 % @
  \catcode58=12 % :
  \expandafter\ifx\csname ProvidesPackage\endcsname\relax
    \def\x#1#2#3[#4]{\endgroup
      \immediate\write-1{Package: #3 #4}%
```

```
      \xdef#1{#4}%
    }%
  \else
    \def\x#1#2[#3]{\endgroup
      #2[{#3}]%
      \ifx#1\@undefined
        \xdef#1{#3}%
      \fi
      \ifx#1\relax
        \xdef#1{#3}%
      \fi
    }%
  \fi
\expandafter\x\csname ver@xintseries.sty\endcsname
\ProvidesPackage{xintseries}%
  [2013/05/14 v1.06b Expandable partial sums with xint package (jfB)]%
```

22.5 \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.

```
\def\xintSeries {\romannumeral0\xintseries }%
\def\xintseries #1#2%
{%
    \expandafter\XINT_series_i\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_series_i #1#2%
{%
    \expandafter\XINT_series_ii\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_series_ii #1#2#3%
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi {\XINT_series_loop {#1}{0}{#2}{#3}}%
   \fi
}%
\def\XINT_series_loop #1#2#3#4%
{%
    \ifnum #3>#1 \else \XINT_series_exit \fi
    \expandafter\XINT_series_loop\expandafter
    {\the\numexpr #1+1\expandafter }\expandafter
    {\romannumeral0\xintadd {#2}{#4{#1}}}%
    {#3}{#4}%
}%
\def\XINT_series_exit \fi #1#2#3#4#5#6#7#8%
{%
    \fi\xint_gobble_ii #6%
```

```
}%
```

22.6 \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.

```
\def\xintiSeries {\romannumeral0\xintiseries }%
\def\xintiseries #1#2%
{%
    \expandafter\XINT_iseries_i\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_iseries_i #1#2%
{%
    \expandafter\XINT_iseries_ii\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_iseries_ii #1#2#3%
{%
   \ifnum #2<#1
      \xint_afterfi { 0}%
   \else
      \xint_afterfi {\XINT_iseries_loop {#1}{0}{#2}{#3}}%
   \fi
}%
\def\XINT_iseries_loop #1#2#3#4%
{%
    \ifnum #3>#1 \else \XINT_iseries_exit \fi
    \expandafter\XINT_iseries_loop\expandafter
    {\the\numexpr #1+1\expandafter }\expandafter
    {\romannumeral0\xintiadd {#2}{#4{#1}}}%
    {#3}{#4}%
}%
\def\XINT_iseries_exit \fi #1#2#3#4#5#6#7#8%
{%
    \fi\xint_gobble_ii #6%
}%
```

22.7 \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.

```
\def\xintPowerSeries {\romannumeral0\xintpowerseries }%
\def\xintpowerseries #1#2%
{%
```

```
5086     \expandafter\XINT_powseries_i\expandafter {\the\numexpr #2}{#1}%
5087 }%
5088 \def\XINT_powseries_i #1#2%
5089 {%
5090     \expandafter\XINT_powseries_ii\expandafter {\the\numexpr #2}{#1}%
5091 }%
5092 \def\XINT_powseries_ii #1#2#3#4%
5093 {%
5094    \ifnum #2<#1
5095       \xint_afterfi { 0/1[0]}%
5096    \else
5097       \xint_afterfi
5098       {\XINT_powseries_loop_i {#3{#2}}{#1}{#2}{#3}{#4}}%
5099    \fi
5100 }%
5101 \def\XINT_powseries_loop_i #1#2#3#4#5%
5102 {%
5103     \ifnum #3>#2 \else\XINT_powseries_exit_i\fi
5104     \expandafter\XINT_powseries_loop_ii\expandafter
5105     {\the\numexpr #3-1\expandafter}\expandafter
5106     {\romannumeral0\xintmul {#1}{#5}}{#2}{#4}{#5}%
5107 }%
5108 \def\XINT_powseries_loop_ii #1#2#3#4%
5109 {%
5110    \expandafter\XINT_powseries_loop_i\expandafter
5111    {\romannumeral0\xintadd {#4{#1}}{#2}}{#3}{#1}{#4}%
5112 }%
5113 \def\XINT_powseries_exit_i\fi #1#2#3#4#5#6#7#8#9%
5114 {%
5115     \fi \XINT_powseries_exit_ii  #6{#7}%
5116 }%
5117 \def\XINT_powseries_exit_ii #1#2#3#4#5#6%
5118 {%
5119     \xintmul{\xintPow {#5}{#6}}{#4}%
5120 }%
```

22.8 \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.

```
5121 \def\xintPowerSeriesX {\romannumeral0\xintpowerseriesx }%
5122 \def\xintpowerseriesx #1#2%
5123 {%
5124     \expandafter\XINT_powseriesx_i\expandafter {\the\numexpr #2}{#1}%
5125 }%
5126 \def\XINT_powseriesx_i #1#2%
5127 {%
```

```
    \expandafter\XINT_powseriesx_ii\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_powseriesx_ii #1#2#3#4%
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi
      {\expandafter\XINT_powseriesx_pre\expandafter
                  {\romannumeral-`0#4}{#1}{#2}{#3}%
      }%
   \fi
}%
\def\XINT_powseriesx_pre #1#2#3#4%
{%
    \XINT_powseries_loop_i {#4{#3}}{#2}{#3}{#4}{#1}%
}%
```

22.9 \xintRationalSeries

This computes F(a)+...+F(b) on the basis of the value of F(a) and the ratios F(n)/F(n-1). As in \xintPowerSeries we use an iterative scheme which has the great advantage to avoid denominator build-up. This makes exact computations possible with exponential type series, which would be completely inaccessible to \xintSeries. #1=a, #2=b, #3=F(a), #4=ratio function Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

```
\def\xintRationalSeries {\romannumeral0\xintratseries }%
\def\xintratseries #1#2%
{%
    \expandafter\XINT_ratseries_i\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_ratseries_i #1#2%
{%
    \expandafter\XINT_ratseries_ii\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_ratseries_ii #1#2#3#4%
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi
      {\XINT_ratseries_loop {#2}{1}{#1}{#4}{#3}}%
   \fi
}%
\def\XINT_ratseries_loop #1#2#3#4%
{%
    \ifnum #1>#3 \else\XINT_ratseries_exit_i\fi
```

```
    \expandafter\XINT_ratseries_loop\expandafter
    {\the\numexpr #1-1\expandafter}\expandafter
    {\romannumeral0\xintadd {1}{\xintMul {#2}{#4{#1}}}}{#3}{#4}%
}%
\def\XINT_ratseries_exit_i\fi #1#2#3#4#5#6#7#8%
{%
    \fi \XINT_ratseries_exit_ii  #6%
}%
\def\XINT_ratseries_exit_ii #1#2#3#4#5%
{%
    \XINT_ratseries_exit_iii #5%
}%
\def\XINT_ratseries_exit_iii #1#2#3#4%
{%
    \xintmul{#2}{#4}%
}%
```

22.10 \xintRationalSeriesX

a,b,initial,ratiofunction,x
This computes F(a,x)+...+F(b,x) on the basis of the value of F(a,x) and the ratios F(n,x)/F(n-1,x). The argument x is first expanded and it is the value resulting from this which is used then throughout. The initial term F(a,x) must be defined as one-parameter macro which will be given x. Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

