

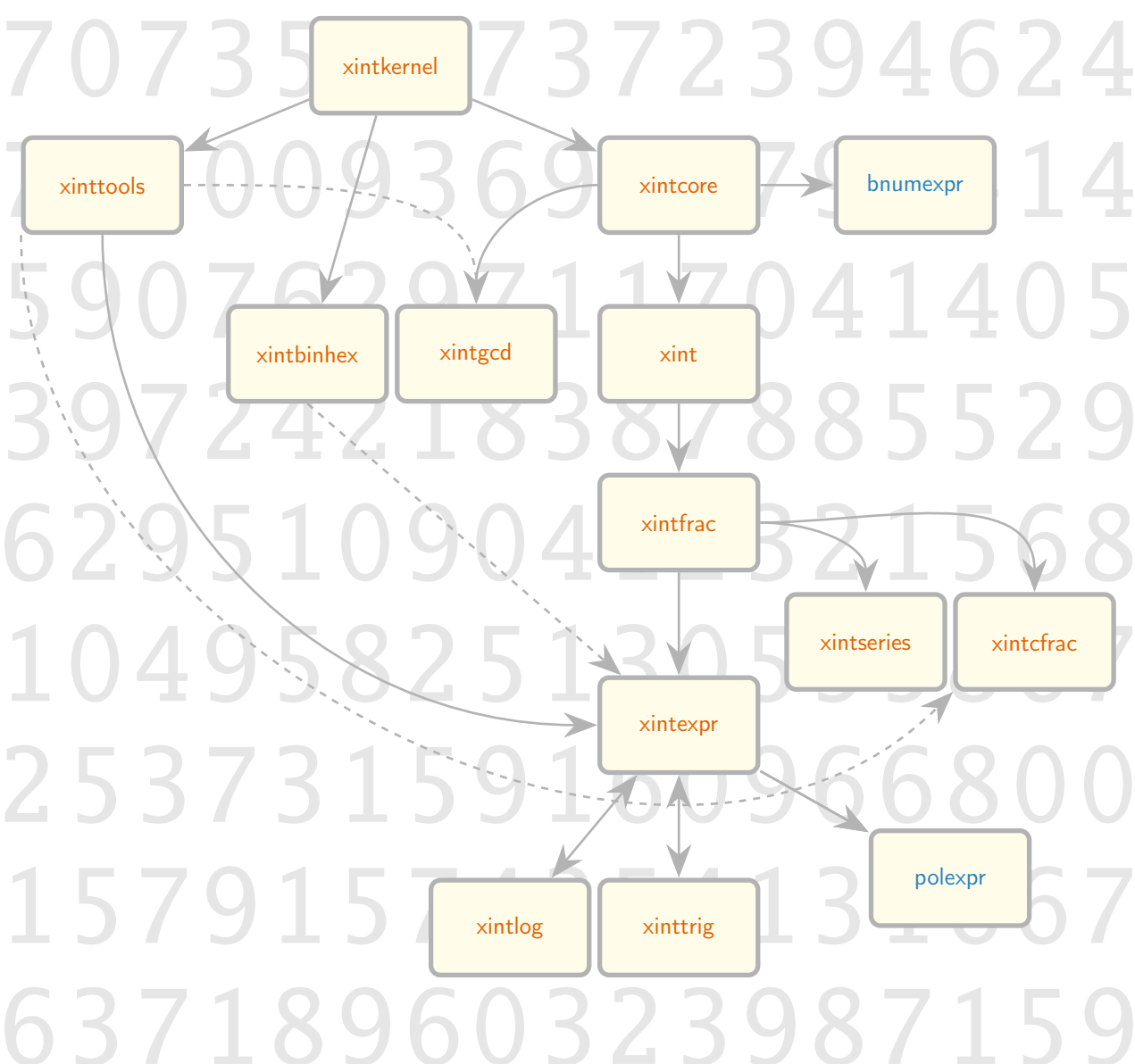
The `xint` bundle

JEAN-FRANÇOIS BURNOL
jfbu (at) free (dot) fr

Package version: 1.3e (2019/04/05); documentation date: 2019/04/05.
From source file `xint.dtx`. Time-stamp: <06-04-2019 at 18:55:58 CEST>.

Contents

1	Read this first	3
1.1	First examples	4
1.2	Quick overview (expressions with <code>xintexpr</code>)	5
1.3	Printing big numbers on the page	7
1.4	Randomly chosen examples	8
1.5	More examples within this document	11
1.6	License and installation instructions	12
1.7	Recent changes	12
2	The syntax of <code>xintexpr</code> expressions	13
2.1	The three parsers	13
2.2	Built-in operators and their precedences	13
(table)	<i>Precedence levels of operators</i>	14
2.3	Built-in functions	16
(table)	<i>Functions in expressions</i>	17
2.4	Tacit multiplication	30
2.5	More examples with dummy variables	31
2.6	User defined variables	32
2.7	User defined functions	35
2.8	List operations	42
2.9	Analogies and differences of <code>\xintiexpr</code> with <code>\numexpr</code>	44
2.10	Chaining expressions for expandable algorithmics	44
3	The <code>xint</code> bundle	47
3.1	Characteristics	47
3.2	Floating point evaluations	49
3.3	Expansion matters	50
3.4	Input formats for macros	52
3.5	Output formats of macros	54
3.6	Count registers and variables	54
3.7	Dimension registers and variables	55
3.8	<code>\ifcase</code> , <code>\ifnum</code> , ... constructs	57
3.9	No variable declarations are needed	57
3.10	When expandability is too much	57
3.11	Possible syntax errors to avoid	58
3.12	Error messages	59
3.13	Package namespace, catcodes	60
3.14	Origins of the package	60
4	Some utilities from the <code>xinttools</code> package	61
4.1	Assignments	61
4.2	Utilities for expandable manipulations	62
4.3	A new kind of for loop	63
4.4	A new kind of expandable loop	63
5	Additional examples using <code>xinttools</code> or <code>xintexpr</code> or both	63
5.1	Completely expandable prime test	63
5.2	Another completely expandable prime test	65
5.3	Miller-Rabin Pseudo-Primality expandably	66
5.4	A table of factorizations	69
5.5	Another table of primes	70
5.6	Factorizing again	72
5.7	The Quick Sort algorithm illustrated	73
6	Macros of the <code>xintkernel</code> package	80
7	Macros of the <code>xintcore</code> package	85
8	Macros of the <code>xint</code> package	90
9	Macros of the <code>xintfrac</code> package	101
10	Macros of the <code>xintexpr</code> package	122
11	Macros of the <code>xinttrig</code> package	140
12	Macros of the <code>xintlog</code> package	144
13	Macros of the <code>xintbinhex</code> package	145
14	Macros of the <code>xintgcd</code> package	148
15	Macros of the <code>xintseries</code> package	151
16	Macros of the <code>xintcfrac</code> package	166
17	Macros of the <code>xinttools</code> package	181



Dependency graph for the **xint** bundle components: modules pointed to by arrows **automatically** import the modules originating the continuous line ended by an arrow. Dashed lines indicate a partial dependency, and to enable the corresponding functionalities of the lower module it is thus necessary to use a suitable `\usepackage` (L^AT_EX) or `\input` (Plain T_EX.)

bnumexpr is a separate (L^AT_EX only) package by the author which uses (by default) **xintcore** as its mathematical engine.

polexpr is a separate (L^AT_EX only) package by the author which requires **xintexpr**.

xinttrig and **xintlog** are loaded automatically by **xintexpr**; they will refuse to be loaded directly (but see `\xintreloadxinttrig`).

1 Read this first

This section provides recommended reading on first discovering the package.

This is release 1.3e.

1. `log()`, `exp()`, `log10()`, `pow10()`, `pow()` are implemented via `poormanlog`: this achieves only 8 or 9 digits of precision.... This situation is provisory, I simply was lacking the time. See `xintlog`.
2. `sin()`, `cos()`, ..., `asin()`, ... are implemented (using high level user interface), up to about 60 digits of precision, see `xinttrig`.
3. `NaN`, `+Infty`, `-Infty`, etc... and a proper internal data structure for storing floating point numbers are *yet to be implemented*.
4. `xint` can handle numbers with thousands of digits, but execution times (and `memory considerations`) limit the practical range to perhaps up to a few hundreds digits.
5. Attention that exact operations with fractions do not reduce to lowest terms (additions and subtractions use `l.c.m.` of denominators), see `reduce()`.

`xinttools` provides utilities of independent interest such as expandable and non-expandable loops. `xintgcd` and `xintcffrac` have a partial dependency on it but it must be required by user explicitly. `xintexpr` loads it automatically.

`xintcore` provides expandable macros implementing addition, subtraction, multiplication, division, and power with arbitrarily long numbers. It is loaded automatically by `xint`, and also by \TeX package `bnumexpr` in its default configuration.

`xint` extends `xintcore` with additional operations on big integers. It loads automatically `xintcore`.

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

`xintexpr` extends `xintfrac` with expandable parsers doing algebra (either exact, float, or limited to big integers) on comma separated expressions using the standard infix notations and parentheses (or sub `xintexpr`-essions). It implements tacit multiplication, functions with one or multiple arguments, Python-like slicing of lists, user-definable variables and user-definable functions, boolean two way or three way branching. Dummy variables can be used for summing or multiplying an expression over a range, or for more complicated iterative evaluations allowing `omit`, `abort`, and `break` keywords. It loads automatically `xintfrac` (hence `xint` and `xintcore`) and `xinttools`. And `xinttrig` and `xintlog` since 1.3e.

New with
1.3e

`xinttrig` trigonometrical functions for `xintexpr`, automatically loaded by it, can not be used separately.

New with
1.3e

`xintlog` logarithm, exponential, power functions for `xintexpr`, automatically loaded by it, can not be used separately.

Further modules:

`xintbinhex` is for conversions to and from binary and hexadecimal bases. Support in `xintexpr` of the \TeX " prefix for hexadecimal inputs requires this module to be loaded by user.

xintgcd implements the Euclidean algorithm and its typesetting. The macro `\xintIrr` (hence the `xintexpr` function `reduce()`) is provided independently in `xintfrac`. Also the `xintexpr` functions `gcd()` and `lcm()` are implemented directly so loading this module is not needed for them (since 1.3d).

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

xintcfrac is provided to help with the computation and display of continued fractions.

All macros from the `xint` packages doing computations are *expandable*, and naturally also the parsers provided by `xintexpr`.

The reasonable range of use of the package arithmetics is with numbers of *up to a few hundred digits*. Although numbers up to about 19950 digits are acceptable inputs, the package is not at his peak efficiency when confronted with such really big numbers having thousands of digits.¹

1.1 First examples

With `\usepackage{xintexpr}` if using \TeX , or `\input xintexpr.sty\relax` for other formats, you can do computations such as the following.

with floats:

```
\xintfloateval{3.25^100/3.2^100, 2^1000000, sqrt(1000!)}\newline
\xintfloateval{[-1] sind(37), cosd(37)}\newline % trim off last digit (via rounding)
\xintfloateval{[8] log10(12345678), pow10(0.1234)}\newline % only 8 or 9 digits!
\xintfloateval{[8] pow(2,1/3)}\newline % only 8 or 9 digits!
\xintfloateval{[8] log(10), exp(1)}\par % only 8 or 9 digits!
```

```
4.713443069476886, 9.900656229295898e301029, 6.343400192933548e1283
0.601815023152048, 0.798635510047293
7.0915149, 1.3286176
1.2599211
2.3025851, 2.7182818
```

The `\xintfloateval` macro using braces was introduced at 1.3d. Formerly, one did:

```
\thexintfloatexpr 10^-3.5\relax\ or rather \xintthefloatexpr 10^-3.5\relax
```

Most of the manual is couched using the `\xintthexpr... \relax` which is the original one from first release of `xintexpr`.

For powers with fractional exponents, see [poormanloghack](#).

with fractions:

Here is an example using a dummy a variable:

```

$$\sum_{i=1}^{25} (-1)^{i-1} \frac{1}{i^2} =$$

\xintFrac{\xinteval{reduce(add((-1)^(i-1)/i**2, i=1..25))}}$
```

```

$$\sum_{i=1}^{25} (-1)^{i-1} \frac{1}{i^2} = \frac{196669262520424458517}{238898057495217120000}$$

```

with integers:

```
\xintiieval{3^159+2^234}\newline
```

¹ The maximal handled size for inputs to multiplication is 19959 digits. This limit is observed with the current default values of some parameters of the `tex` executable (input stack size at 5000, maximal expansion depth at 10000). Nesting of macros will reduce it and it is best to restrain numbers to at most 19900 digits. The output, as naturally is the case with multiplication, may exceed the bound.

```
\xintiieval{lcm(seq(n, n=100..110))}\par
```

```
7282510957931791370332035240020194155624159839406805975150495299651205982251
590910221462160600
```

Float computations are done by default with 16 digits of precision. This can be changed via an assignment to `\xintDigits`:

```
% use braces (or a LaTeX environment) to limit the scope of the \xintDigits assignment
{\xintDigits := 88;\xintfloateval{3.25^100-3.2^100}}\par
```

```
1.21555496665826543032280663867259188613692951880893931399567325222064394651297590367526e51
```

Trigonometrical function need a reload after modifying the float precision, this is done by `\xintreloadxinttrig`.

We can even try daring things:²

```
{\xintDigits:=500;\printnumber{\xintfloateval{sqrt(2)}}}
```

```
1.4142135623730950488016887242096980785696718753769480731766797379907324784621070388503875342
327641572735013846230912297024924836055850737212644121497099935831413222665927505592755799952
050115278206057147010955997160597027453459686201472851741864088919860955232923048430871432142
508397626036279952514079896872533965463318088296406206152583523950547457502877599617298355752
220337531857011354374603408498847160386899970699004815030544027790316454247823068492936918622
15805784631115966687130130156185689872372
```

All operations executed by the parsers are based on underlying macros from packages `xintfrac` and `xint` which are loaded automatically by `xintexpr`. With `xintbinhex` loaded the parsers can handle hexadecimal notation on (even fractional) input.

All macros doing computations ultimately rely on (and reduce to) the `\numexpr` primitive from ϵ -TeX. These ϵ -TeX extensions date back to 1999 and are by default incorporated into the `pdftex` etc... executables from major modern TeX installations since more than ten years now. Only the `te` x binary does not benefit from them, as it has to remain the original D. Knuth's software, but one can then use `etex` on the command line. PDFTeX (in pdf or dvi output mode), LuaTeX, XeTeX all include the ϵ -TeX extensions.

1.2 Quick overview (expressions with `xintexpr`)

This section gives a first few examples of using the expression parsers which are provided by package `xintexpr`. See [section 2](#) for a more detailed description, and perhaps read [subsection 2.1](#) before coming back here.

Loading `xintexpr` automatically also loads packages `xinttools` and `xintfrac`. The latter loads `xint` which loads `xintcore`. All three provide the macros which ultimately do the computations associated in expressions with the various symbols like `+`, `*`, `^`, `!` and functions such as `max`, `sqrt`, `gcd`. The package `xinttools` does not handle computations but provides some useful utilities.

Release 1.2h defines `\thexintexpr` as synonym to `\xinttheexpr`, `\thexintfloatexpr` as synonym of `\xintthefloatexpr`, etc...

New with 1.3d

Release 1.3d defines `\xinteval`, `\xintieval`, `\xintiieval`, and `\xintfloateval` which use braces to delimit their arguments, which will be more familiar to \TeX users than the `\xinttheexpr... \relax` syntax. For example:

```
\xinteval{1+2+3}, \xintfloateval{111/123}, \xintfloateval{[3] 111/123}
6, 0.9024390243902439, 0.902
```

This documentation however uses systematically `\xinttheexpr... \relax` or `\thexintexpr... \relax` syntax for legacy reasons. In this input syntax the `\relax` does not have to be physically present it may arise from pure expansion (see [section 10](#)).

² The `\printnumber` is not part of the package, see [subsection 1.3](#).

Notice that for the `\xinttheiexpr` and `\xintthefloatexpr` parsers which admit an optional argument, this optional argument will have to be located *inside* the braces when using `\xintieval` or `\xintfloateval` input syntax, as exemplified above.

The main reason is that it is sometimes useful to hide the square brackets, for example when using the `\num` macro from package [siunitx](#).

The square root extraction `sqrt()` is allowed in `\xintexpr.\relax` but naturally can't return an exact value, it computes as if it was in `\xintfloatexpr.\relax`.

The synonymous power operators `^` and `**` allow only integral exponents (non-negative in the integer only parser `\xintiexpr`) in `\xintexpr` and half-integral exponents in `\xintfloatexpr`. Since 1.3e fractional powers via the `pow()` function are available, see [xintlog](#). But they achieve currently only about 8 digits of precision.

New with
1.3e

Here is a (partial) list of the recognized symbols:

- the comma (to separate distinct computations or arguments to a function),
- parentheses,
- **operators**:
 - `+`, `-`, `*`, `/`,
 - powers via `^` or equivalently `**`, see [xintlog](#) for allowing with them fractional exponents,
 - `//` for floored division and `/:` its associated modulo,
 - branching via `(x)?{x non zero}{x zero}` and `(x)??{x<0}{x=0}{x>0}` syntax,
 - boolean logic `!`, `&&` or `'and'`, `||` or `'or'`,
 - comparison operators `=` (or `==`), `<`, `>`, `<=`, `>=`, `!=`,
 - factorial via the post-fix operator `!`.
- the `"` is used to prefix hexadecimal input (uppercase, not lowercase); but package [xintbinhex](#) must be loaded additionally to [xintexpr](#),
- various **functions**:
 - rounding and truncating [round\(\)](#), [trunc\(\)](#) to a given fixed point precision or with [float\(\)](#) to a given floating point precision,
 - the square-root [sqrt\(\)](#) achieves correct rounding in arbitrary precision,
 - the [binomial\(\)](#) and (also partial) [factorial\(\)](#),
 - [gcd\(\)](#) and [lcm\(\)](#) for general fractional operands,
 - randomness related functions such as [random\(\)](#) and [randrange\(\)](#) for random floats or integers; they require that \TeX engine provides `\pdfuniformdeviate` or `\uniformdeviate` primitive,
 - trigonometrical functions [xinttrig](#),
 - logarithm and exponential [xintlog](#).
- the capacity to work with dummy variables using functions or generators such as [add\(\)](#), [mul\(\)](#), [seq\(\)](#), [subs\(\)](#), [rseq\(\)](#), [iter\(\)](#), [rrseq\(\)](#), [iterr\(\)](#).

See [subsection 10.1](#) for basic information and [section 2](#) for a complete description.

The normal mode of operation of the parsers is to unveil the parsed material token by token. This means (apart from some exceptions) that all tokens may arise from expansion of encountered macros (or active characters). For example a closing parenthesis does not have to be immediately visible, it may arise later from expansion.

However, this general behaviour has exceptions, in particular constructs with dummy variables need at some location immediately visible balanced parentheses and commas.

The expansion stops only when the ending `\relax` has been found; it is then removed from the token stream, and the final computation result is inserted.

Here is an example of a computation:

```
\xinttheexpr (31.567^2 - 21.56*52)^3/13.52^5\relax
-1936508797861911919204831/4517347060908032[-8]
```

This illustrates that `\xinttheexpr.\relax` does its computations *exactly*. The same example as a floating point evaluation:

```
\xintthefloatexpr (31.567^2 - 21.56*52)^3/13.52^5\relax
-4.286827582100044
```

Again, all computations done by `\xinttheexpr.\relax` are completely exact. Thus, very quickly very big numbers are created (and computation times increase, not to say explode if one goes into handling numbers with thousands of digits). To compute something like 1.23456789^{10000} it is thus better to opt for the floating point version:

```
\xintthefloatexpr 1.23456789^10000\relax
1.411795173056392e915
```

(we can deduce that the exact value has $80000+916=80916$ digits). A bigger example (the scope of the assignment to `\xintDigits` is limited by the braces):

```
{\xintDigits:=24; \xintthefloatexpr 1.23456789123456789^123456789\relax }
```

1.90696640042856610942910e11298145 (<- notice the size of the power of ten: this surely largely exceeds your pocket calculator abilities).

Some examples with dummy variables:

```
\xinttheiexpr add(i^5, i=100..200)\relax\par
\noindent\xinttheexpr add(x/(x+1), x = 1000..1014)\relax\par
\noindent\xinttheexpr reduce(add(x/(x+1), x = 1000..1014))\relax
```

10665624997500

4648482709767835886400149017599415343/310206597612274815392155150733157360

4648482709767835886400149017599415343/310206597612274815392155150733157360

In this example, the fraction obtained by addition was thus already irreducible, but this is not always the case:

By default, the basic operations on fractions are not followed in an automatic manner by reduction to smallest terms: A/B multiplied by C/D returns AC/BD , and A/B added to C/D uses $\text{lcm}(B, D)$ as denominator.

Changed at 1.3!

Make sure to read [section 10](#), [section 2](#) and [subsection 3.5](#).

1.3 Printing big numbers on the page

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

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

It may be used like this:

```
\printnumber {\xintiiQuo{\xintiiPow {2}{1000}}{\xintiiFac{100}}}
```

or as `\printnumber\mybiginteger` or `\printnumber{\mybiginteger}` if `\mybiginteger` was previously defined via a `\newcommand`, a `\def` or an `\edef`.

An alternative is to suitably configure the thousand separator with the `numprint` package (see [footnote 6](#). This will not allow linebreaks when used in math mode; I also tried `siunitx` but even in text mode could not get it to break numbers accross lines). Recently I became aware of the `seqsplit`

package³ which can be used to achieve this splitting accross lines, and does work in inline math mode (however it doesn't allow to separate digits by groups of three, for example).

1.4 Randomly chosen examples

This section is now quite old...

Here are some examples of use of the package macros. The first one uses only the base module *xint*, the next one requires the *xintfrac* package, which deals with decimal numbers, scientific numbers (lowercase *e*), and also fractions (it loads automatically *xint*). Then some examples with expressions, which require the *xintexpr* package (it loads automatically *xintfrac*). And finally some examples using *xintseries*, *xintgcd* which are among the extra packages included in the *xint* distribution.

The printing of the outputs will either use a custom `\printnumber` macro as described in the previous section, or sometimes the `\np` macro from the *numprint* package (see footnote 6).

- 123456⁹⁹:
`\xintiiPow {123456}{99}`: 1147381811662665566332733300084545867470254804234261029758895452
43735908946970320276226470542663205834690270868221168133415250032403876277616895322211762
34295872033762216088606915850757168019716710712087697033536507377487778737784987816067492
99979836658125172327521549705416595667384911533326748541075607669718906235189958323778262
36999811095323939932351899922205645878127014958776791431677354372538584459487155941215192
74163986661258969837372587167573949494355201709502618658016651990307184144322311696783762
96
- 1234/56789 with 1500 digits after the decimal point:
`\xintTrunc {1500}{1234/56789}\dots`: 0.021729560302171195125816619415731919914067865255592
52737325890577400552923981757030410818996636672595044815016992727464825934600010565426402
30005810984521650319604148690767578228177992216802549789571924140238426455827713113455072
05242212400288788321682015883357692510873584673088098047157019845392593636091496592649982
50323125957491767771927662047227456021412597510081177692863054464773107467995562520910732
97559386500906865766257549877617144165243268942929088379791861099860888552360492348870372
98270791878708904893553328989769145433094437302998820194051664935110672841571431087006282
64287097853457535790381940164468471006709045765905368997517124795294863441863741217489302
25057669619116378171829051400799450597827043969782880487418338058426808008593213474440472
26267410942259944707601824296958918100336332740495518498300727253517406539998943457359692
99418901547834968039585130923242177182200778319745021042807585976157354417228688654492942
75778759971121167831798411664230748912641532691190195284298015460740636390850340735001492
67687404250823222807233795277254397858740248991882230713694553522689253200443747908926022
44061349909313423374245012238285583475673105707091162020813890013911144763950765112962012
72920812129109510644667101023085456690556269700117980594833506488932715842856891299371352
71290214654246420961805983553152899329095423409463100248287520470513655813625878251069742
94233038088362182817094859920054940217295603021711951258166194157319199140678652555952732
732589057740055292398175703041081899663667... .
- 0.99⁻¹⁰⁰ with 200 (+1) digits after the decimal point.
`\xinttheiexpr [201] .99^-100\relax`: 2.731999026429026003846671721257837435505351642938572
20708334305725082464555187053430448143013784806140368055624765019253070342696854891531942
616612271015920671913840348851485747943086470963920731779793038
Notice that this is rounded, hence we asked `\xinttheiexpr` for one additional digit. To get a truncated result with 200 digits after the decimal mark, we should have issued `\xinttheiexpr 2 trunc(.99^-100,200)\relax`, rather.

³ <http://ctan.org/pkg/seqsplit>

$$-220045702773594816771390169652074193009609478853 \times (7^{200} - 3^{200}) + 14325894936276369318591320683268320465474416863387714089158381672478991921132820119127462437158039177754976857191287693144240605066991456336143205677696774891 \times (2^{200} - 1) = 1803403947125 = 1803403947125$$

- The Euclidean algorithm applied to 22,206,980,239,027,589,097 and 8,169,486,210,102,119,257: (with `xintgcd`)⁵

```
\xintTypesetEuclideanAlgorithm {22206980239027589097}{8169486210102119257}
22206980239027589097 = 2 × 8169486210102119257 + 5868007818823350583
8169486210102119257 = 1 × 5868007818823350583 + 2301478391278768674
5868007818823350583 = 2 × 2301478391278768674 + 1265051036265813235
2301478391278768674 = 1 × 1265051036265813235 + 1036427355012955439
1265051036265813235 = 1 × 1036427355012955439 + 228623681252857796
1036427355012955439 = 4 × 228623681252857796 + 121932630001524255
228623681252857796 = 1 × 121932630001524255 + 106691051251333541
121932630001524255 = 1 × 106691051251333541 + 15241578750190714
106691051251333541 = 6 × 15241578750190714 + 15241578750189257
15241578750190714 = 1 × 15241578750189257 + 1457
15241578750189257 = 10460932567048 × 1457 + 321
1457 = 4 × 321 + 173
321 = 1 × 173 + 148
173 = 1 × 148 + 25
148 = 5 × 25 + 23
25 = 1 × 23 + 2
23 = 11 × 2 + 1
2 = 2 × 1 + 0
```

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

```
\def\coeff #1{\xintiRound {12}{1/\xintiiSqr{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 $\frac{\pi^2}{144} - \frac{1}{162} = 0.062,366,079,945,836,595,346,844,45\dots$ ⁶ 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/\xintiiSqr{\xintiiMul{the\numexpr 2*#1-3\relax}
{the\numexpr 2*#1+3\relax}}[0]}}
```

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

```
\numprint{\xintFloatPow [24]{2}{999999999}}
2.306,488,000,584,534,696,558,06 × 10301,029,995
```

where the `numprint` package was used (footnote 6), directly in text mode (it can also naturally be used from inside math mode). `xint` provides a simple-minded `\xintFrac` typesetting macro,⁷ which is math-mode only:

```
\xintFrac{\xintFloatPow [24]{2}{999999999}}$
230648800058453469655806 · 10301029995
```

⁵ this example is computed tremendously faster than the other ones, but we had to limit the space taken by the output hence picked up rather small big integers as input. ⁶ This number is typeset using the `numprint` package, with `\npthousandsep {,}\hskip 1pt plus .5pt minus .5pt}`. But the breaking across lines works only in text mode. The number itself was (of course...) computed initially with `xint`, with 30 digits of π as input. See [how xint may compute \$\pi\$ from scratch](#). ⁷ Plain T_EX users of `xint` have `\xintFwOver`.

The exponent differs, but this is because `\xintFrac` does not use a decimal mark in the significant of the output. Admittedly most users will have the need of more powerful (and customizable) number formatting macros than `\xintFrac`.⁸ We have already mentioned `\numprint` which is used above, there is also `\num` from package `siunitx`. The raw output from

```
\xintFloatPow [24]{2}{999999999}
is 2.30648800058453469655806e301029995.
```

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

```
\newwrite\outstream
\immediate\openout\outstream \jobname-out\relax
\immediate\write\outstream {\xintiiQuo{\xintiiPow{2}{1000}}{\xintiiFac{100}}}
% \immediate\closeout\outstream
```

The tex run creates a file `myfile-out.tex`, and then writes to it the quotient from the Euclidean division of 2^{1000} by $100!$. The number of digits is `\xintLen{\xintiiQuo{\xintiiPow{2}{1000}}{\xintiiFac{100}}}` which expands (in two steps) and tells us that $[2^{1000}/100!]$ has 144 digits. This is not so many, let us print them here: 11481324964150750548227839387255106625928055177841861728836634780658265418947047379704195357988766304843582650600615037495317077293118627774829601.

1.5 More examples within this document

- The utilities provided by `xinttools` (section 17), some completely expandable, others not, are of independent interest. Their use is illustrated through various examples: among those, it is shown in subsection 5.7 how to implement in a completely expandable way the [Quick Sort algorithm](#) and also how to illustrate it graphically. Other examples include some dynamically constructed alignments with automatically computed prime number cells: one using a completely expandable prime test and `\xintApplyUnbraced` (subsection 5.1), another one with `\xintFor*` (subsection 5.5).
- One has also a [computation of primes within an \edef](#) (subsection 17.14), with the help of `\xintiloop`. Also with `\xintiloop` an automatically generated table of factorizations (subsection 5.4).
- The code for the title page fun with Fibonacci numbers is given in subsection 2.10 with `\xintFor*` joining the game.
- The computations of π and $\log 2$ (subsection 15.11) using `xint` and the computation of the [convergents of e](#) with the further help of the `xintcfrac` package are among further examples.
- Also included, an [expandable implementation of the Brent-Salamin algorithm](#) for evaluating π .
- The subsection 5.3 implements expandably the Miller-Rabin pseudo-primality test.
- The functionalities of `xintexpr` are illustrated with various other examples, in [subsubsection 2.7.1, Functions with dummy variables](#), subsection 2.5 or [Recursive definitions](#).

Almost all of the computational results interspersed throughout the documentation are not hard-coded in the source file of this document but are obtained via the expansion of the package macros during the \TeX run.

⁸ There should be a `\xintFloatFrac`, but it is lacking.

1.6 License and installation instructions

xint is made available under the [LaTeX Project Public License 1.3c](#) and is included in the major \TeX distributions, thus there is probably no need for a custom install: just use the package manager to update if necessary *xint* to the latest version available.

The [README](#) files on [CTAN](#) explain how to proceed with a custom installation.

On \TeX distributions with a "texdoc" or similar utility,
`texdoc --list xint`

will offer to display one of those files:

- [xint.pdf](#) (this file),
- [sourcexint.pdf](#) (source code),
- [README](#), [README.pdf](#), [README.html](#),
- [CHANGES.pdf](#), and [CHANGES.html](#).

1.7 Recent changes

This is release 1.3e of 2019/04/05.

For more information see [CHANGES.html](#).

Internet: <http://mirrors.ctan.org/macros/generic/xint/CHANGES.html>

The formatted source code is available in file [sourcexint.pdf](#):

`texdoc sourcexint`

1.3e (2019/04/05):

- An *xinttrig* library is automatically loaded by *xintexpr* and provides direct and inverse trigonometrical functions using either degrees or radians (up to 60 digits). It is for the most part implemented using high level user interface, but will probably get some optimizations in future (and perhaps extension to more digits).
- An *xintlog* library is loaded automatically; it uses *poormanlog* to provide logarithms and exponentials. Support for more digits is planned for the future: the *poormanlog* support achieves only about 8 or 9 digits of precision. On the other hand the functions are fast. Perhaps we will keep the current functions achieving limited precision under some other names in future (they are amply precise enough for plots).
- Under the hood refactoring of `\xintNewExpr` related matters for support of user-defined functions; fixed bugs related to functions with no variables. Added `\xintdefefunc`, `\xintdeffloatefunc`, `\xintdefiiefunc` and documented the "protected" nature of the functions defined by the original variants `\xintdeffunc` et al. This whole area is complex and to be still considered work in progress.
- Breaking change: only the `\xintexpr...\relax` syntax is accepted for sub-expressions when doing function definitions via `\xintdeffunc` and `\xintdefefunc` et al. One can not use `\xinteval`, `\xintieval`, `\xintfloateval` for sub-evaluations inside such definitions (anyway there are not efficient there because the parser will have to recollect the digits of the value whereas *xintexpr* keeps some private internal format).
- `inv()`, `ilog10()`.
- `sfloat()`, slight modification of behaviour of `qfloat()`.
- `\xintensuredummy`, `\xintrestorelettervar`.
- the optional argument of `\xintfloatexpr` can be negative, it then tells to round the result to a (rounded) float with a precision equal to `\xinttheDigits` diminished by this argument.

1.3d (2019/01/06):

- `gcd()` and `lcm()` in `\xintexpr...\relax` now handle general arguments, without converting them to integers,
- It is not needed anymore to load package *xintgcd* to benefit from `gcd()` and `lcm()` in the parsers.
- `\xintifsgnexpr`, `\xintifsgnfloatexpr`, `\xintifsgniexpr`.
- `\xintunassignexprfunc` and variants for the other parsers.
- `isone()` and `isint()`.

- `\xinteval`, `\xintieval`, `\xintiieval`, and `\xintfloateval`. Attention: these names were formerly used with some other (barely documented) meanings, for which `\xintexpr`, `\xintiexpr`, etc... are now used.
- Sadly, in `\xintiexpr... \relax` division with a zero dividend and a one-digit divisor got broken at 1.2p. Fixed. Thanks to KPM for report. Sorry for long delay in releasing the bugfix, which was done shortly after 1.3c release.

2 The syntax of *xintexpr* expressions

.1	The three parsers.....	13	.6	User defined variables.....	32
.2	Built-in operators and their precedences ..	13	.7	User defined functions	35
(table)	<i>Precedence levels of operators</i>	14	.8	List operations	42
.3	Built-in functions.....	16	.9	Analogies and differences of <code>\xintiexpr</code> with <code>\numexpr</code>	44
(table)	<i>Functions in expressions</i>	17	.10	Chaining expressions for expandable algo- rithmics	44
.4	Tacit multiplication.....	30			
.5	More examples with dummy variables.....	31			

2.1 The three parsers

There are three expression parsers and two subsidiary ones. They all admit comma separated expressions, and will then output a comma separated list of results.

- `\xinttheiexpr ... \relax` does exact computations *only on integers*. The forward slash `/` does the *rounded* integer division to match behaviour of `\the\numexpr <int>/<int>\relax`.⁹ There are two square root extractors `sqrt()` and `sqrtr()` for truncated and rounded square roots. Scientific notation `6.02e23` is *not* accepted on input, one needs to wrap it as `num(6.02e23)` which will convert to an integer notation `6020000000000000000000`.
- `\xintthefloatexpr ... \relax` does computations with a given precision `P`, as specified via a prior assignment `\xintDigits:=P;`. The default is `P=16` digits. An optional argument controls the precision for *formatting the output* (this is not the precision of the computations themselves). The four basic operations and the square root realize *correct rounding*.¹⁰
- `\xinttheexpr ... \relax` handles integers, decimal numbers, numbers in scientific notation and fractions. The algebraic computations are done *exactly*. The `sqrt()` function is available and obeys either the `\xintDigits` precision or its second optional argument.

Two derived parsers:

- `\xinttheiexpr ... \relax` does all computations like `\xinttheexpr ... \relax` but rounds the result to the nearest integer. With an optional positive argument `[D]`, the rounding is to the nearest fixed point number with `D` digits after the decimal mark.
- `\xinttheboolexpr ... \relax` does all computations like `\xinttheexpr ... \relax` but converts the result to 1 if it is not zero (works also on comma separated expressions). See also the booleans `\xintifboolexpr`, `\xintifbooliexpr`, `\xintifboolfloatexpr`, `\xintifsgnexpr`, `\xintifsgniexpr`, `\xintifsgnfloatexpr` (they do not handle comma separated expressions).

Release 1.3d provides `\xinteval`, `\xintieval`, `\xintiieval`, `\xintfloateval`.

New with
1.3d

2.2 Built-in operators and their precedences

The [Table 1](#) is hyperlinked to the more detailed discussion at each level.

∞ At this highest level of precedence, one finds:

⁹ For floored integer division, see the `//` operator. ¹⁰ when the inputs are already floating point numbers with at most `P`-digits mantissas.

<p>∞: at this top level the non-operator syntax elements whose parsing is always done prior to executing operators preceding them:</p> <ul style="list-style-type: none"> • built-in or user-defined functions, • variables, • and the intrinsic constituents of numbers: decimal mark <code>.</code>, <code>e</code> and <code>E</code> of scientific notation, hexadecimal prefix <code>"</code>. 	
Precedence	``Operators'' at this level
10	the factorial (postfix) operator <code>!</code> and the conditional branching operators <code>?</code> and <code>??</code>
=	the minus sign <code>-</code> as unary operator acquires the precedence level of the previous infix operator
9	the power <code>^</code> , <code>**</code> operators
8	the action of tacit multiplication
7	the multiplication, division, and modulo operators <code>*</code> , <code>/</code> , <code>//</code> , <code>/:</code> (aka <code>'mod'</code>)
6	the addition and subtraction <code>+</code> , <code>-</code>
5	the comparison operators <code><</code> , <code>></code> , <code>==</code> , <code><=</code> , <code>>=</code> , <code>!=</code>
4	Boolean conjunction <code>&&</code> and its alias <code>'and'</code>
3	Boolean disjunction <code> </code> and <code>'or'</code> , and <code>'xor'</code> ; also the sequence generators <code>..</code> , <code>..[</code> , <code>]..</code> , and the Python slicer <code>:</code> have this precedence
2	the comma <code>,</code>
1	the parentheses <code>(,)</code> , list brackets <code>[,]</code> , semicolon <code>;</code> in an <code>iter()</code> or <code>rseq()</code>
<ul style="list-style-type: none"> • In case of equal precedence, the rule is left-associativity: the first encountered operation is executed first. Tacit multiplication has an elevated precedence level hence seemingly breaks left-associativity: <code>(1+2)/(3+4)5</code> is computed as <code>(1+2)/((3+4)*5)</code> and <code>x/2y</code> is interpreted as <code>x/(2*y)</code> when using variables. • List variants <code>^[</code>, <code>**[</code>, <code>]^</code>, <code>]**</code>, <code>*[</code>, <code>/[</code>, <code>]*</code>, <code>]/</code>, <code>+[</code>, <code>-[</code>, <code>]+</code>, <code>]-</code>, share the precedence level of their respective associated operators on numbers. • There may be some evolution in future, perhaps to distinguish some of the constructs which currently share the same precedence or to make room for added syntax elements. 	

Table 1: Precedence levels

- **functions** and **variables**: we approximately describe the situation as saying they have highest precedence. Functions (even the logic functions `!` and `?` whose names consists of a single non-letter character) must be used with parentheses. These parentheses may arise from expansion after the function name is parsed (there are exceptions which are documented at the relevant locations.)