```
\def\xintRationalSeriesX {\romannumeral0\xintratseriesx }%
\def\xintratseriesx #1#2%
{%
    \expandafter\XINT_ratseriesx_i\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_ratseriesx_i #1#2%
{%
    \expandafter\XINT_ratseriesx_ii\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_ratseriesx_ii #1#2#3#4#5%
{%
   \ifnum #2<#1
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi
      {\expandafter\XINT_ratseriesx_pre\expandafter
                    {\romannumeral-`0#5}{#2}{#1}{#4}{#3}%
      }%
   \fi
}%
\def\XINT_ratseriesx_pre #1#2#3#4#5%
{%
```

```
    \XINT_ratseries_loop {#2}{1}{#3}{#4{#1}}{#5{#1}}%
}%
```

22.11 \xintFxPtPowerSeries

I am not two happy with this piece of code. Will make it more economical another day. Modified in 1.06 to give the indices first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

```
\def\xintFxPtPowerSeries {\romannumeral0\xintfxptpowerseries }%
\def\xintfxptpowerseries #1#2%
{%
    \expandafter\XINT_fppowseries_i\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_fppowseries_i #1#2%
{%
    \expandafter\XINT_fppowseries_ii\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_fppowseries_ii #1#2#3#4#5%
{%
   \ifnum #2<#1
      \xint_afterfi { 0}%
   \else
      \xint_afterfi
        {\expandafter\XINT_fppowseries_loop_pre\expandafter
           {\romannumeral0\xinttrunc {#5}{\xintPow {#4}{#1}}}%
         {#1}{#4}{#2}{#3}{#5}%
        }%
   \fi
}%
\def\XINT_fppowseries_loop_pre #1#2#3#4#5#6%
{%
    \ifnum #4>#2 \else\XINT_fppowseries_dont_i \fi
    \expandafter\XINT_fppowseries_loop_i\expandafter
    {\the\numexpr #2+1\expandafter}\expandafter
    {\romannumeral0\xintitrunc {#6}{\xintMul {#5{#2}}{#1}}}%
    {#1}{#3}{#4}{#5}{#6}%
}%
\def\XINT_fppowseries_dont_i \fi\expandafter\XINT_fppowseries_loop_i
    {\fi \expandafter\XINT_fppowseries_dont_ii }%
\def\XINT_fppowseries_dont_ii #1#2#3#4#5#6#7{\xinttrunc {#7}{#2[-#7]}}%
\def\XINT_fppowseries_loop_i #1#2#3#4#5#6#7%
{%
    \ifnum #5>#1 \else \XINT_fppowseries_exit_i \fi
    \expandafter\XINT_fppowseries_loop_ii\expandafter
    {\romannumeral0\xinttrunc {#7}{\xintMul {#3}{#4}}}%
    {#1}{#4}{#2}{#5}{#6}{#7}%
}%
```

```
\def\XINT_fppowseries_loop_ii #1#2#3#4#5#6#7%
{%
    \expandafter\XINT_fppowseries_loop_i\expandafter
    {\the\numexpr #2+1\expandafter}\expandafter
    {\romannumeral0\xintiadd {#4}{\xintiTrunc {#7}{\xintMul {#6{#2}}{#1}}}}%
    {#1}{#3}{#5}{#6}{#7}%
}%
\def\XINT_fppowseries_exit_i\fi\expandafter\XINT_fppowseries_loop_ii
    {\fi \expandafter\XINT_fppowseries_exit_ii }%
\def\XINT_fppowseries_exit_ii #1#2#3#4#5#6#7%
{%
    \xinttrunc {#7}
    {\xintiAdd {#4}{\xintiTrunc {#7}{\xintMul {#6{#2}}{#1}}}[-#7]}%
}%
```

22.12 \xintFxPtPowerSeriesX

```
a,b,coeff,x,D
Modified in 1.06 to give the indices first to a \numexpr rather than expand-
ing twice. I just use \the\numexpr and maintain the previous code after that.
```

```
\def\xintFxPtPowerSeriesX {\romannumeral0\xintfxptpowerseriesx }%
\def\xintfxptpowerseriesx #1#2%
{%
    \expandafter\XINT_fppowseriesx_i\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_fppowseriesx_i #1#2%
{%
    \expandafter\XINT_fppowseriesx_ii\expandafter {\the\numexpr #2}{#1}%
}%
\def\XINT_fppowseriesx_ii #1#2#3#4#5%
{%
   \ifnum #2<#1
      \xint_afterfi { 0}%
   \else
      \xint_afterfi
        {\expandafter \XINT_fppowseriesx_pre \expandafter
         {\romannumeral-`0#4}{#1}{#2}{#3}{#5}%
        }%
   \fi
}%
\def\XINT_fppowseriesx_pre #1#2#3#4#5%
{%
    \expandafter\XINT_fppowseries_loop_pre\expandafter
       {\romannumeral0\xinttrunc {#5}{\xintPow {#1}{#2}}}%
       {#2}{#1}{#3}{#4}{#5}%
}%
\XINT_series_restorecatcodes_endinput%
```

23 Package xintcfrac implementation

The commenting is currently (2013/05/14) very sparse.

Contents

23.1 Catcodes, ε-TEX and reload detection

The code for reload detection is copied from Heiko Oberdiek's packages, and adapted here to check for previous loading of the xintfrac package.

The method for catcodes is slightly different, but still directly inspired by these packages.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \def\space { }%
  \let\z\endgroup
  \expandafter\let\expandafter\x\csname ver@xintcfrac.sty\endcsname
  \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
```

```
    \fi
  \expandafter
  \ifx\csname numexpr\endcsname\relax
     \y{xintcfrac}{\numexpr not available, aborting input}%
     \aftergroup\endinput
  \else
    \ifx\x\relax   % plain-TeX, first loading of xintcfrac.sty
      \ifx\w\relax % but xintfrac.sty not yet loaded.
         \y{xintcfrac}{Package xintfrac is required}%
         \y{xintcfrac}{Will try \string\input\space xintfrac.sty}%
         \def\z{\endgroup\input xintfrac.sty\relax}%
      \fi
    \else
      \def\empty {}%
      \ifx\x\empty % LaTeX, first loading,
      % variable is initialized, but \ProvidesPackage not yet seen
          \ifx\w\relax % xintfrac.sty not yet loaded.
            \y{xintcfrac}{Package xintfrac is required}%
            \y{xintcfrac}{Will try \string\RequirePackage{xintfrac}}%
            \def\z{\endgroup\RequirePackage{xintfrac}}%
          \fi
      \else
        \y{xintcfrac}{I was already loaded, aborting input}%
        \aftergroup\endinput
      \fi
    \fi
  \fi
\z%
```

23.2 Confirmation of xintfrac loading

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode64=11   % @
  \catcode35=6    % #
  \catcode44=12   % ,
  \catcode45=12   % -
  \catcode46=12   % .
  \catcode58=12   % :
  \expandafter
    \ifx\csname PackageInfo\endcsname\relax
      \def\y#1#2{\immediate\write-1{Package #1 Info: #2.}}%
    \else
      \def\y#1#2{\PackageInfo{#1}{#2}}%
    \fi
  \def\empty {}%
```

```
  \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
  \ifx\w\relax % Plain TeX, user gave a file name at the prompt
      \y{xintcfrac}{Loading of package xintfrac failed, aborting input}%
      \aftergroup\endinput
  \fi
  \ifx\w\empty % LaTeX, user gave a file name at the prompt
      \y{xintcfrac}{Loading of package xintfrac failed, aborting input}%
      \aftergroup\endinput
  \fi
\endgroup%
```

23.3 Catcodes

Perhaps catcodes have changed after the loading of **xint** and **xintfrac** and prior to the current loading of **xintcfrac**, so we can not employ the `\XINT_restorecatcodes_endinput` in this style file. But there is no problem using `\XINT_setcatcodes`.