- the `.` as decimal mark; the number scanner treats it as an inherent, optional and unique component of a being formed number. One can do things such as

```
\xinttheexpr 0.^2+2^.0\relax
```

which is 0^2+2^0 and produces `1`.

Since release 1.2 an isolated decimal mark `.` is illegal input in `\xintexpr.\relax`, although it remains legal as argument to the macros of `xintfrac`.

- the `e` and `E`, for scientific notation are intrinsic constituents of number denotations, like the decimal mark.
- the `"` for hexadecimal numbers: it is allowed only at locations where the parser expects to start forming a numeric operand, once encountered it triggers the hexadecimal scanner which looks for successive hexadecimal digits as usual skipping spaces and expanding forward everything; letters (only `ABCDEF`, not `abcdef`), an optional dot (allowed directly in front) and an optional (possibly empty) fractional part. The `"` functionality requires to load package `xintbinhex`.

```
\xinttheexpr "FEDCBA9876543210\relax\newline
\xinttheiexpr 16^5-("F75DE.0A8B9+"8A21.F5746+16^-5)\relax
```

```
18364758544493064720
```

```
0
```

10 The postfix operators `!` and the branching conditionals `?`, `??`.

- `!` computes the factorial of an integer. Attention that the boolean equality test `==` must not follow directly such a `<digits or variable>!` because the parser will handle `!=` as the boolean inequality test... the remaining `=` then causes a parsing error. It is even worse if one uses a single `=` following the `!` because no error arises but an un-intended interpretation. Use parentheses: `(3!)==10`.

- `?` is used as `(stuff)?{yes}{no}`. It evaluates `stuff` and chooses the `yes` branch if the result is non-zero, else it executes `no`. After evaluation of `stuff` it acts as a macro with two mandatory arguments within braces, chooses the correct branch *without evaluating the wrong one*. Once the braces are removed, the parser scans and expands the uncovered material so for example

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

is legal and computes $5+6 \cdot 2^3 = 238333$. It would be better practice to include here the 2^3 inside the branches. The contents of the branches may be arbitrary as long as once glued to what is next the syntax is respected: `\xintexpr (3>2)?{5+(6){7-(1}2^3)\relax` also works.

- `??` is used as `(stuff)??{<0}{=0}{>0}`, where `stuff` is anything, its sign is evaluated and depending on the sign the correct branch is un-braced, the two others are discarded with no evaluation of their contents. The un-braced branch will then be parsed as usual.

```
\def\x{0.33}\def\y{1/3}
```

```
\xinttheexpr (\x-\y)??{sqrt}{0}{1/}\y-\x\relax=5773502691896257[-17]
```

- `=` The minus sign `-` as prefix unary operator inherits the precedence of the infix operator it follows. `\xintexpr -3-4*-5^-7\relax` evaluates as $(-3)-(4*(-(5^(-7))))$ and `-3^-4*-5-7` as $(-(3^(-4))*(-5))-7$.

$2^{\wedge}10$ is perfectly accepted input, no need for parentheses

- 9 The power operator `^`, or equivalently `**`. It is left associative: `\xinttheiexpr 2^2^3\relax` evaluates to 64, not 256. See `\xintFloatPower` for additional information.

Also at this level the list operators `^[`, `**[`, `]^`, and `]**`.

8 see [Tacit multiplication](#).

7 Multiplication and division `*`, `/`. The division is left associative, too: `\xinttheiexpr 100/500/2\relax` evaluates to 1, not 4.

Also the floored division `//` and its associated modulo `/:` (equivalently `'mod'`, quotes mandatory).

Also at this level the list operators `*[`, `/[`, `]*` and `]/`.

In an `\xintiexpr`-ession, `/` does rounded division, to behave like the `/` of `\numexpr`.

Infix operators all at the same level of precedence are left-associative.¹¹ Apply parentheses for disambiguation.

```
\xinttheiexpr 100000//13, 100000/:13, 100000 'mod' 13, trunc(100000/13,10),
trunc(100000/:13/13,10)\relax
```

7692, 4, 4, 7692.3076923076, 0.3076923076

6 Addition and subtraction `+`, `-`. According to the rule above, `-` is left associative: `\xinttheiexpr 100-50-2\relax` evaluates to 48, not 52.

Also the list operators `+[`, `-[`, `]+`, `]-` are at this precedence level.

5 Comparison operators `<`, `>`, `=` (same as `==`), `<=`, `>=`, `!=` all at the same level of precedence, use parentheses for disambiguation.

4 Conjunction (logical and) `&&` or equivalently `'and'` (quotes mandatory).¹²

3 Inclusive disjunction (logical or) `||` and equivalently `'or'` (quotes mandatory).

Also the `'xor'` operator (quotes mandatory) is at this level.

Also the list generation operators `..`, `..[`, `]` are at this level.

Also the `:` for Python slicing of lists.

2 The comma: with `\xinttheiexpr 2^3,3^4,5^6\relax` one obtains as output 8, 81, 15625.¹³

1 The parentheses. The list outer brackets `[`, `]` share the same functional precedence as parentheses. The semi-colon `;` in an `iter` or `rseq` has the same precedence as a closing parenthesis.¹⁴

2.3 Built-in functions

See [Table 2](#) whose elements are hyperlinked to the corresponding definitions.

Functions are at the same top level of priority. All functions even `?()` and `!()` require parentheses around their arguments.

Miscellaneous notes:

New with
1.3d



- since release 1.3d `gcd()` and `lcm()` are extended to apply to fractions too, and they do NOT require the loading of `xintgcd`,
- The randomness related functions `random()`, `qrand()` and `randrange()` require that the TeX engine provides the `\uniformdeviate` or `\pdfuniformdeviate` primitive. This is currently the case for `pdftex`, `(u)ptex`, `luatex`, and will be for `xetex` starting with TeXLive 2019.
- `to gl()` is provided for the case `etoolbox` package is loaded,
- `bool()`, `to gl()` use delimited macros to fetch their argument and the closing parenthesis must be explicit, it can not arise from the spot expansion. The same holds for `qint()`, `qfrac()`, `qfloat()`, `qraw()`, `random()` and `qrand()`.
- Also [functions with dummy variables](#) use delimited macros for some tasks. See the relevant explanations there.



¹¹ i.e. the first two operands are operated upon first. ¹² with releases earlier than 1.1, only single character operators `&` and `|` were available, because the parser did not handle multi-character operators. Their usage in this rôle is now deprecated, and they may be assigned some new meaning in the future. ¹³ The comma is really like a binary operator, which may be called "join". It has lowest precedence of all (apart the parentheses) because when it is encountered all postponed operations are executed in order to finalize its *first* operand; only a new comma or a closing parenthesis or the end of the expression will finalize its *second* operand. ¹⁴ It is not apt to describe the opening parenthesis as an operator, but the closing parenthesis is more closely like a postfix unary operator. It has lowest precedence because when it is encountered all postponed operations are executed to finalize its operand. The start of this operand was decided by the opening parenthesis.

!()	cotd()	lcm()	randrange()
?()	cotg()	len()	reduce()
`*`()	csc()	log()	rem()
`+`()	cscd()	log10()	reversed()
abs()	divmod()	max()	round()
add()	even()	min()	rrseq()
all()	exp()	mod()	rseq()
any()	factorial()	mul()	sec()
acos()	first()	not()	secd()
acosd()	float()	num()	seq()
Arg()	floor()	odd()	sgn()
Argd()	frac()	pArg()	sin()
asin()	gcd()	pArgd()	sinc()
asind()	if()	pfactorial()	sind()
atan()	ifint()	pow()	sqr()
atand()	ifone()	pow10()	sqrt()
atan2()	ifsgn()	preduce()	sqrtr()
atan2d()	ilog10()	qfloat()	subs()
binomial()	isint()	qfrac()	tan()
bool()	isone()	qint()	tand()
ceil()	iter()	qrand()	tg()
cos()	iterr()	qraw()	togl()
cosd()	inv()	quo()	trunc()
cot()	last()	random()	xor()

Table 2: Functions (click on names)

- .3.1 Functions with no argument 18
- .3.2 Functions with a single (numeric) argument 19
- .3.3 Functions with an alphabetical argument 21
- .3.4 Functions with one mandatory and a second but optional argument 22
- .3.5 Functions with two arguments 23
- .3.6 Functions with 3 or 4 arguments 25
- .3.7 Functions with an arbitrary number of arguments 26
- .3.8 Functions requiring dummy variables 27
- .3.9 Trigonometrical functions 30
- .3.10 Logarithm, exponential and power functions 30

2.3.1 Functions with no argument

New with
1.3b

random() returns a random float x verifying $0 \leq x < 1$. It obeys the prevailing precision as set by `\xintDigits`: i.e. with P being the precision the random float multiplied by 10^P is an integer, uniformly distributed in the $0..10^P-1$ range.

This description implies that if x turns out to be <0.1 then its (normalized) mantissa has $P-1$ digits and a trailing zero, if $x < 0.01$ it has $P-2$ digits and two trailing zeros, etc... This is what is observed also with Python's `random()`, of course with `10` replaced there by radix `2`.

```
\pdfsetrandomseed 12345
\xintDigits:=37;%
\xintthefloatexpr random()\relax\newline
\xintthefloatexpr random()\relax\par
```

```
0.2415544817596207455547929850209500042
```

```
0.2584863529993996627285461554203021352
```



Due to the way `\xintexpr... \relax` are handled (see [subsection 10.5](#)), Monte-Carlo type simulations using expressions may relatively easily exhaust \TeX memory. If possible use `\xintNewFloatExpr` to construct from such expressions involving the `random()` function macros not creating the memory impact which is described in [subsection 10.5](#).

grand() returns a random float $0 \leq x < 1$ using `16` digits of precision (i.e. $10^{16}x$ is an integer). This is provided when speed is a at premium as it is optimized for precision being precisely `16`.

New with
1.3b

```
% still with 37 digits as prevailing float precision
\xintthefloatexpr grand(), random()\relax\newline
\xintDigits:=16;%
\xintthefloatexpr grand(), random()\relax\par
```

```
0.488356899132776500000000000000000000, 0.09165461826072383107532471669335645230
```

```
0.9069127435402274, 0.9106687541716861
```

One can use both `grand()` and `random()` inside the `\xintexpr` parser too. But inside the integer only `\xintiexpr` parser they will cause some low-level error as soon as they get involved in any kind of computation as they use an internal format not recognized by the integer-only parser.

See further `randrange()`, which generates random integers.

Currently there is no `uniform()` function¹⁵ but it can be created by user:

```
\xintdefloatfunc uniform(a, b):= a + (b-a)*random();
\romannumeral\xintreplicate{10}%
{%
  \xintthefloatexpr uniform(123.45678, 123.45679)\relax\newline
}%
```

```
123.4567849497100
```

```
123.4567812033226
```

```
123.4567863308250
```

```
123.4567896366777
```

```
123.4567849656655
```

```
123.4567849908270
```

```
123.4567889123433
```

```
123.4567896262979
```

¹⁵ Because I am not sure how to handle rounding issues: should the computation proceed exactly and a rounding be done only at very end?

```
123.4567846543719
123.4567832664043
```

2.3.2 Functions with a single (numeric) argument

num(x) truncates to the nearest integer (truncation towards zero). It has the same sign as *x*, except of course with $-1 < x < 1$ as then **num(x)** is zero.

```
\xinttheexpr num(3.1415^20), num(1e20)\relax
```

8764785276, 10000000000000000000 The output is an explicit integer with as many zeros are as necessary. Even in float expressions, there will be an intermediate stage where all needed digits are there, but then the integer is immediately reparsed as a float to the target precision, either because some operation applies to it, or from the output routine of [\xintfloatexpr](#) if it stood there alone. Hence, inserting something like **num(1e10000)** is costly as it really creates ten thousand zeros, even though later the whole thing becomes a float again. On the other hand naturally **1e10000** without **num()** would be simply parsed as a floating point number and would cause no specific overhead.

frac(x) fractional part. For all numbers $x = \text{num}(x) + \text{frac}(x)$, and **frac(x)** has the same sign as *x* except when *x* is an integer, as then **frac(x)** vanishes.

```
\xintthefloatexpr frac(-355/113), frac(-1129.218921791279)\relax
```

```
-0.1415929203539820, -0.2189217912790000
```

reduce(x) reduces a fraction to smallest terms

```
\xinttheexpr reduce(50!/20!/20!/10!)\relax
```

```
1415997888807961859400
```

Recall that this is NOT done automatically, for example when adding fractions.

preduce(x) internally, fractions may have some power of ten part (for example when they got input in scientific notation). This function ignores the decimal part when doing the reduction. See [\xintPIrr](#).

```
\xinttheexpr preduce(10e3/2), reduce(10e3/2)\relax
```

```
5[3], 5000
```

abs(x) absolute value

sgn(x) sign. See also [\xintifsgnexpr](#).

New with 1.3e **inv(x)** inverse.

floor(x) floor function.

ceil(x) ceil function.

sqr(x) square.

New with 1.3e **ilog10(x)** in [\xintiexpr](#) the integer exponent *a* such that $10^a \leq \text{abs}(x) < 10^{a+1}$; returns (this may evolve in future) **-2147450880** if *x* vanishes (i.e. **0x7fff8000**).

```
\xintiieval{ilog10(1), ilog10(-1234567), ilog10(-123456789123456789), ilog10(2**31)}\par
```

```
0, 6, 17, 9
```

See [ilog10\(\)](#) for the behaviour in [\xintexpr](#)-essions.

sqrt(x) in [\xintiexpr](#), truncated square root; in [\xintexpr](#) or [\xintfloatexpr](#) this is the floating point square root, and there is an optional second argument for the precision. See [sqrt\(\)](#).

sqrtr(x) available *only* in [\xintiexpr](#), rounded square root.

factorial(x) factorial function (like the post-fix ! operator.) When used in [\xintexpr](#) or [\xintfloatexpr](#) there is an optional second argument. See [factorial\(\)](#).

?(x) is the truth value, 1 if non zero, 0 if zero. Must use parentheses.

!(x) is logical not, 0 if non zero, 1 if zero. Must use parentheses.

not(x) logical not.

even(x) is the evenness of the truncation [num\(x\)](#).

```
\xintfloatexpr [3] seq((x,even(x)), x=-5/2..[1/3]..+5/2)\relax
-2.50, 1.00, -2.17, 1.00, -1.83, 0., -1.50, 0., -1.17, 0., -0.833, 1.00, -0.500, 1.00, -0.167,
1.00, 0.167, 1.00, 0.500, 1.00, 0.833, 1.00, 1.17, 0., 1.50, 0., 1.83, 0., 2.17, 1.00, 2.50,
1.00
```

odd(x) is the oddness of the truncation [num\(x\)](#).

```
\xintfloatexpr [3] seq((x,odd(x)), x=-5/2..[1/3]..+5/2)\relax
-2.50, 0., -2.17, 0., -1.83, 1.00, -1.50, 1.00, -1.17, 1.00, -0.833, 0., -0.500, 0., -0.167,
0., 0.167, 0., 0.500, 0., 0.833, 0., 1.17, 1.00, 1.50, 1.00, 1.83, 1.00, 2.17, 0., 2.50, 0.
```

New with 1.3d **isint(x)** evaluates to 1 if *x* is an integer, to 0 if not. See [ifint\(\)](#).

```

\xinttheexpr -5/3..[1/3]..+5/3\relax
\rightarrow \xinttheexpr seq(isint(x), x=-5/3..[1/3]..+5/3)\relax$
-5/3, -4/3, -3/3, -2/3, -1/3, 0, 1/3, 2/3, 3/3, 4/3, 5/3 → 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0
```

New with 1.3d **isone(x)** evaluates to 1 if *x* is 1, to 0 if not. See [ifone\(\)](#).

```

\xintfloatexpr subs(((x-1)/x, x/x, (x+1)/x), x=2**30)\relax
\rightarrow
\xintfloatexpr seq(isone(y), y=subs(((x-1)/x, x/x, (x+1)/x), x=2**30))\relax$
0.9999999990686774, 1.000000000000000, 1.000000000931323 → 0., 1.000000000000000, 0.
```

qint(x) belong with [qfrac\(\)](#), [qfloat\(\)](#), [qraw\(\)](#) to a special category:

1. They require the closing parenthesis of their argument to be immediately visible, it can not arise from expansion.
2. They grab the argument and store it directly; the format must be compatible with what is expected at macro level.
3. And in particular the argument can not be a variable, it has to be numerical.

[qint\(\)](#) achieves the same result as [num](#), but the argument is grabbed as a whole without expansion and handed over to the [\xintiNum](#) macro. The *q* stands for ``quick'', and [qint](#) is thought out for use in [\xintiexpr... \relax](#) with integers having dozens of digits.

Testing showed that using [qint\(\)](#) starts getting advantageous for inputs having more (or *f-expanding* to more) than circa 20 explicit digits. But for hundreds of digits the input gain becomes a negligible proportion of (for example) the cost of a multiplication.

Leading signs and then zeroes will be handled appropriately but spaces will not be systematically stripped. They should cause no harm and will be removed as soon as the number is used with one of the basic operators. This input mode *does not accept decimal part or scientific part*.

```
\def\x{...many many many ... digits}\def\y{...also many many many digits...}
```

```
\xinttheiexpr qint(\x)*qint(\y)+qint(\y)^2\relax\par
```

qfrac(x) does the same as **qint** excepts that it accepts fractions, decimal numbers, scientific numbers as they are understood by the macros of package **xintfrac**. Thus, it is for use in `\xintexpr.\relax`. It is not usable within an `\xintiexpr`-ession, except if hidden inside functions such as **round** or **trunc** which then produce integers acceptable to the integer-only parser. It has nothing to do with **frac** (sigh...).

qfloat(x) does the same as **qfrac** and then converts to a float with the precision given by the setting of `\xintDigits`. This can be used in `\xintexpr` to round a fraction as a float with the same result as with the `float()` function (whereas using `\xintfloatexpr A/B\relax` inside `\xintexpr.\relax` would first round *A* and *B* to the target precision); or it can be used inside `\xintfloatexpr.\relax` as a faster alternative to wrapping the fraction in a sub-`\xintexpr`-ession. For example, the next two computations done with 16 digits of precision do not give the same result:

```
\xintthefloatexpr qfloat(12345678123456785001/12345678123456784999)-0.5\relax\newline
\xintthefloatexpr 12345678123456785001/12345678123456784999-0.5\relax\newline
\xintthefloatexpr 1234567812345679/1234567812345678-0.5\relax\newline
\xintthefloatexpr \xintexpr12345678123456785001/12345678123456784999\relax-0.5\newline
```

```
0.5000000000000000
0.5000000000000010
0.5000000000000010
0.5000000000000000
```

because the second is equivalent to the third, whereas the first one is equivalent to the fourth one. Equivalently one can use **qfrac** to the same effect (the subtraction provoking the rounding of its two arguments before further processing.)

Note that if the input needs no special rounding, the internal form of the output keeps a short mantissa (it does not add padding zeros to make it of length equal to the float precision). For example `qfloat(2[20])` would keep internally the input format.

Changed at 1.3e!

2.3.3 Functions with an alphabetical argument

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

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

will return NO if executed in math mode (the computation is then $100-100=0$) and YES if not (the `if()` conditional is described below; the `\xintifboolexpr` test automatically encapsulates its first argument in an `\xintexpr` and follows the first branch if the result is non-zero (see subsection 10.13)).