```
\begingroup\catcode61\catcode48\catcode32=10\relax%
  \catcode13=5    % ^^M
  \endlinechar=13 %
  \catcode123=1   % {
  \catcode125=2   % }
  \catcode95=11   % _
  \def\x
  {%
      \endgroup
      \edef\XINT_cfrac_restorecatcodes_endinput
      {%
        \catcode93=\the\catcode93   % ]
        \catcode91=\the\catcode91   % [
        \catcode96=\the\catcode96   % `
        \catcode47=\the\catcode47   % /
        \catcode41=\the\catcode41   % )
        \catcode40=\the\catcode40   % (
        \catcode42=\the\catcode42   % *
        \catcode43=\the\catcode43   % +
        \catcode62=\the\catcode62   % >
        \catcode60=\the\catcode60   % <
        \catcode58=\the\catcode58   % :
        \catcode46=\the\catcode46   % .
        \catcode45=\the\catcode45   % -
        \catcode44=\the\catcode44   % ,
        \catcode35=\the\catcode35   % #
        \catcode95=\the\catcode95   % _
        \catcode125=\the\catcode125 % }
        \catcode123=\the\catcode123 % {
        \endlinechar=\the\endlinechar
        \catcode13=\the\catcode13   % ^^M
        \catcode32=\the\catcode32   %
        \catcode61=\the\catcode61\relax   % =
```

```
5395         \noexpand\endinput
5396       }%
5397       \XINT_setcatcodes
5398       \catcode91=12 % [
5399       \catcode93=12 % ]
5400   }%
5401 \x
```

23.4 Package identification

```
5402 \begingroup
5403   \catcode64=11 % @
5404   \catcode58=12 % :
5405   \expandafter\ifx\csname ProvidesPackage\endcsname\relax
5406     \def\x#1#2#3[#4]{\endgroup
5407       \immediate\write-1{Package: #3 #4}%
5408       \xdef#1{#4}%
5409     }%
5410   \else
5411     \def\x#1#2[#3]{\endgroup
5412       #2[{#3}]%
5413       \ifx#1\@undefined
5414         \xdef#1{#3}%
5415       \fi
5416       \ifx#1\relax
5417         \xdef#1{#3}%
5418       \fi
5419     }%
5420   \fi
5421 \expandafter\x\csname ver@xintcfrac.sty\endcsname
5422 \ProvidesPackage{xintcfrac}%
5423   [2013/05/14 v1.06b Expandable continued fractions with xint package (jfB)]%
```

23.5 \xintCFrac

```
5424 \def\xintCFrac {\romannumeral0\xintcfrac }%
5425 \def\xintcfrac #1%
5426 {%
5427     \XINT_cfrac_opt_a #1\Z
5428 }%
5429 \def\XINT_cfrac_opt_a #1%
5430 {%
5431     \ifx#1[\XINT_cfrac_opt_b\fi \XINT_cfrac_noopt #1%
5432 }%
5433 \def\XINT_cfrac_noopt #1\Z
5434 {%
5435     \expandafter\XINT_cfrac_A\romannumeral0\xintraw {#1}\Z
5436     \relax\relax
5437 }%
5438 \def\XINT_cfrac_opt_b\fi\XINT_cfrac_noopt [\Z #1]%
```

```
{%
    \fi\csname XINT_cfrac_opt#1\endcsname
}%
\def\XINT_cfrac_optl #1%
{%
    \expandafter\XINT_cfrac_A\romannumeral0\xintraw {#1}\Z
    \relax\hfill
}%
\def\XINT_cfrac_optc #1%
{%
    \expandafter\XINT_cfrac_A\romannumeral0\xintraw {#1}\Z
    \relax\relax
}%
\def\XINT_cfrac_optr #1%
{%
    \expandafter\XINT_cfrac_A\romannumeral0\xintraw {#1}\Z
    \hfill\relax
}%
\def\XINT_cfrac_A #1/#2\Z
{%
    \expandafter\XINT_cfrac_B\romannumeral0\xintidivision {#1}{#2}{#2}%
}%
\def\XINT_cfrac_B #1#2%
{%
    \XINT_cfrac_C #2\Z {#1}%
}%
\def\XINT_cfrac_C #1%
{%
    \xint_gob_til_zero #1\XINT_cfrac_integer 0\XINT_cfrac_D #1%
}%
\def\XINT_cfrac_integer 0\XINT_cfrac_D 0#1\Z #2#3#4#5{ #2}%
\def\XINT_cfrac_D #1\Z #2#3{\XINT_cfrac_loop_a {#1}{#3}{#1}{{#2}}}%
\def\XINT_cfrac_loop_a
{%
    \expandafter\XINT_cfrac_loop_d\romannumeral0\XINT_div_prepare
}%
\def\XINT_cfrac_loop_d #1#2%
{%
    \XINT_cfrac_loop_e #2.{#1}%
}%
\def\XINT_cfrac_loop_e #1%
{%
    \xint_gob_til_zero #1\xint_cfrac_loop_exit0\XINT_cfrac_loop_f #1%
}%
\def\XINT_cfrac_loop_f #1.#2#3#4%
{%
    \XINT_cfrac_loop_a {#1}{#3}{#1}{{#2}#4}%
}%
\def\xint_cfrac_loop_exit0\XINT_cfrac_loop_f #1.#2#3#4#5#6%
```

```
    {\XINT_cfrac_T #5#6{#2}#4\Z }%
\def\XINT_cfrac_T #1#2#3#4%
{%
  \xint_gob_til_z #4\XINT_cfrac_end\Z\XINT_cfrac_T #1#2{#4+\cfrac{#11#2}{#3}}%
}%
\def\XINT_cfrac_end\Z\XINT_cfrac_T #1#2#3%
{%
    \XINT_cfrac__end #3%
}%
\def\XINT_cfrac__end \Z+\cfrac#1#2{ #2}%
```

23.6 \xintGCFrac

```
\def\xintGCFrac {\romannumeral0\xintgcfrac }%
\def\xintgcfrac #1%
{%
    \XINT_gcfrac_opt_a #1\Z
}%
\def\XINT_gcfrac_opt_a #1%
{%
    \ifx#1[\XINT_gcfrac_opt_b\fi \XINT_gcfrac_noopt #1%
}%
\def\XINT_gcfrac_noopt #1\Z
{%
    \XINT_gcfrac #1+\W/\relax\relax
}%
\def\XINT_gcfrac_opt_b\fi\XINT_gcfrac_noopt [\Z #1]%
{%
    \fi\csname XINT_gcfrac_opt#1\endcsname
}%
\def\XINT_gcfrac_optl #1%
{%
    \XINT_gcfrac #1+\W/\relax\hfill
}%
\def\XINT_gcfrac_optc #1%
{%
    \XINT_gcfrac #1+\W/\relax\relax
}%
\def\XINT_gcfrac_optr #1%
{%
    \XINT_gcfrac #1+\W/\hfill\relax
}%
\def\XINT_gcfrac
{%
    \expandafter\XINT_gcfrac_enter\romannumeral-`0%
}%
\def\XINT_gcfrac_enter {\XINT_gcfrac_loop {}}%
\def\XINT_gcfrac_loop #1#2+#3/%
{%
    \xint_gob_til_w #3\XINT_gcfrac_endloop\W
```

```
    \XINT_gcfrac_loop {{#3}{#2}#1}%
}%
\def\XINT_gcfrac_endloop\W\XINT_gcfrac_loop #1#2#3%
{%
    \XINT_gcfrac_T #2#3#1\Z\Z
}%
\def\XINT_gcfrac_T #1#2#3#4{\XINT_gcfrac_U #1#2{\xintFrac{#4}}}%
\def\XINT_gcfrac_U #1#2#3#4#5%
{%
    \xint_gob_til_z #5\XINT_gcfrac_end\Z\XINT_gcfrac_U
               #1#2{\xintFrac{#5}%
                \ifcase\xintSgn{#4}
                +\or+\else-\fi
                \cfrac{#1\xintFrac{\xintAbs{#4}}#2}{#3}}%
}%
\def\XINT_gcfrac_end\Z\XINT_gcfrac_U #1#2#3%
{%
    \XINT_gcfrac__end #3%
}%
\def\XINT_gcfrac__end #1\cfrac#2#3{ #3}%
```

23.7 \xintGCtoGCx

```
\def\xintGCtoGCx {\romannumeral0\xintgctogcx }%
\def\xintgctogcx #1#2#3%
{%
    \expandafter\XINT_gctgcx_start\expandafter {\romannumeral-`0#3}{#1}{#2}%
}%
\def\XINT_gctgcx_start #1#2#3{\XINT_gctgcx_loop_a {}{#2}{#3}#1+\W/}%
\def\XINT_gctgcx_loop_a #1#2#3#4+#5/%
{%
    \xint_gob_til_w #5\XINT_gctgcx_end\W
    \XINT_gctgcx_loop_b {#1{#4}}{#2{#5}#3}{#2}{#3}%
}%
\def\XINT_gctgcx_loop_b #1#2%
{%
    \XINT_gctgcx_loop_a {#1#2}%
}%
\def\XINT_gctgcx_end\W\XINT_gctgcx_loop_b #1#2#3#4{ #1}%
```

23.8 \xintFtoCs

```
\def\xintFtoCs {\romannumeral0\xintftocs }%
\def\xintftocs #1%
{%
    \expandafter\XINT_ftc_A\romannumeral0\xintraw {#1}\Z
}%
\def\XINT_ftc_A #1/#2\Z
{%
    \expandafter\XINT_ftc_B\romannumeral0\xintidivision {#1}{#2}{#2}%
}%
```

```
\def\XINT_ftc_B #1#2%
{%
    \XINT_ftc_C #2.{#1}%
}%
\def\XINT_ftc_C #1%
{%
    \xint_gob_til_zero #1\XINT_ftc_integer 0\XINT_ftc_D #1%
}%
\def\XINT_ftc_integer 0\XINT_ftc_D 0#1.#2#3{ #2}%
\def\XINT_ftc_D #1.#2#3{\XINT_ftc_loop_a {#1}{#3}{#1}{#2,}}%
\def\XINT_ftc_loop_a
{%
    \expandafter\XINT_ftc_loop_d\romannumeral0\XINT_div_prepare
}%
\def\XINT_ftc_loop_d #1#2%
{%
    \XINT_ftc_loop_e #2.{#1}%
}%
\def\XINT_ftc_loop_e #1%
{%
    \xint_gob_til_zero #1\xint_ftc_loop_exit0\XINT_ftc_loop_f #1%
}%
\def\XINT_ftc_loop_f #1.#2#3#4%
{%
    \XINT_ftc_loop_a {#1}{#3}{#1}{#4#2,}%
}%
\def\xint_ftc_loop_exit0\XINT_ftc_loop_f #1.#2#3#4{ #4#2}%
```

23.9 \xintFtoCx

```
\def\xintFtoCx {\romannumeral0\xintftocx }%
\def\xintftocx #1#2%
{%
    \expandafter\XINT_ftcx_A\romannumeral0\xintraw {#2}\Z {#1}%
}%
\def\XINT_ftcx_A #1/#2\Z
{%
    \expandafter\XINT_ftcx_B\romannumeral0\xintidivision {#1}{#2}{#2}%
}%
\def\XINT_ftcx_B #1#2%
{%
    \XINT_ftcx_C #2.{#1}%
}%
\def\XINT_ftcx_C #1%
{%
    \xint_gob_til_zero #1\XINT_ftcx_integer 0\XINT_ftcx_D #1%
}%
\def\XINT_ftcx_integer 0\XINT_ftcx_D 0#1.#2#3#4{ #2}%
\def\XINT_ftcx_D #1.#2#3#4{\XINT_ftcx_loop_a {#1}{#3}{#1}{#2#4}{#4}}%
\def\XINT_ftcx_loop_a
```

```
{%
    \expandafter\XINT_ftcx_loop_d\romannumeral0\XINT_div_prepare
}%
\def\XINT_ftcx_loop_d #1#2%
{%
    \XINT_ftcx_loop_e #2.{#1}%
}%
\def\XINT_ftcx_loop_e #1%
{%
    \xint_gob_til_zero #1\xint_ftcx_loop_exit0\XINT_ftcx_loop_f #1%
}%
\def\XINT_ftcx_loop_f #1.#2#3#4#5%
{%
    \XINT_ftcx_loop_a {#1}{#3}{#1}{#4{#2}#5}{#5}%
}%
\def\xint_ftcx_loop_exit0\XINT_ftcx_loop_f #1.#2#3#4#5{ #4{#2}}%
```

23.10 \xintFtoGC

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

23.11 \xintFtoCC

```
\def\xintFtoCC {\romannumeral0\xintftocc }%
\def\xintftocc #1%
{%
    \expandafter\XINT_ftcc_A\expandafter {\romannumeral0\xintraw {#1}}%
}%
\def\XINT_ftcc_A #1%
{%
    \expandafter\XINT_ftcc_B
    \romannumeral0\xintraw {\xintAdd {1/2[0]}{#1[0]}}\Z {#1[0]}%
}%
\def\XINT_ftcc_B #1/#2\Z
{%
    \expandafter\XINT_ftcc_C\expandafter {\romannumeral0\xintiquo {#1}{#2}}%
}%
\def\XINT_ftcc_C #1#2%
{%
    \expandafter\XINT_ftcc_D\romannumeral0\xintsub {#2}{#1}\Z {#1}%
}%
\def\XINT_ftcc_D #1%
{%
    \xint_UDzerominusfork
      #1-\dummy \XINT_ftcc_integer
      0#1\dummy \XINT_ftcc_En
       0-\dummy {\XINT_ftcc_Ep #1}%
    \krof
}%
\def\XINT_ftcc_Ep #1\Z #2%
```