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

togl(name) returns 1 if the \LaTeX package `etoolbox`¹⁶ has been used to define a toggle named *name*, and this toggle is currently set to `true`. Using `togl` in an `\xintexpr.\relax` without having loaded `etoolbox` will result in an error from `\iftoggle` being a non-defined macro. If `etoolbox` is loaded but `togl` is used on a name not recognized by `etoolbox` the error message will be of the type `ERROR: Missing \endcsname inserted.'`, with further information saying that `\protect` should have not been encountered (this `\protect` comes from the expansion of the non-expandable `etoolbox` error message).

¹⁶ <http://www.ctan.org/pkg/etoolbox>

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

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

A straight `\ifsometest{YES}{NO}` would do the same more efficiently, the point of `\ifsometest10` is to allow arbitrary boolean combinations using the (described later) `&&` and `||` logic operators: `\ifsometest10 && \ifsomeothertest10 || \ifsomeotherthertest10`, etc... `YES` or `NO` above stand for material compatible with the `\xintexpr` parser syntax.

See also `\xintifboolexpr`, in this context.

2.3.4 Functions with one mandatory and a second but optional argument

round(x[, n]) Rounds its first argument to a fixed point number, having a number of digits after decimal mark given by the second argument. For example `round(-2^9/3^5,12)=-2.106995884774`.

trunc(x[, n]) Truncates its first argument to a fixed point number, having a number of digits after decimal mark given by the second argument. For example `trunc(-2^9/3^5,12)=-2.106995884773`.

float(x[, n]) Rounds its first argument to a floating point number, with a precision given by the second argument. `float(-2^9/3^5,12)=-210699588477[-11]`.

Note for this example and the earlier ones that when the surrounding parser is `\xintfloatexpr`..`\relax` the fraction first argument (here `2^9/3^5`) will already have been computed as floating point number (with numerator and denominator handled separately first), even before the second argument is seen and a fortiori before the `round`, `trunc` or `float` is executed. The general float precision is the one governing these initial steps. To avoid that, use `\xintexpr2^9/3^5\relax` wrapper. Then the rounding or truncation will be applied on the exact fraction.

New with 1.3e **sfloat(x[, n])** It is the same as `float()`, but in case of a short (non-fractional) input it gets stored internally without adding zeros to make the mantissa have the `\xinttheDigits` length. One may wonder then what is the utility of `sfloat()`? See for an example of use the documentation of `\xintdefunc`. Notice however that this is a bit experimental and may evolve in future when `xint` gets a proper internal data structure for floating point numbers. The non-normalized format is useful for multiplication or division, but float additions and subtractions usually convert their arguments to a normalized mantissa.

New with 1.3e **ilog10(x[, n])** If there is an optional argument `n`, returns the (relative) integer `a` such that $10^a \leq \text{abs}(\text{float}(x, n)) < 10^{a+1}$. In absence of the optional argument:

- in `\xintexpr`, it returns the exponent `a` such that $10^a \leq \text{abs}(x) < 10^{a+1}$.
- in `\xintfloatexpr`, the input is first rounded to `\xinttheDigits` float precision, then the exponent `a` is evaluated.

```
\xintfloateval{ilog10(99999999/100000000, 8), ilog10(-999999995/1000000000, 8),
ilog10(-999999995/1000000000, 9)}\newline
\xinteval{ilog10(-999999995/1000000000), ilog10(-999999995/1000000000, 8)}
```

```
0., 1.0000000000000000, 0.
0, 1
```


If the input vanishes the function outputs `-2147450880` (i.e. `-0x7fff8000` which is near the minimal TeX number `-0x7fffffff`). This is also subject to change.

The `integer-only` variant for `\xintiexpr` admits no optional argument.

`sqrt(x[, n])` in `\xintexpr...\relax` and `\xintfloatexpr...\relax` it achieves the precision given by the optional second argument. For legacy reasons the `sqrt` function in `\xintiexpr` truncates (to an integer), whereas `sqrt` in `\xintfloatexpr...\relax` (and in `\xintexpr...\relax` which borrows it) rounds (in the sense of floating numbers). There is `sqrtr` in `\xintiexpr` for rounding to nearest integer.

```
\xinttheexpr sqrt(2,31)\relax\ and \xinttheiexpr sqrt(num(2e60))\relax
```

1414213562373095048801688724210[-30] and 1414213562373095048801688724209

There is an `integer only` variant for `\xintiexpr`.

`factorial(x[, n])` when the second optional argument is made use of inside `\xintexpr...\relax`, this switches to the use of the float version, rather than the exact one.

```
\xinttheexpr factorial (100,32)\relax, {\xintDigits:=32;\xintthefloatexpr
factorial (100)\relax}\newline
\xinttheexpr factorial (50)\relax\newline
\xinttheexpr factorial (50, 32)\relax
```

93326215443944152681699238856267[126], 9.3326215443944152681699238856267e157
 30414093201713378043612608166064768844377641568960512000000000000
 30414093201713378043612608166065[33]

The `integer only variant` of course has no optional second argument.

`randrange(A[, B])` when used with a single argument `A` returns a random integer $0 \leq x < A$, and when used with two arguments `A` and `B` returns a random integer $A \leq x < B$. As in Python it is an «empty range» error in first case if `A` is zero or negative and in second case if `B` \leq `A`.

New with
1.3b

The function can be used in all three parsers. Of course the size is not limited (but in the float parser, the integer will be rounded if involved in any operation).

```
\pdfsetrandomseed 12345
\xinttheiexpr randrange(10**20)\relax\newline
\xinttheiexpr randrange(1234*10**16, 1235*10**16)\relax\newline
\printnumber{\xinttheiexpr randrange(10**199,10**200)\relax}\par
```

12545314555479298502
 12341249468233524155
 38724271496566552250944896366777081662436330824968873373120332258200044549497099786643312
 91066875417168619069127435402274480091654618260723831075324716693356452348835689913277652
 395258486352999399662728

For the support macros see `\xintRandomDigits`, `\xintiiRandRange`, `\xintiiRandRangeAtoB`. For some details regarding how `xint` uses the engine provided generator of pseudo-random numbers, see `\xintUniformDeviate`.

2.3.5 Functions with two arguments

`quo(f, g)` first truncates the arguments to convert them to integers then computes the Euclidean quotient. Hence it computes an integer.

`rem(f, g)` first truncates the arguments to convert them to integers then computes the Euclidean remainder. Hence it computes an integer.

mod(f, g) computes $f - g \cdot \text{floor}(f/g)$. Hence its output is a general fraction or floating point number or integer depending on the parser where it is used.

Prior to 1.2p it computed $f - g \cdot \text{trunc}(f/g)$.

The `/:` and `'mod'` infix operators are both mapped to the same underlying macro as this `mod(f, g)` function. At 1.3 this macro produces smaller denominators when handling fractions than formerly.

Changed at 1.3!

```
\xinttheexpr mod(11/7,1/13), reduce(((11/7)/(1/13))*1/13+mod(11/7,1/13)),
mod(11/7,1/13)- (11/7)/:(1/13), (11/7)/(1/13)\relax\newline
\xintthefloatexpr mod(11/7,1/13)\relax\par
```

3/91, 11/7, 0, 20
0.03296703296703260

Attention! the precedence rules mean that $29/5 /: 3/5$ is handled like $((29/5)/:3)/5$. This is coherent with behaviour of Python language for example:

```
>>> 29/5 % 3/5, 11/3 % 17/19, 11/57
(0.5599999999999999, 0.19298245614035087, 0.19298245614035087)
>>> (29/5) % (3/5), (11/3) % (17/19), 5/57
(0.4, 0.08771929824561386, 0.08771929824561403)
```

For comparison (observe on the last lines how `\xintfloatexpr` is more accurate than Python!):

```
\noindent\xinttheexpr 29/5 /: 3/5, 11/3 /: 17/19\relax\newline
\xinttheexpr (29/5) /: (3/5), (11/3) /: (17/19)\relax\newline
\xintthefloatexpr 29/5 /: 3/5, 11/3 /: 17/19, 11/57\relax\newline
\xintthefloatexpr (29/5) /: (3/5), (11/3) /: (17/19), 5/57\relax\newline
5/57 = \xinttheexpr trunc(5/57, 20)\relax\dots\newline
```

14/25, 11/57
2/5, 5/57
0.5600000000000000, 0.1929824561403509, 0.1929824561403509
0.4000000000000000, 0.08771929824561420, 0.08771929824561404
5/57 = 0.08771929824561403508...

Regarding some details of behaviour in `\xintfloatexpr`, see discussion of `divmod` function next.

divmod(f, g) computes the two mathematical values `floor(f/g)` and `mod(f,g)=f - g*floor(f/g)` and produces them separated with a comma, in other terms it is analogous to the Python `divmod` function. Its output is equivalent to using `f//g`, `f/:g` but its implementation avoids doing twice the needed division.

In `\xintfloatexpr... \relax` the modulo is rounded to the prevailing precision. The quotient is like in the other parsers an exact integer. It will be rounded as soon as it is used in further operations, or via the global output routine of `\xintfloatexpr`.

```
\xintdefvar Q, R := divmod(3.7, 1.2);%
\xinttheexpr Q, R, 1.2Q + R\relax\newline
\xintdefiivar Q, R := divmod(100, 17);%
\xinttheiexpr Q, R, 17Q + R\relax\newline
\xintdeffloatvar Q, R := divmod(100, 17e-20);%
\xintthefloatexpr Q, R, 17e-20 * Q + R\relax\newline
% show Q exactly, although defined as float it can be used in iexpr:
\xinttheiexpr Q\relax\ (we see it has more than 16 digits)\par
\xintunassignvar{Q}\xintunassignvar{R}%
```

3, 1[-1], 37[-1]
5, 15, 100
5.882352941176471e20, 9.000000000000000e-20, 100.0000000000000
588235294117647058823 (we see it has more than 16 digits)



Again: f/g or the first item output by `divmod(f, g)` is an integer q which when computed inside `\xintfloatexpr.\relax` is not yet rounded to the prevailing float precision; the second item $f-q*g$ is the rounding to float precision of the exact mathematical value evaluated with this exact q . *This behaviour may change in future major release; perhaps q will be rounded and $f-q*g$ will correspond to usage of this rounded q .*

As `\xintfloatexpr` rounds its global result, or rounds operands at each arithmetic operation, it requires special circumstances to show that the q is produced unrounded. Either as in the above example or this one with comparison operators:

```
\xintDigits := 4;%
\xintthefloatexpr if(12345678//23=537000, 1, 0), 12345678//23\relax\newline
\xintthefloatexpr if(float(12345678//23)=537000, 1, 0)\relax\par
\xintDigits := 16;%
```

```
0., 537000.
1.000
```

In the first line, the comparison is done with `floor(12350000/23)=536957` (notice in passing that `12345678//23` was evaluated as `12350000//23` because the operands are first rounded to prevailing precision), hence the conditional takes the "False" branch. In the second line the `float` forces rounding of the output to 4 digits, and the conditional takes the "True" branch.

This example shows also that comparison operators in `\xintfloatexpr.\relax` act on unrounded operands.

binomial(x, y) computes binomial coefficients. It returns zero if $y < 0$ or $x < y$ and raises an error if $x < 0$ (or if $x > 99999999$.)

```
\xinttheexpr seq(binomial(20, i), i=0..20)\relax
```

```
1, 20, 190, 1140, 4845, 15504, 38760, 77520, 125970, 167960, 184756, 167960, 125970, 77520,
38760, 15504, 4845, 1140, 190, 20, 1
```

```
\printnumber{\xintthefloatexpr seq(binomial(100, 50+i), i=-5..+5)\relax}%
```

```
6.144847121413618e28, 7.347099819081500e28, 8.441348728306404e28, 9.320655887504988e28, 9.2
.891308288780803e28, 1.008913445455642e29, 9.891308288780803e28, 9.320655887504988e28, 8.2
441348728306404e28, 7.347099819081500e28, 6.144847121413618e28
```

The arguments must be (expand to) short integers.

pfactorial(a, b) computes partial factorials i.e. `pfactorial(a,b)` evaluates the product $(a+1) \dots b$.

```
\xinttheexpr seq(pfactorial(20, i), i=20..30)\relax
```

```
1, 21, 462, 10626, 255024, 6375600, 165765600, 4475671200, 125318793600, 3634245014400,
109027350432000
```

The arguments must (expand to) short integers. See [subsection 8.38](#) for the behaviour if the arguments are negative.

2.3.6 Functions with 3 or 4 arguments

if(cond,yes,no) (twofold-way conditional)

checks if `cond` is true or false and takes the corresponding branch. Any non zero number or fraction is logical true. The zero value is logical false. Both ``branches'' are evaluated (they are not really branches but just numbers). See also the `?` operator.

ifint(x,yes,no) (twofold-way conditional)

checks if x is an integer and in that case chooses the ``yes'' branch. See also `isint()`.

New with
1.3a

ifone(x,yes,no) (twofold-way conditional)

New with 1.3a

checks if `x` is equal to one and in that case chooses the `yes` branch. Slightly more efficient than `if(x=1,...)`. See also `isone()`.

ifsgn(cond,<0,=0,>0) (threefold-way conditional)

checks the sign of `cond` and proceeds correspondingly. All three are evaluated. See also the `??` operator.

2.3.7 Functions with an arbitrary number of arguments

Except for `qraw()`, this argument may be generated by one or many `a..b` or `a..[d]..b` constructs, separated by commas.

New with 1.3a

qraw(a,b,c,...,z) is provided for comma separated values. The input must obide by the suitable format depending on the parser: strict integers, or raw fractions, or floats in internal (non-documented) notation. Also, avoid spaces around the commas. The usefulness is when some loop generates hundreds of comma separated values. Without `qraw()`, each new value means a new usage of a `\csname..\endcsname` for storage of the growing list, with potential impact on \TeX memory (see [subsection 10.5](#)). See `qint()`, `qfrac()`, `qfloat()`, the difference being that `qraw()` does no post-processing at all of its input, apart from complete expansion. This allows it to accept comma separated values, as the internal storage also uses commas.

all(x, y, ...) inserts a logical **AND** in-between its arguments and evaluates the resulting logical assertion (as for all functions, all arguments are evaluated, see the `?` operator for `lazy` conditional branching; an example is to be found in [subsection 5.3](#).)

any(x, y, ...) inserts a logical **OR** in-between its arguments and evaluates the resulting logical assertion,

xor(x, y, ...) inserts a logical **XOR** in-between its arguments and evaluates the resulting logical assertion,

`+(x, y, ...) adds (left ticks mandatory):

```
\xinttheexpr `+(1,3,19), `+(1*2,3*4,19*20)\relax
```

23, 394

`*(x, y, ...) multiplies (left ticks mandatory):

```
\xinttheexpr `*(1,3,19), `*(1^2,3^2,19^2), `*(1*2,3*4,19*20)\relax
```

57, 3249, 9120

max(x, y, ...) maximum of the (arbitrarily many) arguments,

min(x, y, ...) minimum of the (arbitrarily many) arguments,

Changed at 1.3d!

gcd(x, y, ...) computes the positive generator of the fractional ideal of rational numbers $x\mathbb{Z} + y\mathbb{Z} + \dots \subset \mathbb{Q}$. When the inputs are integers it is advantageous to use a sub `\xintiexpr`-ession, as the integer-only macro is more efficient than the one accepting general fractional inputs. Notice that this may require some `num()` wrapper when using variables, as they may well be in fraction format, and `\xintiexpr` accepts only strict integers. Since 1.3d, this function and `lcm()` are available whether or not package `xintgcd` is loaded. Note that like other operations with fractions it does not always produce a fraction in irreducible format. This example shows also how to reduce an n-uple to its primitive part:

```
\xinttheexpr gcd(7/300, 11/150, 13/60)\relax\newline
```

```
$(7/300, 11/150, 13/60)\to
```

```
(\xinttheexpr seq(reduce(x), x = [7/300, 11/150, 13/60]/gcd(7/300, 11/150, 13/60))\relax)$
```

1/300
 (7/300, 11/150, 13/60) → (7, 22, 65)

Perhaps a future release will provide a `primpart()` function as built-in functionality.

Changed
 at 1.3d!

lcm(x, y, ...) computes the positive generator of the fractional ideal of rational numbers $x\mathbb{Z} \cap y\mathbb{Z} \cap \dots \subset \mathbb{Q}$. When the inputs are integers it is advantageous to use a sub `\xintiiexpr`-expression, as the integer-only macro is more efficient than the one accepting general fractional inputs.

```
\xinttheexpr lcm(7/300, 11/150, 13/60)\relax
```

1001/30

first(x, y, ...) first item of the list argument:

```
\xinttheiexpr first(last(-7..3), 58, 97..105)\relax
```

3

last(x, y, ...) last item of the list argument:

```
\xinttheiexpr last(-7..3, 58, first(97..105))\relax
```

97

reversed(x, y, ...) reverses the order of the comma separated list:

```
\xinttheiexpr first(reversed(123..150)), last(reversed(123..150))\relax
```

150, 123

len(x, y, ...) computes the number of items in a comma separated list. Earlier syntax was `[a,b,...,z][0]` but since 1.2g this now returns the first element of the list.

```
\xinttheiexpr len(1..50, 101..150, 1001..1050)\relax
```

150

2.3.8 Functions requiring dummy variables

The ``functions'' `add()`, `mul()`, `seq()`, `subs()`, `rseq()`, `iter()`, `rrseq()`, `iterr()` use delimited macros to identify the ``<letter>=''' part.¹⁷ This is done in a way allowing nesting via correctly balanced parentheses. The <letter> must not have been assigned a value before via `\xintdefvar`.

This ,<letter>= must be visible when the parser has finished absorbing the function name and the opening parenthesis. For `rseq()`, `iter()`, `rrseq()` and `iterr()` this is delayed to after the parser has assimilated a starting part delimited by a semi-colon; this mandatory segment may be generated entirely by expansion and the ,<letter>= may appear during this expansion.

After ,<letter>=, the expansion and parsing will generate a list of values (for example from an `a..b` specification, there may be multiple ones themselves separated by commas). After this step is complete the parser will know the values which will be assigned to <letter>. The special `<letter>=<integer>++` syntax offers a variant not pre-computing the iterated over list (which currently must thus proceed by steps of one.)

`seq()`, `rseq()`, `iter()`, `rrseq()`, `iterr()` but not `add()`, `mul()`, `subs()` admit the `omit`, `abort`, and `break()` keywords. In the case of a potentially infinite list generated by the `<integer>++` syntax, use of `abort` or of `break()` is mandatory, naturally.

Dummy variables are necessarily single-character letters, and all lowercase and uppercase Latin letters are pre-configured for that usage.

subs(expr, letter=values) for variable substitution

```
\xinttheexpr subs(subs(seq(x*z,x=1..10),z=y^2),y=10)\relax\newline
```

¹⁷ In the current implementation any token can be used rather than a =. What is looked for is a comma followed by two tokens, the first one will be the <letter>.

100, 200, 300, 400, 500, 600, 700, 800, 900, 1000

Attention that `xz` generates an error, one must use explicitly `x*z`, else the parser expects a variable with name `xz`.

`subs` is useful when defining macros for which some argument will be used more than once but may itself be a complicated expression or macro, and should be evaluated only once, for matters of efficiency.