```
{%
    \expandafter\XINT_ftcc_loop_a\expandafter
    {\romannumeral0\xintdiv {1[0]}{#1}}{#2+1/}%
}%
\def\XINT_ftcc_En #1\Z #2%
{%
    \expandafter\XINT_ftcc_loop_a\expandafter
    {\romannumeral0\xintdiv {1[0]}{#1}}{#2+-1/}%
}%
\def\XINT_ftcc_integer #1\Z #2{ #2}%
\def\XINT_ftcc_loop_a #1%
{%
    \expandafter\XINT_ftcc_loop_b
    \romannumeral0\xintraw {\xintAdd {1/2[0]}{#1}}\Z {#1}%
}%
\def\XINT_ftcc_loop_b #1/#2\Z
{%
    \expandafter\XINT_ftcc_loop_c\expandafter
    {\romannumeral0\xintiquo {#1}{#2}}%
}%
\def\XINT_ftcc_loop_c #1#2%
{%
    \expandafter\XINT_ftcc_loop_d
    \romannumeral0\xintsub {#2}{#1[0]}\Z {#1}%
}%
\def\XINT_ftcc_loop_d #1%
{%
    \xint_UDzerominusfork
      #1-\dummy \XINT_ftcc_end
      0#1\dummy \XINT_ftcc_loop_N
       0-\dummy {\XINT_ftcc_loop_P #1}%
    \krof
}%
\def\XINT_ftcc_end #1\Z #2#3{ #3#2}%
\def\XINT_ftcc_loop_P #1\Z #2#3%
{%
    \expandafter\XINT_ftcc_loop_a\expandafter
    {\romannumeral0\xintdiv {1[0]}{#1}}{#3#2+1/}%
}%
\def\XINT_ftcc_loop_N #1\Z #2#3%
{%
    \expandafter\XINT_ftcc_loop_a\expandafter
    {\romannumeral0\xintdiv {1[0]}{#1}}{#3#2+-1/}%
}%
```

23.12 \xintFtoCv

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

```
    \xinticstocv {\xintFtoCs {#1}}%
}%
```

23.13 \xintFtoCCv

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

23.14 \xintCstoF

```
\def\xintCstoF {\romannumeral0\xintcstof }%
\def\xintcstof #1%
{%
    \expandafter\XINT_cstf_prep \romannumeral-`0#1,\W,%
}%
\def\XINT_cstf_prep
{%
    \XINT_cstf_loop_a 1001%
}%
\def\XINT_cstf_loop_a #1#2#3#4#5,%
{%
    \xint_gob_til_w #5\XINT_cstf_end\W
    \expandafter\XINT_cstf_loop_b
    \romannumeral0\xintraw {#5}.{#1}{#2}{#3}{#4}%
}%
\def\XINT_cstf_loop_b #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_cstf_loop_c\expandafter
    {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
    {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
    {\romannumeral0\xintiadd {\XINT_Mul {#2}{#6}}{\XINT_Mul {#1}{#4}}}%
    {\romannumeral0\xintiadd {\XINT_Mul {#2}{#5}}{\XINT_Mul {#1}{#3}}}%
}%
\def\XINT_cstf_loop_c #1#2%
{%
    \expandafter\XINT_cstf_loop_d\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_cstf_loop_d #1#2%
{%
    \expandafter\XINT_cstf_loop_e\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_cstf_loop_e #1#2%
{%
    \expandafter\XINT_cstf_loop_a\expandafter{#2}#1%
}%
\def\XINT_cstf_end #1.#2#3#4#5{\xintraw {#2/#3}[0]}%
```

23.15 \xintiCstoF

```
\def\xintiCstoF {\romannumeral0\xinticstof }%
\def\xinticstof #1%
{%
    \expandafter\XINT_icstf_prep \romannumeral-`0#1,\W,%
}%
\def\XINT_icstf_prep
{%
    \XINT_icstf_loop_a 1001%
}%
\def\XINT_icstf_loop_a #1#2#3#4#5,%
{%
    \xint_gob_til_w #5\XINT_icstf_end\W
    \expandafter
    \XINT_icstf_loop_b \romannumeral-`0#5.{#1}{#2}{#3}{#4}%
}%
\def\XINT_icstf_loop_b #1.#2#3#4#5%
{%
    \expandafter\XINT_icstf_loop_c\expandafter
    {\romannumeral0\xintiadd {#5}{\XINT_Mul {#1}{#3}}}%
    {\romannumeral0\xintiadd {#4}{\XINT_Mul {#1}{#2}}}%
    {#2}{#3}%
}%
\def\XINT_icstf_loop_c #1#2%
{%
    \expandafter\XINT_icstf_loop_a\expandafter {#2}{#1}%
}%
\def\XINT_icstf_end#1.#2#3#4#5{\xintraw {#2/#3}[0]}%
```

23.16 \xintGCtoF

```
\def\xintGCtoF {\romannumeral0\xintgctof }%
\def\xintgctof #1%
{%
    \expandafter\XINT_gctf_prep \romannumeral-`0#1+\W/%
}%
\def\XINT_gctf_prep
{%
    \XINT_gctf_loop_a 1001%
}%
\def\XINT_gctf_loop_a #1#2#3#4#5+%
{%
    \expandafter\XINT_gctf_loop_b
    \romannumeral0\xintraw {#5}.{#1}{#2}{#3}{#4}%
}%
\def\XINT_gctf_loop_b #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_gctf_loop_c\expandafter
    {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
```

```
5807     {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
5808     {\romannumeral0\xintiadd {\XINT_Mul {#2}{#6}}{\XINT_Mul {#1}{#4}}}%
5809     {\romannumeral0\xintiadd {\XINT_Mul {#2}{#5}}{\XINT_Mul {#1}{#3}}}%
5810 }%
5811 \def\XINT_gctf_loop_c #1#2%
5812 {%
5813     \expandafter\XINT_gctf_loop_d\expandafter {\expandafter{#2}{#1}}%
5814 }%
5815 \def\XINT_gctf_loop_d #1#2%
5816 {%
5817     \expandafter\XINT_gctf_loop_e\expandafter {\expandafter{#2}#1}%
5818 }%
5819 \def\XINT_gctf_loop_e #1#2%
5820 {%
5821     \expandafter\XINT_gctf_loop_f\expandafter {\expandafter{#2}#1}%
5822 }%
5823 \def\XINT_gctf_loop_f #1#2/%
5824 {%
5825     \xint_gob_til_w #2\XINT_gctf_end\W
5826     \expandafter\XINT_gctf_loop_g
5827     \romannumeral0\xintraw {#2}.#1%
5828 }%
5829 \def\XINT_gctf_loop_g #1/#2.#3#4#5#6%
5830 {%
5831     \expandafter\XINT_gctf_loop_h\expandafter
5832     {\romannumeral0\XINT_mul_fork #1\Z #6\Z }%
5833     {\romannumeral0\XINT_mul_fork #1\Z #5\Z }%
5834     {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
5835     {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
5836 }%
5837 \def\XINT_gctf_loop_h #1#2%
5838 {%
5839     \expandafter\XINT_gctf_loop_i\expandafter {\expandafter{#2}{#1}}%
5840 }%
5841 \def\XINT_gctf_loop_i #1#2%
5842 {%
5843     \expandafter\XINT_gctf_loop_j\expandafter {\expandafter{#2}#1}%
5844 }%
5845 \def\XINT_gctf_loop_j #1#2%
5846 {%
5847     \expandafter\XINT_gctf_loop_a\expandafter {#2}#1%
5848 }%
5849 \def\XINT_gctf_end #1.#2#3#4#5{\xintraw {#2/#3}[0]}%
```

23.17 \xintiGCtoF