The substituted variable may be a comma separated list (this is impossible with `seq` which will always pick one item after the other from a list).

```
\xinttheexpr subs([x]^2,x=-123,17,32)\relax
```

15129, 289, 1024

See the examples related to the `3x3` determinant in the [subsection 10.15](#) for an illustration of list substitution.

add(expr, letter=values) addition

```
\xinttheiexpr add(x^3,x=1..50), add(x(x+1), x=1,3,19)\relax\newline
```

1625625, 394

See ``+`` for syntax without a dummy variable.

mul(expr, letter=values) multiplication

```
\xinttheiexpr mul(x^2, x=1,3,19), mul(2n+1,n=1..10)\relax\newline
```

3249, 13749310575

See ``*`` for syntax without a dummy variable.

seq(expr, letter=values) comma separated values generated according to a formula

```
\xinttheiexpr seq(x(x+1)(x+2)(x+3),x=1..10), `(seq(3x+2,x=1..10))\relax
```

24, 120, 360, 840, 1680, 3024, 5040, 7920, 11880, 17160, 1162274713600

```
\xinttheiexpr seq(seq(i^2+j^2, i=0..j), j=0..10)\relax
```

0, 1, 2, 4, 5, 8, 9, 10, 13, 18, 16, 17, 20, 25, 32, 25, 26, 29, 34, 41, 50, 36, 37, 40, 45, 52, 61, 72, 49, 50, 53, 58, 65, 74, 85, 98, 64, 65, 68, 73, 80, 89, 100, 113, 128, 81, 82, 85, 90, 97, 106, 117, 130, 145, 162, 100, 101, 104, 109, 116, 125, 136, 149, 164, 181, 200

rseq(initial value; expr, letter=values) recursive sequence, `@` for the previous value.

```
\printnumber {\xintthefloatexpr subs(rseq(1; @/2+y/2@, i=1..10),y=1000)\relax } \newline
```

1.0000000000000000, 500.50000000000000, 251.2490009990010, 127.6145581634591, 67.72532736082604, 41.24542607499115, 32.74526934448864, 31.64201586865079, 31.62278245070105, 31.622772660168434, 31.62277660168379



Attention: in the example above `y/2@` is interpreted as `y/(2*@)`. With versions 1.2c or earlier it would have been interpreted as `(y/2)*@`.

In case the initial stretch is a comma separated list, `@` refers at the first iteration to the whole list. Use parentheses at each iteration to maintain this ``nuple'`. For example:

```
\printnumber{\xintthefloatexpr rseq(1,10^6;
(sqrt([@][0]*[@][1]),([@][0]+[@][1])/2), i=1..7)\relax }
```

1.0000000000000000, 1.0000000000000000e6, 1000.000000000000, 500000.5000000000, 22360.690952533499, 250500.2500000000, 74842.22521066670, 136430.4704776675, 101048.3052657827, 1056326.3478441671, 103316.8617608946, 103342.3265549749, 103329.5933734841, 103329.5941579348, 103329.5937657094, 103329.5937657095

iter(initial value; expr, letter=values) is exactly like `rseq`, except that it only prints the last iteration. Strangely it was lacking from 1.1 release, or rather what was available from 1.1 to 1.2f is what is called now `iterr()` (described below).

The new `iter()` is convenient to handle compactly higher order iterations. We can illustrate its use with an expandable (!) implementation of the Brent-Salamin algorithm for the computation of π :

```
\xintDigits:= 91;
\xintdeffloatfunc BS(a, b, t, p):= (a+b)/2, sqrt(a*b), t-p(a-b)^2, \xintiiexpr 2p\relax;
\xintthefloatexpr [88] % use 3 guard digits (output value is *rounded*)
  iter(1, 1/sqrt(2), 1, 1; % initial values
    ([@][0]-[@][1]<2[-45])? % if a-b is small enough stop iterating and ...
    {break(([@][0]+[@][1])^2/[@][2])} % ... do final computation,
    {BS(@)}, % else do iteration via pre-defined (for convenience) function BS.
    i=1++) % This generates infinite iteration. The i is not used.
\relax
\xintDigits:=16;%
```

3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628035
 You can try with `\xintDigits:=1001;` and `2[-501]` in place of `\xintDigits:=91;` and `2[-45]`, but don't make a final rounding to only 88 digits of course ... and better wrap the whole thing in `\message` or `\immediate\write128` because it will run in the right margin (about 7s on my laptop last time I tried). By the way here is how the BS function is defined internally:

```
Function BS for \xintfloatexpr parser associated to \XINT_flexpr_userfunc_B
S with meaning macro:#1#2#3#4->\XINTinFloatDiv {\XINTinFloatAdd {#1}{#2}}{2},\X
INTinFloatSqrtdigits {\XINTinFloatMul {#1}{#2}},\XINTinFloatSub {#3}{\XINTinFlo
atMul {#4}{\XINTinFloatPowerH {\XINTinFloatSub {#1}{#2}}{2}}},\xintiiMul {2}{#4
}
```

rrseq(initial values; expr, letter=values) recursive sequence with multiple initial terms. Say, there are K of them. Then `@1`, ..., `@4` and then `@@(n)` up to $n=K$ refer to the last K values. Notice the difference with `rseq` for which `@` refers to the complete list of all initial terms if there are more than one and may thus be a ``list'' object. This is impossible with `rrseq`. This construct is effective for scalar finite order recursions, and may be perhaps a bit more efficient than using the `rseq` syntax with a ``list'' value.

```
\xinttheiiexpr rrseq(0,1; @1+@2, i=2..30)\relax
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946,
17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040
\xinttheiiexpr rseq(1; 2@, i=1..10)\relax
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
\xinttheiiexpr rseq(1; 2@+1, i=1..10)\relax
1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047
\xinttheiiexpr rseq(2; @(@+1)/2, i=1..5)\relax
2, 3, 6, 21, 231, 26796
\xinttheiiexpr rrseq(0,1,2,3,4,5; @1+@2+@3+@4+@@(5)+@@(6), i=1..20)\relax
0, 1, 2, 3, 4, 5, 15, 30, 59, 116, 229, 454, 903, 1791, 3552, 7045, 13974, 27719, 54984, 109065,
216339, 429126, 851207, 1688440, 3349161, 6643338
```

I implemented an `Rseq` which at all times keeps the memory of *all* previous items, but decided to drop it as the package was becoming big.

iterr(initial values; expr, letter=values) same as `rrseq` but does not print any value until the last K .

```
\xinttheiiexpr iterr(0,1; @1+@2, i=2..5, 6..10)\relax
```



```
% the iterated over list is allowed to have disjoint defining parts.
```

34, 55

Recursions may be nested, with `@@@(n)` giving access to the values of the outer recursion. . . and there is even `@@@@(n)` to access the outer outer recursion but I never tried it!

The following keywords may be placed within the generating expression of a `seq()`, `rseq()`, `iter()`, `rrseq()`, or `iterr()`: :

abort stop here and now.

omit omit this value.

break `break(stuff)` to abort and have `stuff` as last value.

`<integer>++` serves to generate a potentially infinite list. In conjunction with an `abort` or `break()` this is often more efficient than iterating over a pre-established list of values.

```
\xinttheiexpr iter(1;(@>10^40)?{break(@)}{2@},i=1++)\relax
```

10889035741470030830827987437816582766592 is the smallest power of 2 with at least forty one digits.

The `i=<integer>++` syntax (any letter is allowed in place of `i`) works only in the form `<letter>=<integer>++`, something like `x=10,17,30++` is not legal. The `<integer>` must be a TeX-allowable integer.

```
First Fibonacci number at least |2^31| and its index
% we use iterr to refer via @1 and @2 to the previous and previous to previous.
\xinttheiexpr iterr(0,1; (@1>=2^31)?{break(i)}{@2+@1}, i=1++)\relax
```

First Fibonacci number at least 2^{31} and its index 2971215073, 47

Some additional examples are to be found in [subsection 2.5](#).

2.3.9 Trigonometrical functions

See `xinttrig`.

2.3.10 Logarithm, exponential and power functions

See `xintlog`.

2.4 Tacit multiplication

Tacit multiplication (insertion of a `*`) applies when the parser is currently either scanning the digits of a number (or its decimal part or scientific part, or hexadecimal input), or is looking for an infix operator, and:

- (1.) *encounters a count or dimen or skip register or variable or an ϵ -TeX expression, or*
- (2.) *encounters a sub-`\xintexpression`, or*
- (3.) *encounters an opening parenthesis, or*
- (4.) *encounters a letter (which is interpreted as signaling the start of either a variable or a function name), or*
- (5.) (of course, only when in state "looking for an operator") *encounters a digit.*

For example, if x , y , z are variables all three of $(x+y)z$, $x(y+z)$, $(x+y)(x+z)$ will create a tacit multiplication.



Furthermore starting with release 1.2e, whenever tacit multiplication is applied, in all cases it always ``ties'' more than normal multiplication or division, but still less than power. Thus $x/2y$ is interpreted as $x/(2y)$ and similarly for $x/2\max(3,5)$ but x^2y is still interpreted as $(x^2)*y$ and $2n!$ as $2*n!$.

```
\xintdefvar x:=30;\xintdefvar y:=5;%
\xinttheexpr (x+y)x, x/2y, x^2y, x!, 2x!, x/2max(x,y)\relax
1050, 30/10, 4500, 265252859812191058636308480000000, 530505719624382117272616960000000,
30/60
```

Since 1.2q tacit multiplication is triggered also in cases such as $(1+2)5$ or $10!20!30!$.

```
\xinttheexpr (10+7)5, 4!4!, add(i, i=1..10)10, max(x, y)100\relax
85, 576, 550, 3000
```

The ``tie more'' rule applies to all cases of tacit multiplication. It impacts only situations when a division was the last seen operator, as the normal rule for the `xintexpr` parsers is left-associativity in case of equal precedence.

```
\xinttheexpr 1/(3)5, (1+2)/(3+4)(5+6), 2/x(10), 2/10x, 3/y\xintiexpr 5+6\relax, 1/x(y)\relax\
differ from\newline\xinttheexpr 1/3*5, (1+2)/(3+4)*(5+6), 2/x*(10), 2/10*x,
3/y*\xintiexpr 5+6\relax, 1/x*(y)\relax\par
```

1/15, 3/77, 2/300, 2/300, 3/55, 1/150 differ from
5/3, 33/7, 20/30, 60/10, 33/5, 5/30

Note that `y\xinttheiexpr 5+6\relax` would have tried to use a variable with name `y11` rather than doing `y*11`: tacit multiplication works only in front of sub-`\xintexpressions`, not in front of `\xinttheexpressions` which are unlocked into explicit digits.

Here is an expression whose meaning is completely modified by the ``tie more'' property of tacit multiplication:

```
\xintdefunc e(z):=1+z(1+z/2(1+z/3(1+z/4)));
will be parsed as
\xintdefunc e(z):=1+z*(1+z/(2*(1+z/(3*(1+z/4))));
which is not at all the presumably hoped for:
\xintdefunc e(z):=1+z*(1+(z/2)*(1+(z/3)*(1+(z/4))));
```

2.5 More examples with dummy variables

These examples were first added to this manual at the time of the 1.1 release (2014/10/29).

```
Prime numbers are always cool
\xinttheiexpr seq((seq((subs((x/:m)?{(m*m>x)?{1}{0}}{-1},m=2n+1))
??{break(0)}{omit}{break(1)},n=1+))){x}{omit},
x=10001..[2]..10200)\relax
```

Prime numbers are always cool 10007, 10009, 10037, 10039, 10061, 10067, 10069, 10079, 10091, 10093, 10099, 10103, 10111, 10133, 10139, 10141, 10151, 10159, 10163, 10169, 10177, 10181, 10193

The syntax in this last example may look a bit involved (... and it is so I admit). First `x/:m` computes x modulo m (this is the modulo with respect to truncated division, which here for positive arguments is like Euclidean division; in `\xintexpr... \relax`, `a/:b` is such that $a = b*(a/b)+a/:b$, with `a/b` the algebraic quotient `a/b` truncated to an integer.). The `(x)?{yes}{no}` construct checks if `x` (which *must* be within parentheses) is true or false, i.e. non zero or zero. It then executes either the `yes` or the `no` branch, the non chosen branch is not evaluated. Thus if `m` divides `x` we are in the second (``false'') branch. This gives a `-1`. This `-1` is the argument to a `??` branch which is of the type `(y)?{y<0}{y=0}{y>0}`, thus here the `y<0`, i.e., `break(0)` is chosen. This `0` is thus given to another `?` which consequently chooses `omit`, hence the number is not kept in the list. The numbers which survive are the prime numbers.

```
The first Fibonacci number beyond |2^64| bound is
```

```
\xinttheiexpr subs(iterr(0,1;(@1>N)?{break(i)}{+@2},i=1++),N=2^64)\relax{}
```

and the previous number was its index.

The first Fibonacci number beyond 2^{64} bound is 19740274219868223167, 94 and the previous number was its index.

One more recursion:

```
\def\syr #1{\xinttheiexpr rseq(#1; (@<=1)?{break(i)}{odd(@)?{3@+1}{@//2}},i=0++)\relax}
```

The $3x+1$ problem: \syr{231}\par

The $3x+1$ problem: 231, 694, 347, 1042, 521, 1564, 782, 391, 1174, 587, 1762, 881, 2644, 1322, 661, 1984, 992, 496, 248, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1, 127

OK, a final one:

```
\def\syrMax #1{\xinttheiexpr iterr(#1,#1;even(i)?
    {(@<=1)?{break(i/2)}{odd(@)?{3@2+1}{@2//2}}
    {(@1>@2)?{+@2}},i=0++)\relax }
```

With initial value 1161, the maximal number attained is \syrMax{1161} and that latter number is the number of steps which was needed to reach 1.\par

With initial value 1161, the maximal number attained is 190996, 181 and that latter number is the number of steps which was needed to reach 1.

Look at the [Brent-Salamin algorithm implementation](#) for a more interesting recursion.

2.6 User defined variables

Since release 1.1 it is possible to make an assignment to a variable name and let it be known to the parsers of [xintexpr](#).

```
% definitions
\xintdefvar Pi:=3.141592653589793238462643;%
\xintdefvar x_1 := 10;\xintdefvar x_2 := 20;\xintdefvar y@3 := 30;%
\xintdefiivar List := seq(x(x+1)/2, x=0..10);%
% usage
$x_1\cdot x_2\cdot y@3+1=\xinttheiexpr x_1*x_2*y@3+1\relax$\newline
$\pi^{100}\approx\xintthefloatexpr Pi^{100}\relax$\newline
\xinttheiexpr List\relax\ contains \xinttheiexpr [List][7]\relax.\par
```

$x_1 \cdot x_2 \cdot y@3 + 1 = 6001$

$\pi^{100} \approx 5.187848314319574e49$

0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55 contains 28.

For catcodes issues (particularly, for the semi-colon used to delimit the fetched expression), see the discussion of [\xintexprSafeCatcodes](#).

Both syntaxes `\xintdefvar foo := <expr>;` and `\xintdefvar foo = <expr>;` are accepted.

Spaces in the variable name or around the equal sign are removed and are immaterial.

As shown above a variable can be assigned a "list" value. Simultaneous assignments are allowed:

```
\xintdefvar x1, x2, x3 := 3, 10^2, -1;%
\xintdefiivar A, B := 1500, 135;%
\xintloop
\xintifboolexpr{B}
  {\xintdefiivar A, B := B, A 'mod' B;\iftrue}
  {\iffalse}
\repeat
```

The last non zero remainder is \xinttheiexpr A\relax.

New with
1.3c

The last non zero remainder is 15.

The variable names are expanded in an `\edef` (and stripped of spaces). Example:

```
\xintdefvar x\xintListWithSep{, x}{\xintSeq{0}{10}} := seq(2**i, i = 0..10);%
```

This defines the variables `x0`, `x1`, . . . , `x10` for future usage.

Legal variable names are composed of letters, digits, `_` and `@` and characters. A variable name must start with a letter:

- the first character can not be a digit,
- and names starting with `@` or `_` are reserved (worldwide, extra-terrestrial locations included if under UNO supervision) by author for internal purposes.

`x_1x` is a licit variable name, as well as `x_1x_` and `x_1x_2` and `x_1x_2y` etc... hence tacit multiplication fails in cases like `x_1x_2` with `x_1` and `x_2` defined as variables; the parser goes not go to the effort of tracing back its steps, and it is too late when it realizes `x_1x_2` isn't a valid variable name. An explicit infix `*` operator is needed.

Single letter names `a..z` and `A..Z` are pre-declared by the package for use as special type of variables called ```dummy variables''`. It is allowed to overwrite their original meanings and assign them values. See further `\xintunassignvar`.

The assignments are done with `\xintdefvar`, `\xintdefiivar`, or `\xintdeffloatvar` and the variable value will be computed using respectively `\xintexpr`, `\xintiexpr` or `\xintfloatexpr`. It can then be used in all three parsers if the parser understands the format. Currently this means that variables using `\xintdefvar` or `\xintdeffloatvar` can not be used in the `\xintiexpr` parser, and variables defined via `\xintdefiivar` can be used in all parsers.

When defining a variable with `\xintdeffloatvar`, it is important to know that the rounding to `\xinttheDigits` digits of precision happens inside `\xintfloatexpr` only if an operation is executed. Thus, for a variable definition which uses no operations (and *only* for them), the value is recorded inside the variable with all its digits preserved. If `\xinttheDigits` changes afterwards, the variable will be rounded to that precision in force at time of use.

```
\xintdeffloatvar e:=2.7182818284590452353602874713526624977572470936999595749669676;%
\xinttheexpr e\relax\newline % shows the recorded value
\xintthefloatexpr e\relax\newline % output rounds
\xintthefloatexpr 1+e\relax\newline % the rounding was done by addition (trust me...)
\xintdeffloatvar e:=float(2.7182818284590452353602874713526624977572470936999595749669676);%
\xinttheexpr e\relax\par % use of float forced immediate rounding
```

27182818284590452353602874713526624977572470936999595749669676[-61]

2.718281828459045

3.718281828459045

2718281828459045[-15]

In the next examples we examine the effect of cumulated float operations on rounding errors:

```
\xintdefvar e_1:=add(1/i!, i=0..10);% exact sum
\xintdeffloatvar e_2:=add(1/i!, i=0..10);% float sum
\xintthefloatexpr e_1, e_2\relax\newline
\xintdefvar e_3:=e_1+add(1/i!, i=11..20);% exact sum
\xintdeffloatvar e_4:=e_2+add(1/i!, i=11..20);% float sum
\xintthefloatexpr e_3, e_4\relax\newline
\xintdeffloatvar e:=2.7182818284590452353602874713526624977572470936999595749669676;%
\xintDigits:=24;
\xintthefloatexpr[16] e, e^1000, e^1000000\relax (e rounded to 24 digits first)\newline
\xintDigits:=16;
\xintthefloatexpr e, e^1000, e^1000000\relax (e rounded to 16 digits first)\par
```

2.718281801146384, 2.718281801146385

2.718281828459045, 2.718281828459046

2.718281828459045, 1.970071114017047e434, 3.033215396802088e434294(e rounded to 24 digits first)

2.718281828459045, 1.970071114016876e434, 3.033215396539459e434294(e rounded to 16 digits first)

With `\xintverbosetrue` the values of the assigned variables will be written to the log. For example like this (the line numbers here are artificial):

```
Package xintexpr Info: (on line 2875)
```

```

Variable "e" defined with value 2718281828459045235360287471352662497757247
0936999595749669676[-61].
Package xintexpr Info: (on line 2879)
Variable "e" defined with value 2718281828459045[-15].
Package xintexpr Info: (on line 2886)
Variable "e_1" defined with value 9864101/3628800[0].
Package xintexpr Info: (on line 2887)
Variable "e_2" defined with value 2718281801146385[-15].
Package xintexpr Info: (on line 2889)
Variable "e_3" defined with value 6613313319248080001/2432902008176640000[0
].
Package xintexpr Info: (on line 2890)
Variable "e_4" defined with value 2718281828459046[-15].
Package xintexpr Info: (on line 2892)
Variable "e" defined with value 2718281828459045235360287471352662497757247
0936999595749669676[-61].

```

2.6.1 \xintunassignvar

Changed
at 1.3d!



Variable declarations obey the current scope. To let a (multi-letter) name be unknown to (all parsers of) `xintexpr` without waiting the end of the scope one issues `\xintunassignvar{<variable>}`. Prior to 1.3d, this only redefined the variable to represent the value 0.

In the special case of `\xintunassignvar{<letter>}`, the effect is different, as it is synonymous with `\xintnewdummy{<letter>}`: the (catcode 11) `<letter>` recovers or acquires meaning as a dummy variable in the current scope.

```

\xintFor #1 in {e_1, e_2, e_3, e_4, e} \do {\xintunassignvar {#1}}
% overwriting a dummy letter
\xintdefvar i := 3;%
\xinttheiexpr add(i, i = 1..10)\relax\ ("i" has the fixed value 3)\newline
\xintunassignvar{i}% back to normal
\xinttheiexpr add(i, i = 1..10)\relax\ ("i" is again a dummy variable)\par

```

30 ("i" has the fixed value 3)

55 ("i" is again a dummy variable)

Under `\xintglobaldefstrue` regime the effect of `\xintunassignvar` is global.

2.6.2 \xintnewdummy

Any catcode 11 character can serve as a dummy variable, via this declaration:

```
\xintnewdummy{<character>}
```

For example with XeTeX or LuaTeX the following works:

```

% use a Unicode engine
\input xintexpr.sty
\xintnewdummy ξ% or any other letter character !
\xinttheexpr add(ξ, ξ=1..10)\relax
\bye

```

Under `\xintglobaldefstrue` regime the effect of `\xintnewdummy` is global.

2.6.3 \xintensuredummy, \xintrestorelettervar

New with
1.3e

Use

```

\xintensuredummy{<character>}
...
... code using the (catcode 11) character as a dummy variable
...
\xintrestorelettervar{<character>}

```

if other parts need the letter as an assigned variable name. For example `xinttrig` being written at high level needs a few genuine dummy variables, and it uses `\xintensuredummy` to be certain everything is ok.

2.7 User defined functions

2.7.1	<code>\xintdeffunc</code>	35
2.7.2	<code>\xintdefiifunc</code>	37
2.7.3	<code>\xintdeffloatfunc</code>	37
2.7.4	Some examples of recursive definitions	37
2.7.5	<code>\xintdefefunc</code>	38
2.7.6	<code>\xintdeffloatefunc</code>	38
2.7.7	<code>\xintdefiiefunc</code>	38
2.7.8	Using the same name for both a variable and a function	38
2.7.9	<code>\xintunassignexprfunc</code> , <code>\xintunassigniexprfunc</code> , <code>\xintunassignfloatexprfunc</code>	38
2.7.10	<code>\ifxintverbose</code> conditional	38
2.7.11	<code>\ifxintglobaldefs</code> conditional	39
2.7.12	Functions expanding to comma separated values	39
2.7.13	Example with the RUMP test	40
2.7.14	<code>\xintNewFunction</code>	41

2.7.1 `\xintdeffunc`

Since release 1.2c it is possible to declare functions:

```
\xintdeffunc
  Rump(x,y):=1335 y^6/4 + x^2 (11 x^2 y^2 - y^6 - 121 y^4 - 2) + 11 y^8/2 + x/2y;
(notice the numerous tacit multiplications in this expression; and that  $x/2y$  is interpreted as  $x/(2y)$ .)
```

Here are a few important items (bookmark this for reading again later once you have gained experience in using this interface...):

- The function names are composed of letters, digits, underscores or @ signs. A function name must start with a letter. It may be a single letter (see [subsubsection 2.7.8](#)).
- The variables used in the function signature are single letters (lowercase or uppercase) which have not been re-declared via `\xintdefvar` as assigned variables. The choice of the letters is entirely up to the user and has nil influence on the actual function, naturally.
- A function can have at most nine variables.
- The mechanism for functions shares a common code base with the one implementing `\xintNewExpr`. This means it shares its features and also its [limitations](#).



Changed at 1.3e!

Most notably, the 1.3d `\xinteval`, `\xintieval`, `\xintfloateval` can not be used inside the parsed expression: only the lower level syntax `\xintexpr... \relax` et al. is accepted (and not `\xinttheexpr` et al. which are about the same as `\xinteval` et al.). Prior to 1.3e `\xintNewExpr` and `\xintdeffunc` diverged on that point, but their behaviour is now identical.

- In order to allow recursive constructs, a core mechanism is implemented which inhibits immediate expansion in a new definition; think of `\xintdeffunc` as being analogous to a `\protected \edef`. This means that another function `bar(x,...)` whose definition uses `foo(17.5)` will only store that it should at some point compute `foo(17.5)`, in place of storing its actual value.
- If `foo(x)` definition is not recursive, then you should use `\xintdefefunc` rather. This is analogous to an `\edef` without the `\protected`. Then `bar(x,...)` (defined with `\xintdeffunc` or `\xintdefefunc`) will store the actual evaluation of `foo(17.5)`.

New with 1.3e


```
\xintdeffunc f(x):=add(i^2,i=1..x);
```

is not currently possible. See [subsection 10.15.3](#) and the next subsection.

On the other hand a construct like the following has no issue, as the values iterated over do not depend upon the function parameters:

```
\xintdeffunc f(x):=iter(1{;} @*x/i+1, i=10..1);% one must hide the first semi-colon !
\xinttheexpr e(1), f(1)\relax
```

9864101/3628800, 9864101/3628800

Another problem is with trying to do $g(f(x))$ where $g()$ expects two arguments and $f()$ was defined to output two comma separated values. This works fine numerically but not with a variable x inside the definition of another function.

See [subsection 2.7.12](#) for more about comma separated values in output and input.

2.7.2 \xintdefiifunc

With `\xintdeffunc` the created function is known by the `\xintexpr` parser only. Cryptic error messages will signal failures of using with another parser a function declared for one parser (particularly if the name is a single letter, because the parser will have made an attempt to use the letter as a dummy variable.)

For usage in the `\xintiexpr` parser, it is required to use `\xintdefiifunc`.

2.7.3 \xintdeffloatfunc

With `\xintdeffunc` the created function is known by the `\xintexpr` parser only. For usage in the `\xintfloatexpr` parser, it is required to use `\xintdeffloatfunc`. See `\xintdeffunc` for more information on this point.

2.7.4 Some examples of recursive definitions

Since 1.3, it is possible to make recursive definitions. Here are two examples:

```
\xintdeffunc GCD(a,b):=if(b,GCD(b,a/:b),a);
```

This of course is the Euclidean algorithm: it will be here applied to variables which may be fractions. For example:

```
\xinttheexpr GCD(385/102, 605/238)\relax
```

55/714

But there is already a built-in `gcd()` (which accepts arbitrarily many arguments):

```
\xinttheexpr gcd(385/102, 605/238)\relax
```

55/714

Since 1.3d the built-in `gcd()` accepts inputs being fractions and produces the positive generator of the corresponding fractional ideal. And loading of `xintgcd` is not needed for this function to be available.

New with
1.3d

Our second example is modular exponentiation:

```
\xintdefiifunc powmod_a(x, m, n) :=
  ifone(m,
    % m=1, return x modulo n
    x /: n,
    % m > 1 test if odd or even and do recursive call
    if(odd(m), (x*sqr(powmod_a(x, m/2, n))) /: n,
      sqr(powmod_a(x, m/2, n)) /: n
    )
  );
\xintdefiifunc powmod(x, m, n) := if(m, powmod_a(x, m, n), 1);
```

I have made the definition here for the `\xintiexpr` parser; we could do the same for the `\xintexpr` parser (but its usage with big powers would quickly create big denominators, think `powmod(1/2, 21000, 1)` for example.)

```
\xinttheiexpr seq(powmod(x, 1000, 128), x=9, 11, 13, 15, 17, 19, 21)\relax\par
```

65, 97, 33, 1, 1, 33, 97

The function assumes the exponent is non-negative (the Python `pow` behaves the same), but zealous users will add the necessary code for negative exponents, after having defined another function for modular inverse!

It is mandatory for such definitions to use the `if()` function, and not the `(x)?{A}{B}` construct which much choose a branch. The parsing of the `if()` function keeps the memory of the two alternative branches; to the contrary, the *constructed* `powmod` function will expand *only* the then relevant branch. This is of course absolutely needed for things such as the Euclide algorithm where it would be catastrophic to evaluate both branches as the first one involves a division by `b` and the algorithm stops only when `b` is actually zero.

If function `A` needs function `B` which needs function `A` start by giving to `B` some dummy definition, define `A`, then define `B` properly. TODO: add some example here...

2.7.5 `\xintdefefunc`

New with 1.3e Think of former described variant `\xintdeffunc` as doing the same as this `\xintdefefunc` but with an extra protection added to the defined function. If you don't need recursivity, `\xintdefefunc` is the better tool, as numerical evaluation involving it and arising in further definitions will be converted on the spot to actual values, rather than being delayed for expansion to actual use of the defined function in `\xinteval` or `\xintieval`.

2.7.6 `\xintdeffloatefunc`

New with 1.3e The ```unprotected''` variant of `\xintdeffloatfunc`.

2.7.7 `\xintdefiiefunc`

New with 1.3e The ```unprotected''` variant of `\xintdefiifunc`.

2.7.8 Using the same name for both a variable and a function

It is licit to overload a variable name (all Latin letters are predefined as dummy variables) with a function name and vice versa. The parsers will decide from the context if the function or variable interpretation must be used (dropping various cases of tacit multiplication as normally applied).

```
\xintdefiifunc f(x):=x^3;
\xinttheiiepr add(f(f),f=100..120)\relax\newline
\xintdeffunc f(x,y):=x^2+y^2;
\xinttheexpr mul(f(f(f,f),f(f,f)),f=1..10)\relax
\xintunassigniieprfunc{f}\xintunassignexprfunc{f}%
```

28205100

186188134867578885427848806400000000

2.7.9 `\xintunassignexprfunc`, `\xintunassigniieprfunc`, `\xintunassignfloatexprfunc`

New with 1.3d Function names can be unassigned via `\xintunassignexprfunc{<name>}`, `\xintunassigniieprfunc{<name>}`, and `\xintunassignfloatexprfunc{<name>}`.

```
\xintunassignexprfunc{e}
\xintunassignexprfunc{f}
```

Warning: no check is done to avoid undefining built-in functions...

2.7.10 `\ifxintverbose conditional`

With `\xintverbosetrue` the meanings of the functions (or rather their associated macros) will be written to the log. For example the `Rump` declaration above generates this in the log file:

```
Function Rump for \xintexpr parser associated to \XINT_expr_userfunc_Rump w
ith meaning macro:#1#2->\xintAdd {\xintAdd {\xintAdd {\xintDiv {\xintMul {1335}
```

```
{\xintPow {#2}{6}}{4}}{\xintMul {\xintPow {#1}{2}}{\xintSub {\xintSub {\xintSu
b {\xintMul {11}{\xintMul {\xintPow {#1}{2}}{\xintPow {#2}{2}}}{\xintPow {#2}{
6}}}{\xintMul {121}{\xintPow {#2}{4}}}{2}}}{\xintDiv {\xintMul {11}{\xintPow
{#2}{8}}}{2}}}{\xintDiv {#1}{\xintMul {2}{#2}}}
```

and the declaration `\xintdeffunc f(x):=iter(1{;} @*x/i+1, i=10..1)`; generates:

```
Function f for \xintexpr parser associated to \XINT_expr_userfunc_f with me
aning macro:#1->\xintAdd {\xintDiv {\xintMul {\xintAdd {\xintDiv {\xintMul {\xi
ntAdd {\xintDiv {\xintMul {\xintAdd {\xintDiv {\xintMul {\xintAdd {\xintDiv {\xi
ntMul {\xintAdd {\xintDiv {\xintMul {\xintAdd {\xintDiv {\xintAdd {\xintDiv {\xi
ntDiv {\xintMul {\xintAdd {\xintDiv {\xintMul {\xintAdd {\xintDiv {\xintMul {
1}{#1}}{10/1[0]}}{1}}{#1}}{9/1[0]}}{1}}{#1}}{8/1[0]}}{1}}{#1}}{7/1[0]}}{1}}{#1
}}{6/1[0]}}{1}}{#1}}{5/1[0]}}{1}}{#1}}{4/1[0]}}{1}}{#1}}{3/1[0]}}{1}}{#1}}{2/1[0
]}}{1}}{#1}}{1/1[0]}}{1}}
```

Starting with 1.2d the definitions made by `\xintNewExpr` have local scope, hence this is also the case with the definitions made by `\xintdeffunc`. See also `\ifxintglobaldefs` conditional.

2.7.11 `\ifxintglobaldefs` conditional

If true user defined variables (`\xintdefvar`, ...) and functions (`\xintdeffunc`, ...) for the expression parsers, as well as macros obtained via `\xintNewExpr` et al have global scope. If false (default) they have local scope.

New with
1.3c

2.7.12 Functions expanding to comma separated values

It is possible to define functions which expand to comma-separated values, for example the declarations:

```
\xintdeffunc f(x):= x, x^2, x^3, x^x;
\xintdeffunc g(x):= x^[0..x];% x^[1, 2, 3, x] would be like f above.
```

will generate

```
Function f for \xintexpr parser associated to \XINT_expr_userfunc_f with me
aning macro:#1->#1,\xintPow {#1}{2},\xintPow {#1}{3},\xintPow {#1}{#1}
```

```
Function g for \xintexpr parser associated to \XINT_expr_userfunc_g with me
aning macro:#1->\xintApply::csv {\xintPow {#1}}{\xintSeq::csv {0}{#1}}
```

and we can check that they work:

```
\xinttheexpr f(10)\relax; \xinttheexpr g(10)\relax
10, 100, 1000, 10000000000; 1, 10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000,
1000000000, 10000000000
```

However please consider this as WIP. They are some known (or half-known, because the author gets a headache whenever he reconsiders the whole thing) thorny issues, mostly related to the fact that `xintexpr` has no proper variable type for lists. See [subsection 2.8](#).

Here is another aspect: the documentation of release 1.3c included this paragraph:

It is possible to define functions of variables which stand for lists (see [subsection 2.8](#)), or functions defining lists of comma separated items. For example the scalar product and cross product of 3-dimensional vectors can be defined this way:

```
\xintdeffunc dprod(V, W) := [V][0]*[W][0] + [V][1]*[W][1] + [V][2]*[W][2];
\xintdeffunc cprod(V, W) := [V][1]*[W][2] - [V][2]*[W][1],
[V][2]*[W][0] - [V][0]*[W][2],
[V][0]*[W][1] - [V][1]*[W][0];
\xintdeffunc Det3(U, V, W) := dprod(cprod(U, V), W);
```

But it should be added promptly that due to absence to typed variables of type list (see [subsection 2.8](#)), usage of the above is very subtle: although it is possible to define variables `U`, `V`, `W` expanding to three components each it is impossible to use them with `Det3(U, V, W)` because

`\xintexpr` will convert U, V, W to a comma separated list of 9 numbers, and `Det3` was defined as a function of only 3 things.

The only way (currently) is to define U, V, W as *functions* (possibly of no variable), then to define a new *function* $Z = \text{Det3}(U, V, W)$, and finally to evaluate Z with no argument:

```
\xintdeffunc V() := 1, 1, 1;
\xintdeffunc W() := 1, 5, 25;
\xintdeffunc Y() := 1, 10, 100;
\xintdeffunc Z() := Det3(V(), W(), Y());
% \xinteval{Det3(V(), W(), Y())} does NOT work, one must go via Z()
\xinteval{Z()}
```

180

It is better for pure numerics to define the `Det3()` initially as a function of 9 variables. But the above works well if one really wants to work with variables:

```
\xintdeffunc V(x) := 1, x, x^2;
\xintdeffunc Z(x,y,z) := Det3(V(x), V(y), V(z));
\xinteval{Z(1, 5, 10)}
```

180

This can be combined with usage of my other package [polexpr](#). I thank Thomas SÖLL who explored precisely that during 2018.

Changed
at 1.3e!

To tell the whole truth, until 1.3e the above worked *only* with at least one variable, the syntax with no variables had a bug.

Cleaning up:

```
\xintunassignexprfunc{g}
\xintunassignexprfunc{V}
\xintunassignexprfunc{W}
\xintunassignexprfunc{Y}
\xintunassignexprfunc{Z}
\xintunassignexprfunc{dprod}\xintunassignexprfunc{cprod}\xintunassignexprfunc{Det3}
```

2.7.13 Example with the Rump test

Let's try out our `Rump()` function:

```
\xinttheexpr Rump(77617,33096)\relax.
```

-54767/66192. Nothing problematic for an exact evaluation, naturally !

Thus to test the Rump polynomial (it is not quite a polynomial with its $x/2y$ final term) with floats, we *must* also declare `Rump` as a function to be used there:

```
\xintdeffloatfunc
Rump(x,y):=333.75 y^6 + x^2 (11 x^2 y^2 - y^6 - 121 y^4 - 2) + 5.5 y^8 + x/2y;
```

The numbers are scanned with the current precision, hence as here it is 16, they are scanned exactly in this case. We can then vary the precision for the evaluation.

```
\def\CR{\cr}
\halign
{\tabskip1ex
\hfil\bfseries#\xintDigits:=\xintloopindex;\xintthefloatexpr Rump(77617,33096)#\cr
\xintloop [8+1]
\xintloopindex &\relax\CR
\ifnum\xintloopindex<40 \repeat
}
```

8 7.0000000e29

9 -1.0000000e28

10 5.0000000e27

11 -3.0000000e26

12 4.0000000e25

13 3.0000000e24

14 3.0000000e23

15 -2.0000000e22

```

16 1.0000000000000000e21
17 -5.0000000000000000e20
18 1.17260394005317863
19 1.0000000000000000e18
20 -9.99999999999999827e16
21 1.00000000000000011726e16
22 3.0000000000000001172604e15
23 -9.999999999999827396060e13
24 -1.9999999999998273960599e13
25 -1.999999999999827396059947e12
26 1.1726039400531786318588349
27 -5.9999999999998273960599468214e10
28 -9.9999999999998273960599468213681e8
29 2.0000000117260394005317863186e8
30 1.00000011726039400531786318588e7
31 -999998.8273960599468213681411651
32 200001.17260394005317863185883490
33 -9998.82739605994682136814116509548
34 -1998.827396059946821368141165095480
35 -198.82739605994682136814116509547982
36 21.1726039400531786318588349045201837
37 -0.8273960599468213681411650954798162920
38 -0.82739605994682136814116509547981629200
39 -0.827396059946821368141165095479816292000
40 -0.8273960599468213681411650954798162919990

```

2.7.14 \xintNewFunction

The syntax is analogous to the one of `\xintNewExpr` but achieves something *completely different* from `\xintNewExpr/\xintdeffunc`. Here is an example:

```
\xintNewFunction {foo}[3]{add(mul(x+i, i=#1..#2),x=1..#3)}
```

We now have a genuine function `foo(, ,)` of three variables which we can use fully in *all three parsers*, be it with numerical arguments or variables or whatever.

```
\xinttheexpr seq(foo(0, 3, j), j= 1..10)\relax
```

24, 144, 504, 1344, 3024, 6048, 11088, 19008, 30888, 48048 See [subsection 5.3](#) for some additional examples.