```
5850 \def\xintiGCtoF {\romannumeral0\xintigctof }%
5851 \def\xintigctof #1%
5852 {%
5853     \expandafter\XINT_igctf_prep \romannumeral-‘0#1+\W/%
```

```
}%
\def\XINT_igctf_prep
{%
    \XINT_igctf_loop_a 1001%
}%
\def\XINT_igctf_loop_a #1#2#3#4#5+%
{%
    \expandafter\XINT_igctf_loop_b
    \romannumeral-`0#5.{#1}{#2}{#3}{#4}%
}%
\def\XINT_igctf_loop_b #1.#2#3#4#5%
{%
    \expandafter\XINT_igctf_loop_c\expandafter
    {\romannumeral0\xintiadd {#5}{\XINT_Mul {#1}{#3}}}%
    {\romannumeral0\xintiadd {#4}{\XINT_Mul {#1}{#2}}}%
    {#2}{#3}%
}%
\def\XINT_igctf_loop_c #1#2%
{%
    \expandafter\XINT_igctf_loop_f\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_igctf_loop_f #1#2#3#4/%
{%
    \xint_gob_til_w #4\XINT_igctf_end\W
    \expandafter\XINT_igctf_loop_g
    \romannumeral-`0#4.{#2}{#3}#1%
}%
\def\XINT_igctf_loop_g #1.#2#3%
{%
    \expandafter\XINT_igctf_loop_h\expandafter
    {\romannumeral0\XINT_mul_fork #1\Z #3\Z }%
    {\romannumeral0\XINT_mul_fork #1\Z #2\Z }%
}%
\def\XINT_igctf_loop_h #1#2%
{%
    \expandafter\XINT_igctf_loop_i\expandafter {#2}{#1}%
}%
\def\XINT_igctf_loop_i #1#2#3#4%
{%
    \XINT_igctf_loop_a {#3}{#4}{#1}{#2}%
}%
\def\XINT_igctf_end #1.#2#3#4#5{\xintraw {#4/#5}[0]}%
```

23.18 \xintCstoCv

```
\def\xintCstoCv {\romannumeral0\xintcstocv }%
\def\xintcstocv #1%
{%
    \expandafter\XINT_cstcv_prep \romannumeral-`0#1,\W,%
}%
```

```
\def\XINT_cstcv_prep
{%
    \XINT_cstcv_loop_a {}1001%
}%
\def\XINT_cstcv_loop_a #1#2#3#4#5#6,%
{%
    \xint_gob_til_w #6\XINT_cstcv_end\W
    \expandafter\XINT_cstcv_loop_b
    \romannumeral0\xintraw {#6}.{#2}{#3}{#4}{#5}{#1}%
}%
\def\XINT_cstcv_loop_b #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_cstcv_loop_c\expandafter
    {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
    {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
    {\romannumeral0\xintiadd {\XINT_Mul {#2}{#6}}{\XINT_Mul {#1}{#4}}}%
    {\romannumeral0\xintiadd {\XINT_Mul {#2}{#5}}{\XINT_Mul {#1}{#3}}}%
}%
\def\XINT_cstcv_loop_c #1#2%
{%
    \expandafter\XINT_cstcv_loop_d\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_cstcv_loop_d #1#2%
{%
    \expandafter\XINT_cstcv_loop_e\expandafter {\expandafter{#2}#1}%
}%
\def\XINT_cstcv_loop_e #1#2%
{%
    \expandafter\XINT_cstcv_loop_f\expandafter{#2}#1%
}%
\def\XINT_cstcv_loop_f #1#2#3#4#5%
{%
    \expandafter\XINT_cstcv_loop_g\expandafter
    {\romannumeral0\xintraw {#1/#2}}{#5}{#1}{#2}{#3}{#4}%
}%
\def\XINT_cstcv_loop_g #1#2{\XINT_cstcv_loop_a {#2{#1[0]}}}%
\def\XINT_cstcv_end #1.#2#3#4#5#6{ #6}%
```

23.19 \xintiCstoCv

```
\def\xintiCstoCv {\romannumeral0\xinticstocv }%
\def\xinticstocv #1%
{%
    \expandafter\XINT_icstcv_prep \romannumeral-`0#1,\W,%
}%
\def\XINT_icstcv_prep
{%
    \XINT_icstcv_loop_a {}1001%
}%
\def\XINT_icstcv_loop_a #1#2#3#4#5#6,%
```

```
{%
    \xint_gob_til_w #6\XINT_icstcv_end\W
    \expandafter
    \XINT_icstcv_loop_b \romannumeral-`0#6.{#2}{#3}{#4}{#5}{#1}%
}%
\def\XINT_icstcv_loop_b #1.#2#3#4#5%
{%
    \expandafter\XINT_icstcv_loop_c\expandafter
    {\romannumeral0\xintiadd {#5}{\XINT_Mul {#1}{#3}}}%
    {\romannumeral0\xintiadd {#4}{\XINT_Mul {#1}{#2}}}%
    {{#2}{#3}}%
}%
\def\XINT_icstcv_loop_c #1#2%
{%
    \expandafter\XINT_icstcv_loop_d\expandafter {#2}{#1}%
}%
\def\XINT_icstcv_loop_d #1#2%
{%
    \expandafter\XINT_icstcv_loop_e\expandafter
    {\romannumeral0\xintraw {#1/#2}}{{#1}{#2}}%
}%
\def\XINT_icstcv_loop_e #1#2#3#4{\XINT_icstcv_loop_a {#4{#1[0]}}#2#3}%
\def\XINT_icstcv_end #1.#2#3#4#5#6{ #6}%
```

23.20 \xintGCtoCv

```
\def\xintGCtoCv {\romannumeral0\xintgctocv }%
\def\xintgctocv #1%
{%
    \expandafter\XINT_gctcv_prep \romannumeral-`0#1+\W/%
}%
\def\XINT_gctcv_prep
{%
    \XINT_gctcv_loop_a {}1001%
}%
\def\XINT_gctcv_loop_a #1#2#3#4#5#6+%
{%
    \expandafter\XINT_gctcv_loop_b
    \romannumeral0\xintraw {#6}.{#2}{#3}{#4}{#5}{#1}%
}%
\def\XINT_gctcv_loop_b #1/#2.#3#4#5#6%
{%
    \expandafter\XINT_gctcv_loop_c\expandafter
    {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
    {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
    {\romannumeral0\xintiadd {\XINT_Mul {#2}{#6}}{\XINT_Mul {#1}{#4}}}%
    {\romannumeral0\xintiadd {\XINT_Mul {#2}{#5}}{\XINT_Mul {#1}{#3}}}%
}%
\def\XINT_gctcv_loop_c #1#2%
{%
```

```
5995     \expandafter\XINT_gctcv_loop_d\expandafter {\expandafter{#2}{#1}}%
5996 }%
5997 \def\XINT_gctcv_loop_d #1#2%
5998 {%
5999     \expandafter\XINT_gctcv_loop_e\expandafter {\expandafter{#2}{#1}}%
6000 }%
6001 \def\XINT_gctcv_loop_e #1#2%
6002 {%
6003     \expandafter\XINT_gctcv_loop_f\expandafter {#2}#1%
6004 }%
6005 \def\XINT_gctcv_loop_f #1#2%
6006 {%
6007     \expandafter\XINT_gctcv_loop_g\expandafter
6008     {\romannumeral0\xintraw {#1/#2}}{{#1}{#2}}%
6009 }%
6010 \def\XINT_gctcv_loop_g #1#2#3#4%
6011 {%
6012     \XINT_gctcv_loop_h {#4{#1[0]}}{#2#3}%
6013 }%
6014 \def\XINT_gctcv_loop_h #1#2#3/%
6015 {%
6016     \xint_gob_til_w #3\XINT_gctcv_end\W
6017     \expandafter\XINT_gctcv_loop_i
6018     \romannumeral0\xintraw {#3}.#2{#1}%
6019 }%
6020 \def\XINT_gctcv_loop_i #1/#2.#3#4#5#6%
6021 {%
6022     \expandafter\XINT_gctcv_loop_j\expandafter
6023     {\romannumeral0\XINT_mul_fork #1\Z #6\Z }%
6024     {\romannumeral0\XINT_mul_fork #1\Z #5\Z }%
6025     {\romannumeral0\XINT_mul_fork #2\Z #4\Z }%
6026     {\romannumeral0\XINT_mul_fork #2\Z #3\Z }%
6027 }%
6028 \def\XINT_gctcv_loop_j #1#2%
6029 {%
6030     \expandafter\XINT_gctcv_loop_k\expandafter {\expandafter{#2}{#1}}%
6031 }%
6032 \def\XINT_gctcv_loop_k #1#2%
6033 {%
6034     \expandafter\XINT_gctcv_loop_l\expandafter {\expandafter{#2}#1}%
6035 }%
6036 \def\XINT_gctcv_loop_l #1#2%
6037 {%
6038     \expandafter\XINT_gctcv_loop_m\expandafter {\expandafter{#2}#1}%
6039 }%
6040 \def\XINT_gctcv_loop_m #1#2{\XINT_gctcv_loop_a {#2}#1}%
6041 \def\XINT_gctcv_end #1.#2#3#4#5#6{ #6}%
```

23.21 \xintiGCtoCv

```
\def\xintiGCtoCv {\romannumeral0\xintigctocv }%
\def\xintigctocv #1%
{%
    \expandafter\XINT_igctcv_prep \romannumeral-`0#1+\W/%
}%
\def\XINT_igctcv_prep
{%
    \XINT_igctcv_loop_a {}1001%
}%
\def\XINT_igctcv_loop_a #1#2#3#4#5#6+%
{%
    \expandafter\XINT_igctcv_loop_b
    \romannumeral-`0#6.{#2}{#3}{#4}{#5}{#1}%
}%
\def\XINT_igctcv_loop_b #1.#2#3#4#5%
{%
    \expandafter\XINT_igctcv_loop_c\expandafter
    {\romannumeral0\xintiadd {#5}{\XINT_Mul {#1}{#3}}}%
    {\romannumeral0\xintiadd {#4}{\XINT_Mul {#1}{#2}}}%
    {{#2}{#3}}%
}%
\def\XINT_igctcv_loop_c #1#2%
{%
    \expandafter\XINT_igctcv_loop_f\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_igctcv_loop_f #1#2#3#4/%
{%
    \xint_gob_til_w #4\XINT_igctcv_end_a\W
    \expandafter\XINT_igctcv_loop_g
    \romannumeral-`0#4.#1#2{#3}%
}%
\def\XINT_igctcv_loop_g #1.#2#3#4#5%
{%
    \expandafter\XINT_igctcv_loop_h\expandafter
    {\romannumeral0\XINT_mul_fork #1\Z #5\Z }%
    {\romannumeral0\XINT_mul_fork #1\Z #4\Z }%
    {{#2}{#3}}%
}%
\def\XINT_igctcv_loop_h #1#2%
{%
    \expandafter\XINT_igctcv_loop_i\expandafter {\expandafter{#2}{#1}}%
}%
\def\XINT_igctcv_loop_i #1#2{\XINT_igctcv_loop_k #2{#2#1}}%
\def\XINT_igctcv_loop_k #1#2%
{%
    \expandafter\XINT_igctcv_loop_l\expandafter
    {\romannumeral0\xintraw {#1/#2}}%
}%
\def\XINT_igctcv_loop_l #1#2#3{\XINT_igctcv_loop_a {#3{#1[0]}}#2}%
```

```
\def\XINT_igctcv_end_a #1.#2#3#4#5%
{%
    \expandafter\XINT_igctcv_end_b\expandafter
    {\romannumeral0\xintraw {#2/#3}}%
}%
\def\XINT_igctcv_end_b #1#2{ #2{#1[0]}}%
```

23.22 \xintCntoF

Modified in 1.06 to give the N first to a \numexpr rather than expanding twice.
I just use \the\numexpr and maintain the previous code after that.

```
\def\xintCntoF {\romannumeral0\xintcntof }%
\def\xintcntof #1%
{%
    \expandafter\XINT_cntf\expandafter {\the\numexpr #1}%
}%
\def\XINT_cntf #1#2%
{%
   \ifnum #1>0
      \xint_afterfi {\expandafter\XINT_cntf_loop\expandafter
                     {\the\numexpr #1-1\expandafter}\expandafter
                     {\romannumeral-`0#2{#1}}{#2}}%
   \else
      \xint_afterfi
         {\ifnum #1=0
              \xint_afterfi {\expandafter\space \romannumeral-`0#2{0}}%
          \else \xint_afterfi { 0/1[0]}%
          \fi}%
   \fi
}%
\def\XINT_cntf_loop #1#2#3%
{%
    \ifnum #1>0 \else \XINT_cntf_exit \fi
    \expandafter\XINT_cntf_loop\expandafter
    {\the\numexpr #1-1\expandafter }\expandafter
    {\romannumeral0\xintadd {\xintDiv {1[0]}{#2}}{#3{#1}}}%
    {#3}%
}%
\def\XINT_cntf_exit \fi
    \expandafter\XINT_cntf_loop\expandafter
    #1\expandafter #2#3%
{%
    \fi\xint_gobble_ii #2%
}%
```

23.23 \xintGCntoF

Modified in 1.06 to give the N first to a \numexpr rather than expanding twice.
I just use \the\numexpr and maintain the previous code after that.