This construct is only syntactic sugar to benefit from functional notation. Each time the created «function-macro» `foo()` will be encountered the corresponding expression will get inserted as a sub-expression (of the same type as the surrounding one), the macro parameters having been replaced with the (already evaluated) function arguments, and the parser *will then have to parse the expression*. It is very much like a macro substitution, but with parentheses and comma separated arguments (which can be arbitrary expressions themselves).

```
Function foo for the expression parsers is associated to \XINT_expr_macrofu
nc_foo with meaning macro:#1#2#3->add(mul(x+i, i=\XINT_expr_wrapit {#1}..\XINT_
expr_wrapit {#2}),x=1..\XINT_expr_wrapit {#3})
```

Thus, this works with quite arbitrary constructs, contrarily to the mechanism of `\xintdeffunc`. It is not currently possible to define a `foo` function like the one above via `\xintdeffunc`.¹⁸

One can declare a function `foo` with `[0]` arguments: it may be used as `foo()` or `foo(nil)` (prior to 1.3b only the latter was accepted).

Changed
at 1.3b!

¹⁸ Or rather, it turns out that no error is raised on making the definition via `\xintdeffunc` but the created supporting macro is only garbage and raises errors on use.

2.8 List operations

By *list* we hereby mean simply comma-separated values, for example `3, -7, 1e5`. This section describes some syntax which allows to manipulate such lists, for example `[3, -7, 1e5][1]` extracts `-7` (we follow the Python convention of enumerating starting at zero.)

In the context of dummy variables, lists can be used in substitutions:

```
\xinttheiexpr subs(`(L), L = 1, 3, 5, 7, 9)\relax\newline
```

25

and also the `rseq` and `iter` constructs allow `@` to refer to a list:

```
\xinttheiexpr iter(0, 1; ([@][1], [@][0]+[@][1]), i=1..10)\relax\newline
```

55, 89

where each step constructs a new list with two entries.

However, despite appearances there is not really internally a notion of a *list type* and it is currently impossible to create, manipulate, or return on output a *list of lists*. There is a special reserved variable `nil` which stands for the empty list.

The syntax which is explained next includes in particular what are called *list itemwise operators* such as:

```
\xinttheiexpr 37+[13,100,1000]\relax\newline
```

50, 137, 1037

This part of the syntax is considered provisory, for the reason that its presence might make more difficult some extensions in the future. On the other hand the Python-like slicing syntax should not change.

- `a..b` constructs the **small** integers from the ceil `[a]` to the floor `[b]` (possibly a decreasing sequence): one has to be careful if using this for algorithms that `1..0` for example is not empty or `1` but expands to `1, 0`. Again, `a..b` can not be used with `a` and `b` greater than $2^{31} - 1$. Also, only about at most **5000** integers can be generated (this depends upon some \TeX memory settings).

The `..` has lower precedence than the arithmetic operations.

```
\xinttheexpr 1.5+0.4..2.3+1.1\relax; \xinttheexpr 1.9..3.4\relax; \xinttheexpr 2..3\relax
```

2, 3; 2, 3; 2, 3

- `a..[d]..b` allows to generate big integers, or also fractions, it proceeds with step (non necessarily integral nor positive) `d`. It does *not* replace `a` by its ceil, nor `b` by its floor. The generated list is empty if `b-a` and `d` are of opposite signs; if `d=0` or if `a=b` the list expands to single element `a`.

```
\xinttheexpr 1.5..[1.01]..11.23\relax
```

15[-1], 251[-2], 352[-2], 453[-2], 554[-2], 655[-2], 756[-2], 857[-2], 958[-2], 1059[-2]

- `[list][n]` extracts the `n+1`th element if `n>=0`. If `n<0` it extracts from the tail. List items are numbered (since 1.2g) as in Python, the first element corresponding to `n=0`. `len(list)` computes the number of items of the list.

```
\xinttheiexpr \empty[0..10][6], len(0..10), [0..10][-1], [0..10][23*18-22*19]\relax\
(and 23*18-22*19 has value \the\numexpr 23*18-22*19\relax).
```

6, 11, 10, 7 (and $23*18-22*19$ has value -4).

See the next frame for why the example above has `\empty` token at start.

As shown, it is perfectly legal to do operations in the index parameter, which will be handled by the parser as everything else. The same remark applies to the next items.

- `[list][:n]` extracts the first `n` elements if `n>0`, or suppresses the last `|n|` elements if `n<0`.

```
\xinttheiexpr [0..10][:6]\relax\ and \xinttheiexpr [0..10][::-6]\relax
```

0, 1, 2, 3, 4, 5 and 0, 1, 2, 3, 4

- `[list][n:]` suppresses the first `n` elements if `n>0`, or extracts the last `|n|` elements if `n<0`.

```
\xinttheiexpr [0..10][6:]\relax\ and \xinttheiexpr [0..10][-6:]\relax
```

6, 7, 8, 9, 10 and 5, 6, 7, 8, 9, 10

- More generally, `[list][a:b]` works according to the Python ``slicing'' rules (inclusive of negative indices). Notice though that there is no optional third argument for the step, which always defaults to `+1`.

```
\xinttheiexpr [1..20][6:13]\relax\ = \xinttheiexpr [1..20][6-20:13-20]\relax
```

7, 8, 9, 10, 11, 12, 13 = 7, 8, 9, 10, 11, 12, 13

- It is naturally possible to nest these things:

```
\xinttheexpr [[1..50][13:37]][10:-10]\relax
```

24, 25, 26, 27

- itemwise operations either on the left or the right are possible:

```
\xinttheiexpr 123*[1..10]^2\relax
```

123, 492, 1107, 1968, 3075, 4428, 6027, 7872, 9963, 12300



List operations are implemented using square brackets, but the `\xintiexpr` and `\xintfloa` `atexpr` parsers also check to see if an optional parameter within brackets is specified before the start of the expression. To avoid the resulting confusion if this `[` actually serves to delimit comma separated values for list operations, one can either:

- insert something before the bracket such as `\empty` token,

```
\xinttheiexpr \empty [1,3,6,99,100,200][2:4]\relax
```

6, 99

- use parentheses:

```
\xinttheiexpr ([1,3,6,99,100,200][2:4])\relax
```

6, 99

Notice though that `([1,3,6,99,100,200])[2:4]` would not work: it is mandatory for `][` and `][:` not to be interspersed with parentheses. Spaces are perfectly legal:

```
\xinttheiexpr \empty[1..10 ] [ : 7 ]\relax
```

1, 2, 3, 4, 5, 6, 7

Similarly all the `+[, *[, ... and]**,]/, ...` operators admit spaces but nothing else between their constituent characters.

```
\xinttheiexpr \empty [ 1 . . 1 0 ] * * 1 1 \relax
```

1, 2048, 177147, 4194304, 48828125, 362797056, 1977326743, 8589934592, 31381059609, 100000000000

In an other vein, the parser will be confused by `1..[a,b,c][1]`, and one must write `1..([a,b,c][1])`. And things such as `[100,300,500,700][2]/11` or `[100,300,500,700][2]/11` are syntax errors and one must use parentheses, as in `([100,300,500,700][2])/11`.

2.9 Analogies and differences of `\xintiexpr` with `\numexpr`

`\xintiexpr`.`\relax` is a parser of expressions knowing only (big) integers. There are, besides the enlarged range of allowable inputs, some important differences of syntax between `\numexpr` and `\xintiexpr` and variants:

- Contrarily to `\numexpr`, the `\xintiexpr` parser will stop expanding only after having encountered (and swallowed) a mandatory `\relax` token.
- In particular, spaces between digits (and not only around infix operators or parentheses) do not stop `\xintiexpr`, contrarily to the situation with `numexpr`: `\the\numexpr 7 + 3 5\relax` expands (in one step)¹⁹ to `105\relax`, whereas `\xintthe\xintiexpr 7 + 3 5\relax` expands (in two steps) to `42`.²⁰
- Inside an `\edef`, an expression `\xintiexpr...relax` get fully evaluated, whereas `\numexpr` without `\the` or `\number` prefix would not, if not itself embedded in another `\the\numexpr` or similar context.
- (ctd.) The private format to which `\xintiexpr...relax` (et al.) evaluates needs `\xintthe` prefix to be printed on the page, or be used in macros (expanding their argument.) The `\the` TeX primitive prefix would not work here.
- (ctd.) As a synonym to `\xintthe\xintiexpr` one can use `\xinttheiexpr`, or (since 1.2h) `\thexintiexpr`.
- (ctd.) One can embed a `\numexpr...relax` (with its `\relax!`) inside an `\xintiexpr...relax` without `\the` or `\number`, but the reverse situation requires use of `\xintthe`.
- `\numexpr -(1)\relax` is illegal. But `\xintiexpr -(1)\relax` is perfectly legal and gives the expected result (what else ?).
- `\numexpr 2\cnta\relax` is illegal (with `\cnta` a `\count` register.) But `\xintiexpr 2\cnta\relax` is perfectly legal and will do the tacit multiplication.
- `\the\numexpr` or `\number\numexpr` expands in one step, but `\xintthe\xintiexpr` or `\xinttheiexpr` needs two steps.

2.10 Chaining expressions for expandable algorithmics

We will see in this section how to chain `\xintexpr`-expressions with `\expandafter`'s, like it is possible with `\numexpr`. For this it is convenient to use `\romannumeral0\xintexpr` which is the once-expanded form of `\xintexpr`, as we can then chain using only one `\expandafter` each time.

For example, here is the code employed on the title page to compute (expandably, of course!) the 1250th Fibonacci number:

```
\catcode`_ 11
\def\Fibonacci #1{% \Fibonacci{N} computes F(N) with F(0)=0, F(1)=1.
  \expandafter\Fibonacci_a\expandafter
    {\the\numexpr #1\expandafter}\expandafter
    {\romannumeral0\xintiexpro 1\expandafter\relax\expandafter}\expandafter
    {\romannumeral0\xintiexpro 1\expandafter\relax\expandafter}\expandafter
    {\romannumeral0\xintiexpro 1\expandafter\relax\expandafter}\expandafter
    {\romannumeral0\xintiexpro 0\relax}}
%
\def\Fibonacci_a #1{%
  \ifcase #1
```

¹⁹ The `\numexpr` triggers continued expansion after the space following the 3 to check if some operator like + is upstream. But after having found the 5 it treats it as an end-marker. ²⁰ Since 1.21 one can also use the underscore `_` to separate digits for readability of long numbers.

```

\expandafter\Fibonacci_end_i
\or
\expandafter\Fibonacci_end_ii
\else
\ifodd #1
\expandafter\expandafter\expandafter\Fibonacci_b_ii
\else
\expandafter\expandafter\expandafter\Fibonacci_b_i
\fi
\fi {#1}%
}% * signs are omitted from the next macros, tacit multiplications
\def\Fibonacci_b_i #1#2#3{\expandafter\Fibonacci_a\expandafter
{\the\numexpr #1/2\expandafter}\expandafter
{\romannumeral0\xintiipro sq(#2)+sq(#3)\expandafter\relax\expandafter}\expandafter
{\romannumeral0\xintiipro (2#2-#3)#3\relax}%
}% end of Fibonacci_b_i
\def\Fibonacci_b_ii #1#2#3#4#5{\expandafter\Fibonacci_a\expandafter
{\the\numexpr (#1-1)/2\expandafter}\expandafter
{\romannumeral0\xintiipro sq(#2)+sq(#3)\expandafter\relax\expandafter}\expandafter
{\romannumeral0\xintiipro (2#2-#3)#3\expandafter\relax\expandafter}\expandafter
{\romannumeral0\xintiipro #2#4+#3#5\expandafter\relax\expandafter}\expandafter
{\romannumeral0\xintiipro #2#5+#3(#4-#5)\relax}%
}% end of Fibonacci_b_ii
% code as used on title page:
%\def\Fibonacci_end_i #1#2#3#4#5{\xintthe#5}
%\def\Fibonacci_end_ii #1#2#3#4#5{\xinttheiiexpr #2#5+#3(#4-#5)\relax}
% new definitions:
\def\Fibonacci_end_i #1#2#3#4#5{{#4}{#5}}% {F(N+1)}{F(N)} in \xintexpr format
\def\Fibonacci_end_ii #1#2#3#4#5%
{\expandafter
{\romannumeral0\xintiipro #2#4+#3#5\expandafter\relax
\expandafter}\expandafter
{\romannumeral0\xintiipro #2#5+#3(#4-#5)\relax}}% idem.
% \FibonacciN returns F(N) (in encapsulated format: needs \xintthe for printing)
\def\FibonacciN {\expandafter\xint_secondoftwo\romannumeral-`0\Fibonacci }%
\catcode`_ 8

```

The macro `\Fibonacci` produces not one specific value $F(N)$ but a pair of successive values $\{F(N)\}\{F(N+1)\}$ which can then serve as starting point of another routine devoted to compute a whole sequence $F(N)$, $F(N+1)$, $F(N+2)$, \dots . Each of $F(N)$ and $F(N+1)$ is kept in the encapsulated internal `xintexpr` format.

`\FibonacciN` produces the single $F(N)$. It also keeps it in the private format; thus printing it will need the `\xintthe` prefix.

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

```

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

```

The way we use `\expandafter`'s to chain successive `\xintiipro` evaluations is exactly analogous to what is possible with `\numexpr`. The various `\romannumeral0\xintiipro` could very well all have been `\xintexpr`'s but then we would have needed `\expandafter\expandafter\expandafter` each time.

There is a difference though: `\numexpr` does *NOT* expand inside an `\edef`, and to force its expansion we must prefix it with `\the` or `\number` or `\romannumeral` or another `\numexpr` which is itself prefixed, etc. . . .

But `\xintexpr`, `\xintexpr`, . . . , expand fully in an `\edef`, with the completely expanded re-

sult encapsulated in a private format.

Using `\xintthe` as prefix is necessary to print the result (like `\the` or `\number` in the case of `\numexpr`), but it is not necessary to get the computation done (contrarily to the situation with `\numexpr`).

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

```
\fdef \X {\FibonacciN {100}}
```

but it is usually about as efficient to employ `\edef`. And if we want

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

then `\edef` is necessary.

Allright, so let's now give the code to generate $\{F(N)\{F(N+1)\{F(N+2)\}\dots}$, using `\Fibonacci` for the first two and then using the standard recursion $F(N+2)=F(N+1)+F(N)$:

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

This `\FibonacciSeq` macro is completely expandable but it is not *f-expandable*.

This is not a problem in the next example which uses `\xintFor*` as the latter applies repeatedly full expansion to what comes next each time it fetches an item from its list argument. Thus `\xintFor*` still manages to generate the list via iterated full expansion.

```
\newcounter{myindex}% not which would overwrite theindex environment!
% (many have probably been bitten by this trap)
\tabskip 1ex
\fdef\Fibxxx{\FibonacciN {30}}%
\setcounter{myindex}{30}%
\ vbox{\halign{\bfseries#.\hfil#\hfil &\hfil #\cr
\xintFor* #1 in {\FibonacciSeq {30}{59}}\do
{\themyindex &\xintthe#1 &
\xintiiRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{myindex}\cr }}%
}\vrule
\ vbox{\halign{\bfseries#.\hfil#\hfil &\hfil #\cr
\xintFor* #1 in {\FibonacciSeq {60}{89}}\do
{\themyindex &\xintthe#1 &
\xintiiRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{myindex}\cr }}%
}\vrule
\ vbox{\halign{\bfseries#.\hfil#\hfil &\hfil #\cr
\xintFor* #1 in {\FibonacciSeq {90}{119}}\do
{\themyindex &\xintthe#1 &
\xintiiRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{myindex}\cr }}%
}%
```

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

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

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

3 The `xint` bundle

.1	Characteristics	47	.8	<code>\ifcase</code> , <code>\ifnum</code> , ... constructs	57
.2	Floating point evaluations	49	.9	No variable declarations are needed	57
.3	Expansion matters	50	.10	When expandability is too much	57
.4	Input formats for macros	52	.11	Possible syntax errors to avoid	58
.5	Output formats of macros	54	.12	Error messages	59
.6	Count registers and variables	54	.13	Package namespace, catcodes	60
.7	Dimension registers and variables	55	.14	Origins of the package	60

3.1 Characteristics

The main characteristics are:

1. exact algebra on ``big numbers'', integers as well as fractions,
2. floating point variants with user-chosen precision,
3. the computational macros are compatible with expansion-only context,
4. the bundle comes with parsers (integer-only, or handling fractions, or doing floating point computations) of infix operations implementing beyond infix operations extra features such as dummy variables.

Since 1.2 ``big numbers'' must have less than about 19950 digits: the maximal number of digits for addition is at 19968 digits, and it is 19959 for multiplication. The reasonable range of use of the package is with numbers of up to a few hundred digits.²¹

\TeX does not know off-hand how to print on the page such very long numbers, see subsection 1.3.

Integers with only 10 digits and starting with a 3 already exceed the \TeX bound; and \TeX does not have a native processing of floating point numbers (multiplication by a decimal number of a dimension register is allowed --- this is used for example by the *pgf* basic math engine.)

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

The *bigintcalc* package by HEIKO OBERDIEK provided expandable macros (using some of `\numexpr` possibilities, when available) on arbitrarily big integers, beyond the \TeX bound. It does not provide an expression parser.²² *xint* did it again using more of `\numexpr` for higher speed, and in a later evolution added handling of exact fractions, of scientific numbers, and an expression parser. Arbitrary precision floating points operations were added as a derivative, and not part of the initial design goal.

The concept of signed infinities, signed zeroes, NaN's, error traps...²³ have not been implemented, only the notion of `scientific notation with a given number of significant figures'.²⁴

The \LaTeX 3 project has implemented expandably floating-point computations with 16 significant figures (*l3fp*), including functions such as exp, log, sine and cosine.²⁵

More directly related to the *xint* bundle there is the *l3bigint* package, also devoted to big integers and in development a.t.t.o.w (2015/10/09, no division yet). It is part of the experimental trunk of the \LaTeX 3 Project and provides an expression parser for expandable arithmetic with big integers. Its author Bruno LE FLOCH succeeded brilliantly into implementing expandably the Karatsuba multiplication algorithm and he achieves *sub-quadratic growth for the computation time*. This shows up very clearly with numbers having thousands of digits, up to the maximum which a.t.t.o.w is at 8192 digits.