```
\def\xintGCntoF {\romannumeral0\xintgcntof }%
\def\xintgcntof #1%
{%
    \expandafter\XINT_gcntf\expandafter {\the\numexpr #1}%
}%
\def\XINT_gcntf #1#2#3%
{%
   \ifnum #1>0
      \xint_afterfi {\expandafter\XINT_gcntf_loop\expandafter
                     {\the\numexpr #1-1\expandafter}\expandafter
                     {\romannumeral-`0#2{#1}}{#2}{#3}}%
   \else
      \xint_afterfi
         {\ifnum #1=0
              \xint_afterfi {\expandafter\space\romannumeral-`0#2{0}}%
          \else \xint_afterfi { 0/1[0]}%
          \fi}%
   \fi
}%
\def\XINT_gcntf_loop #1#2#3#4%
{%
    \ifnum #1>0 \else \XINT_gcntf_exit \fi
    \expandafter\XINT_gcntf_loop\expandafter
    {\the\numexpr #1-1\expandafter }\expandafter
    {\romannumeral0\xintadd {\xintDiv {#4{#1}}{#2}}{#3{#1}}}%
    {#3}{#4}%
}%
\def\XINT_gcntf_exit \fi
    \expandafter\XINT_gcntf_loop\expandafter
    #1\expandafter #2#3#4%
{%
    \fi\xint_gobble_ii #2%
}%
```

23.24 \xintCntoCs

Modified in 1.06 to give the N first to a \numexpr rather than expanding twice. I just use \the\numexpr and maintain the previous code after that.

```
\def\xintCntoCs {\romannumeral0\xintcntocs }%
\def\xintcntocs #1%
{%
    \expandafter\XINT_cntcs\expandafter {\the\numexpr #1}%
}%
\def\XINT_cntcs #1#2%
{%
   \ifnum #1<0
      \xint_afterfi { 0/1[0]}%
   \else
```

```
      \xint_afterfi {\expandafter\XINT_cntcs_loop\expandafter
                     {\the\numexpr #1-1\expandafter}\expandafter
                     {\expandafter{\romannumeral-`0#2{#1}}}{#2}}%
   \fi
}%
\def\XINT_cntcs_loop #1#2#3%
{%
    \ifnum #1>-1 \else \XINT_cntcs_exit \fi
    \expandafter\XINT_cntcs_loop\expandafter
    {\the\numexpr #1-1\expandafter }\expandafter
    {\expandafter{\romannumeral-`0#3{#1}},#2}{#3}%
}%
\def\XINT_cntcs_exit \fi
    \expandafter\XINT_cntcs_loop\expandafter
    #1\expandafter #2#3%
{%
    \fi\XINT_cntcs__exit #2%
}%
\def\XINT_cntcs__exit #1,{ }%
```

23.25 \xintCntoGC

```
Modified in 1.06 to give the N first to a \numexpr rather than expanding twice.
I just use \the\numexpr and maintain the previous code after that.
```

```
\def\xintCntoGC {\romannumeral0\xintcntogc }%
\def\xintcntogc #1%
{%
    \expandafter\XINT_cntgc\expandafter {\the\numexpr #1}%
}%
\def\XINT_cntgc #1#2%
{%
   \ifnum #1<0
      \xint_afterfi { 0/1[0]}%
   \else
      \xint_afterfi {\expandafter\XINT_cntgc_loop\expandafter
                     {\the\numexpr #1-1\expandafter}\expandafter
                     {\expandafter{\romannumeral-`0#2{#1}}}{#2}}%
   \fi
}%
\def\XINT_cntgc_loop #1#2#3%
{%
    \ifnum #1>-1 \else \XINT_cntgc_exit \fi
    \expandafter\XINT_cntgc_loop\expandafter
    {\the\numexpr #1-1\expandafter }\expandafter
    {\expandafter{\romannumeral-`0#3{#1}}+1/#2}{#3}%
}%
\def\XINT_cntgc_exit \fi
    \expandafter\XINT_cntgc_loop\expandafter
```

```
    #1\expandafter #2#3%
{%
    \fi\XINT_cntgc__exit #2%
}%
\def\XINT_cntgc__exit #1+1/{ }%
```

23.26 \xintGCntoGC

```
Modified in 1.06 to give the N first to a \numexpr rather than expanding twice.
I just use \the\numexpr and maintain the previous code after that.
```

```
\def\xintGCntoGC {\romannumeral0\xintgcntogc }%
\def\xintgcntogc #1%
{%
    \expandafter\XINT_gcntgc\expandafter {\the\numexpr #1}%
}%
\def\XINT_gcntgc #1#2#3%
{%
   \ifnum #1<0
      \xint_afterfi { {0/1[0]}}%
   \else
      \xint_afterfi {\expandafter\XINT_gcntgc_loop\expandafter
                     {\the\numexpr #1-1\expandafter}\expandafter
                     {\expandafter{\romannumeral-`0#2{#1}}}{#2}{#3}}%
   \fi
}%
\def\XINT_gcntgc_loop #1#2#3#4%
{%
    \ifnum #1>-1 \else \XINT_gcntgc_exit \fi
    \expandafter\XINT_gcntgc_loop_b\expandafter
    {\expandafter{\romannumeral-`0#4{#1}}/#2}{#3{#1}}{#1}{#3}{#4}%
}%
\def\XINT_gcntgc_loop_b #1#2#3%
{%
    \expandafter\XINT_gcntgc_loop\expandafter
    {\the\numexpr #3-1\expandafter}\expandafter
    {\expandafter{\romannumeral-`0#2}+#1}%
}%
\def\XINT_gcntgc_exit \fi
    \expandafter\XINT_gcntgc_loop_b\expandafter #1#2#3#4#5%
{%
    \fi\XINT_gcntgc__exit #1%
}%
\def\XINT_gcntgc__exit #1/{ }%
```

23.27 \xintCstoGC

```
\def\xintCstoGC {\romannumeral0\xintcstogc }%
\def\xintcstogc #1%
```

```
{%
    \expandafter\XINT_cstc_prep \romannumeral-`0#1,\W,%
}%
\def\XINT_cstc_prep #1,{\XINT_cstc_loop_a {{#1}}}%
\def\XINT_cstc_loop_a #1#2,%
{%
    \xint_gob_til_w #2\XINT_cstc_end\W
    \XINT_cstc_loop_b {#1}{#2}%
}%
\def\XINT_cstc_loop_b #1#2{\XINT_cstc_loop_a {#1+1/{#2}}}%
\def\XINT_cstc_end\W\XINT_cstc_loop_b #1#2{ #1}%
```

23.28 \xintGCtoGC

```
\def\xintGCtoGC {\romannumeral0\xintgctogc }%
\def\xintgctogc #1%
{%
    \expandafter\XINT_gctgc_start \romannumeral-`0#1+\W/%
}%
\def\XINT_gctgc_start {\XINT_gctgc_loop_a {}}%
\def\XINT_gctgc_loop_a #1#2+#3/%
{%
    \xint_gob_til_w #3\XINT_gctgc_end\W
    \expandafter\XINT_gctgc_loop_b\expandafter
    {\romannumeral-`0#2}{#3}{#1}%
}%
\def\XINT_gctgc_loop_b #1#2%
{%
    \expandafter\XINT_gctgc_loop_c\expandafter
    {\romannumeral-`0#2}{#1}%
}%
\def\XINT_gctgc_loop_c #1#2#3%
{%
    \XINT_gctgc_loop_a {#3{#2}+{#1}/}%
}%
\def\XINT_gctgc_end\W\expandafter\XINT_gctgc_loop_b
{%
    \expandafter\XINT_gctgc__end
}%
\def\XINT_gctgc__end #1#2#3{ #3{#1}}%
\XINT_cfrac_restorecatcodes_endinput%
```