The *l3bigint* multiplication from late 2015 is observed to be roughly $3x$ -- $4x$ faster than the one from `\xintiexpr` in the range of 4000 to 5000 digits integers, and isn't far from being $9x$ faster

²¹ For example multiplication of integers having from 50 to 100 digits takes roughly of the order of the millisecond on a 2012 desktop computer. I compared this to using Python3: using `timeit` module on a wrapper defined as `return w*z` with random integers of 100 digits, I observe on the same computer a computation time of roughly $4 \cdot 10^{-7}$ s per call. And with `return str(w*z)` then this becomes more like $16 \cdot 10^{-7}$ s per call. And with `return str(int(W)*int(Z))` where `W` and `Z` are strings, this becomes about $26 \cdot 10^{-7}$ s (I am deliberately ignoring Python's `Decimal` module here...) Anyway, my sentence from earlier version of this documentation: *this is, I guess, at least about 1000 times slower than what can be expected with any reasonable programming language*, is about right. I then added: *nevertheless as compilation of a typical \LaTeX document already takes of the order of seconds and even dozens of seconds for long ones, this leaves room for reasonably many computations via *xintexpr* or via direct use of the macros of *xint/xintfrac*.* ²² One can currently use package *bnumexpr* to associate the *bigintcalc* macros with an expression parser. This may be unavailable in future if *bnumexpr* becomes more tightly associated with future evolutions or variants of *xintcore*. ²³ The latter exist as work-in-progress for some time in the source code. ²⁴ multiplication of two floats with `P=\xinttheDigits` digits is first done exactly then rounded to `P` digits, rather than using a specially tailored multiplication for floating point numbers which would be more efficient (it is a waste to evaluate fully the multiplication result with `2P` or `2P-1` digits.) ²⁵ at the time of writing (2014/10/28) the *l3fp* (exactly represented) floating point numbers have their exponents limited to ± 9999 .

at 8000 digits. On the other hand `\xintiexpr`'s multiplication is found to be on average roughly 2.5x faster than `l3bigint`'s for numbers up to 100 digits and the two packages achieve about the same speed at 900 digits: but each such multiplication of numbers of 900 digits costs about one or two tenths of a second on a 2012 desktop computer, whereas the order of magnitude is rather the ms for numbers with 50--100 digits.²⁶

Even with the superior `l3bigint` Karatsuba multiplication it takes about 3.5s on this 2012 desktop computer for a single multiplication of two 5000-digits numbers. Hence it is not possible to do routinely such computations in a document. I have long been thinking that without the expandability constraint much higher speeds could be achieved, but perhaps I have not given enough thought to sustain that optimistic stance.²⁷

I remain of the opinion that if one really wants to do computations with *thousands* of digits, one should drop the expandability requirement. Indeed, as clearly demonstrated long ago by the [pi computing file](#) by D. ROEGEL one can program \TeX to compute with many digits at a much higher speed than what `xint` achieves: but, direct access to memory storage in one form or another seems a necessity for this kind of speed and one has to renounce at the complete expandability.²⁸

3.2 Floating point evaluations

Floating point macros are provided by package `xintfrac` to work with a given arbitrary precision P . The default value is $P = 16$ meaning that the significands of the produced (non-zero) numbers have 16 decimal digits. The syntax to set the precision to P is

```
\xintDigits:=P;
```

The value is local to the group or environment (if using \TeX). To query the current value use `\xinttheDigits`.

Most floating point macros accept an optional first argument $[P]$ which then sets the target precision and replaces the `\xintDigits` assigned value (the $[P]$ must be repeated if the arguments are themselves `xintfrac` macros with arguments of their own.) In this section P refers to the prevailing `\xinttheDigits` float precision or to the target precision set in this way as an optional argument.

`\xintfloatexpr[Q]...\relax` also admits an optional argument $[Q]$ but it has an altogether different meaning: the computations are always done with the prevailing `\xinttheDigits` precision and the optional argument Q is used for the final rounding. This makes sense only if $Q < \text{\xinttheDigits}$ and is intended to clean up the result from dubious last digits.

The IEEE 754²⁹ requirement of *correct rounding* for addition, subtraction, multiplication, division and square root is achieved (in arbitrary precision) by the macros of `xintfrac` hence also by the infix operators `+`, `-`, `*`, `/`.

This means that for operands given with at most P significant digits (and arbitrary exponents) the output coincides exactly with the rounding of the exact theoretical result (barring overflow or underflow).

Due to a typographical oversight, this documentation (up to 1.2j) adjoined `^` and `**` to the above list of infix operators. But as is explained in [subsection 9.80](#), what is guaranteed regarding integer powers is an error of at most `0.52ulp`, not the correct rounding. Half-integer powers are computed as square roots of integer powers.

The rounding mode is `round to nearest, ties away from zero`. It is not customizable.

Currently `xintfrac` has no notion of NaNs or signed infinities or signed zeroes, but this is intended for the future.

²⁶ I have tested this again on 2016/12/19, but the macros have not changed on the `l3bigint` side and barely on the `xintcore` side, hence I got again the same results. . . ²⁷ The `apnum` package implements (non-expandably) arbitrary precision fixed point algebra and (v1.6) functions `exp`, `log`, `sqrt`, the trigonometrical direct and inverse functions. ²⁸ The Lua \TeX project possibly makes endeavours such as `xint` appear even more insane that they are, in truth: `xint` is able to handle fast enough computations involving numbers with less than one hundred digits and brings this to all engines.

Since release 1.2f, square root extraction achieves correct rounding in arbitrary precision.

The power function in the expression parsers accepts integer exponents and also half-integer exponents for float expressions.³⁰ A preliminary implementation of fractional powers is available see *xintlog*. Trigonometrical functions are available (*xinttrig*).

The maximal floating point decimal exponent is currently 2147483647 which is the maximal number handled by \TeX . The minimal exponent is its opposite. But this means that overflow or underflow are detected only via low-level *\numexpr* arithmetic overflows which are basically un-recoverable. Besides there are some border effects as the routines need to add or subtract lengths of numbers from exponents, possibly triggering the low-level overflows. In the future not only the Precision but also the maximal and minimal exponents *Emin* and *Emax* will be specifiable by the user.

Since 1.2f, the float macros round their inputs to the target precision *P* before further processing. Formerly, the initial rounding was done to *P+2* digits (and at least *P+3* for the power operation.)

The more ambitious model would be for the computing macros to obey the intrinsic precision of their inputs, i.e. to compute the correct rounding to *P* digits of the exact mathematical result corresponding to inputs allowed to have their own higher precision.³¹ This would be feasible by *xintfrac* which after all knows how to compute exactly, but I have for the time being decided that for reasons of efficiency, the chosen model is the one of rounding inputs to the target precision first.

The float macros of *xintfrac* have to handle inputs which not only may have much more digits than the target float precision, but may even be fractions: in a way this means infinite precision.

From releases 1.08a to 1.2j a fraction input *AeM/BeN* had its numerator and denominator *A* and *B* truncated to *Q+2* digits of precision, then the substituted fraction was correctly rounded to *Q* digits of precision (usually with *Q* set to *P+2*) and then the operation was implemented on such rounded inputs. But this meant that two fractions representing the same rational number could end up being rounded differently (with a difference of one unit in the last place), if it had numerators and denominators with at least *Q+3* digits.

Starting with release 1.2k a fractional input *AeM/BeN* is handled intrinsically: the fraction, independently of its representation *AeM/BeN*, is *correctly rounded* to *P* digits during the input parsing. Hence the output depends only on its arguments as mathematical fractions and not on their representatives as quotients.

Notice that in float expressions, the */* is treated as operator, and is applied to arguments which are generally already *P*-floats, hence the above discussion becomes relevant in this context only for the special input form *qfloat(A/B)* or when using a sub-expression *\xintexpr A/B\relax* embedded in the float expression with *A* or *B* having more digits than the prevailing float precision *P*.

3.3 Expansion matters

3.3.1 Full expansion of the first token

The whole business of *xint* is to build upon *\numexpr* and handle arbitrarily large numbers. Each basic operation is thus done via a macro: *\xintiiAdd*, *\xintiiSub*, *\xintiiMul*, *\xintiiDivision*. In order to handle more complex operations, it must be possible to nest these macros. An expandable macro can not execute a *\def* or an *\edef*. But the macro must expand its arguments to find the digits it is supposed to manipulate. \TeX provides a tool to do the job of (expandable !) repeated expansion of the first token found until hitting something non expandable, such as a digit, a *\de* token, a brace, a *\count* token, etc... is found. A space token also will stop the expansion (and

²⁹ The IEEE 754-1985 standard was for hardware implementations of binary floating-point arithmetic with a specific value for the precision (24 bits for single precision, 53 bits for double precision). The newer IEEE 754-2008 (https://en.wikipedia.org/wiki/IEEE_floating_point) normalizes five basic formats, three binaries and two decimals (16 and 34 decimal digits) and discusses extended formats with higher precision. These standards are only indirectly relevant to libraries like *xint* dealing with arbitrary precision. ³⁰ Half-integer exponents work inside expressions, but not via the *\xintFloatPower* macro.

³¹ The MPFR library <http://www.mpfr.org/> implements this but it does not know fractions!

be swallowed, contrarily to the non-expandable tokens).

By convention in this manual *f-expansion* ('`full expansion' or '`full first expansion') will be this \TeX process of expanding repeatedly the first token seen. For those familiar with \TeX 3 (which is not used by `xint`) this is what is called in its documentation full expansion (whereas expansion inside `\edef` would be described I think as '`exhaustive' expansion).

Most of the package macros, and all those dealing with computations³², are expandable in the strong sense that they expand to their final result via this *f-expansion*. This will be signaled in their descriptions via a star in the margin.

These macros not only have this property of *f-expandability*, they all begin by first applying *f-expansion* to their arguments. Again from \TeX 3's conventions this will be signaled by a margin annotation next to the description of the arguments.

3.3.2 Summary of important expandability aspects

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

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

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

To the contrary `\xinttheiexpr` and others have no issues with things such as `\xinttheiexpr \relax`.

2. using `\if...\fi` constructs inside the package macro arguments requires suitably mastering \TeX techniques (`\expandafter`'s and/or swapping techniques) to ensure that the *f-expansion* will indeed absorb the `\else` or closing `\fi`, else some error will arise in further processing. Therefore it is highly recommended to use the package provided conditionals such as `\xintifEq`, `\xintifGt`, `\xintifSgn`,... or, for \TeX users and when dealing with short integers the `etoolbox`³³ expandable conditionals (for small integers only) such as `\ifnumequal`, `\ifnumgreater`, ... Use of *non-expandable* things such as `\ifthenelse` is impossible inside the arguments of `xint` macros.

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

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

3. after the definition `\def\x {12}`, one can not use `-x` as input to one of the package macros: the *f-expansion* will act only on the minus sign, hence do nothing. The only way is to use the `\xintOpp` macro (or `\xintiiOpp` which is integer only) which obtains the opposite of a given number.

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

4. With the definition

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

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

³² except `\xintXTrunc`. ³³ <http://www.ctan.org/pkg/etoolbox>

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

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

or use the lowercase form of `\xintAdd`:

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

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

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

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

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

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

6. In the expression parsers of `xintexpr` such as `\xintexpr.\relax`, `\xintfloatexpr.\relax` the contents are expanded completely from left to right until the ending `\relax` is found and swallowed, and spaces and even (to some extent) catcodes do not matter.
7. For all variants, prefixing with `\xintthe` allows to print the result or use it in other contexts. Shortcuts `\xinttheexpr`, `\xintthefloatexpr`, `\xinttheiexpr`, ... are available.

3.4 Input formats for macros

Macros can have different types of arguments (we do not consider here the `\xintexpr`-parsers but only the macros of `xintcore/xint/xintfrac`). In a macro description, a margin annotation signals what is the argument type.

`num`
`x`

1. \TeX integers are handled inside a `\numexpr.\relax` hence may be count registers or variables. Beware that `-(1+1)` is not legal and raises an error, but `0-(1+1)` is. Also `2\cnta` with `\cnta` a `\count` isn't legal. Integers must be kept less than 2147483647 in absolute value, although the scaling operation `(a*b)/c` computes the intermediate product with twice as many bits.

The slash `/` does a `rounded` division which is a fact of life of `\numexpr` which I have found very annoying in at least nine cases out of ten, not to say ninety-nine cases out of one hundred. Besides, it is at odds with \TeX 's `\divide` which does a truncated division (non-expandably).

But to follow-suit `/` also does rounded integer division in `\xintiexpr.\relax`, and the operator `//` does there the truncated division.

`f`

2. the strict format applies to macros handling big integers but only *f-expanding* their arguments. After this *f-expansion* the input should be a string of digits, optionally preceded by a unique minus sign. The first digit can be zero only if it is the only digit. A plus sign is not accepted. `-0` is not legal in the strict format. Macros of `xint` with a double `ii` require this 'strict' format for the inputs.

`Num`
`f`

3. the extended integer format applies when the macro parses its arguments via `\xintNum`. The input may then have arbitrarily many leading minus and plus signs, followed by leading zeroes, and further digits. With `xintfrac` loaded, `\xintNum` is extended to accept fractions and its action is to truncate them to integers.

At 1.20 many macros from `xintcore/xint` which use `\xintNum` to parse their arguments got deprecated, see [subsection 7.29](#), [subsection 8.53](#), and [subsection 8.54](#).

All these macros have now been removed at 1.3.

Changed
at 1.3!

Frac
f

4. the fraction input format applies to the arguments of `xintfrac` macros handling genuine fractions. It allows two types of inputs: general and restricted. The restricted type is parsed faster, but... is restricted.

general: inputs of the shape `A.BeC/D.EeF`. Example:

```
\noindent\xintRaw{+-0367.8920280e17/-++278.289287e-15}\newline
\xintRaw{+++1253.2782e+++3/---0087.123e---5}\par
```

`-3678920280/278289287[31]`

`-12532782/87123[7]`

The input parser does not reduce fractions to smallest terms. Here are the rules of this general fraction format:

- everything is optional, absent numbers are treated as zero, here are some extreme cases:
`\xintRaw{}`, `\xintRaw{.}`, `\xintRaw{./1.e}`, `\xintRaw{-e}`, `\xintRaw{e/-1}`
`0/1[0]`, `0/1[0]`, `0/1[0]`, `0/1[0]`, `0/1[0]`
- `AB` and `DE` may start with pluses and minuses, then leading zeroes, then digits.
- `C` and `F` will be given to `\numexpr` and can be anything recognized as such and not provoking arithmetic overflow (the lengths of `B` and `E` will also intervene to build the final exponent naturally which must obey the \TeX bound).
- the `/`, `.` (numerator and/or denominator) and `e` (numerator and/or denominator) are all optional components.
- each of `A`, `B`, `C`, `D`, `E` and `F` may arise from *f-expansion* of a macro.
- the whole thing may arise from *f-expansion*, however the `/`, `.`, and `e` should all come from this initial expansion. The `e` of scientific notation is mandatorily lowercased.

restricted: inputs either of the shape `A[N]` or `A/B[N]`, which represents the fraction `A/B` times 10^N . The whole thing or each of `A`, `B`, `N` (but then not `/` or `[]`) may arise from *f-expansion*, `A` (after expansion) *must* have a unique optional minus sign and no leading zeroes, `B` (after expansion) *must* be a positive integer with no signs and no leading zeroes, `[N]` if present will be given to `\numexpr`. Any deviation from the rules above will result in errors.

Frac
f

Notice that `*`, `+` and `-` contrarily to the `/` (which is treated simply as a kind of delimiter) are not acceptable within arguments of this type (see [subsection 3.6](#) for some exceptions to this.)

Generally speaking, there should be no spaces among the digits in the inputs (in arguments to the package macros). Although most would be harmless in most macros, there are some cases where spaces could break havoc.³⁴ So the best is to avoid them entirely.

This is entirely otherwise inside an `\xintexpr`-ession, where spaces are ignored (except when they occur inside arguments to some macros, thus escaping the `\xintexpr` parser). See the [section 10](#).

There are also some slightly more obscure expansion types: in particular, the `\xintApplyInline` and `\xintFor*` macros from *xinttools* apply a special iterated *f-expansion*, which gobbles spaces, to the non-braced items (braced items are submitted to no expansion because the opening brace stops

³⁴ The `\xintNum` macro does not remove spaces between digits beyond the first non zero ones; however this should not really alter the subsequent functioning of the arithmetic macros, and besides, since *xintcore* 1.2 there is an initial parsing of the entire number, during which spaces will be gobbled. However I have not done a complete review of the legacy code to be certain of all possibilities after 1.2 release. One thing to be aware of is that `\numexpr` stops on spaces between digits (although it provokes an expansion to see if an infix operator follows); the exponent for `\xintiiPow` or the argument of the factorial `\xintiiFac` are only subjected to such a `\numexpr` (there are a few other macros with such input types in *xint*). If the input is given as, say `1 2\x` where `\x` is a macro, the macro `\x` will not be expanded by the `\numexpr`, and this will surely cause problems afterwards. Perhaps a later *xint* will force `\numexpr` to expand beyond spaces, but I decided that was not really worth the effort. Another immediate cause of problems is an input of the type `\xintiiAdd {<space>\x }{\y }`, because the space will stop the initial expansion; this will most certainly cause an arithmetic overflow later when the `\x` will be expanded in a `\numexpr`. Thus in conclusion, damages due to spaces are unlikely if only explicit digits are involved in the inputs, or arguments are single macros with no preceding space.

f* it) coming from their list argument; this is denoted by a special symbol in the margin. Some other macros such as `\xintSum` from `xintfrac` first do an *f*-expansion, then treat each found (braced or not) item (skipping spaces between such items) via the general fraction input parsing, this is signaled as here in the margin where the signification of the * is thus a bit different from the previous case.

n, resp. *o* $f \rightarrow \overset{\text{Frac}}{*} f$ A few macros from `xinttools` do not expand, or expand only once their argument. This is also signaled in the margin with notations à la \TeX 3.

3.5 Output formats of macros

We do not consider here the `\xintexpr`-parsers but only the macros from `xintcore`, `xint` and `xintfrac`. Macros of other components of the bundle may have their own output formats, for example for continuous fractions with `xintcfrac`. There are mainly three types of outputs:

- arithmetic macros from `xintcore/xint` deliver integers in the strict format as described in the previous section.
- arithmetic macros from `xintfrac` produce on output the strict fraction format `A/B[N]`, which stands for $(A/B) \times 10^N$, where *A* and *B* are integers, *B* is positive, and *N* is a "short" integer. The output is not reduced to smallest terms. The *A* and *B* may end with zeroes (i.e. *N* does not represent all powers of ten). The denominator *B* is always strictly positive. There is no + sign. The - is always first if present (i.e. the denominator on output is always positive.) The output will be expressed as such a fraction even if the inputs are both integers and the mathematical result is an integer. The *B*=1 is not removed.³⁵
- macros with `Float` in their names produce on output scientific format with `P=\xinttheDigits` digits, a lowercase *e* and an exponent *N*. The first digit is not zero, it is preceded by an optional minus sign and is followed by a dot and *P*-1 digits. Trailing zeroes are not trimmed. There is one exceptional case:
 - if the value is mathematically zero, it is output as `0.e0`, i.e. zeros after the decimal mark are removed and the exponent is always 0.

Future versions of the package may modify this.

3.6 Count registers and variables

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

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

The macros allowing the extended format for long numbers or dealing with fractions will to *some extent* allow the direct use of count registers and even infix algebra inside their arguments: a count register `\mycountA` or `\count 255` is admissible as numerator or also as denominator, with no need to be prefixed by `\the` or `\number`. It is possible to have as argument an algebraic expression as would be acceptable by a `\numexpr...\relax`, under this condition: *each of the numerator and*

³⁵ refer to the documentation of `\xintPraw` for an alternative.

denominator is expressed with at most nine tokens.^{36 37} Important: a slash for rounded division in a `\numexpr` should be written with braces `{/}` to not be confused with the `xintfrac` delimiter between numerator and denominator (braces will be removed internally and the slash will count for one token). Example: `\mycountA+\mycountB{/}17/1+\mycountA*\mycountB`, or `\count 0+\count 2{/}17/1+\count 0*\count 2`.

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

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

1. let the numerator and the denominator be presented as `\the\numexpr...\relax`,
2. or as `\numexpr {...}\relax` (the braces are removed during processing; they are not legal for `\numexpr...\relax` syntax.)

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

3.7 Dimension registers and variables

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

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

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

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

A [table of dimensions](#) illustrates that the internal values used by \TeX do not correspond always to the closest rounding. For example a millimeter exact value in terms of `sp` units is `72.27/10/2.54*65536=186467.981...` and \TeX uses internally `186467sp` (\TeX truncates to get an integral multiple of the `sp` unit; see at the end of this section the exact rules applied internally by \TeX).

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

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

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

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

```
`1157 dd' =1237.998474121093...pt
`1357 dd' =1451.999938964843...pt
```

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

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

³⁶ The 1.2k and earlier versions manual claimed up to 8 tokens, but low-level TeX error arose if the `\numexpr ... \relax` occupied exactly 8 tokens and evaluated to zero. With 1.21 and later, up to 9 tokens are always safe and one may even drop the ending `\relax`. But well, all these explanations are somewhat silly because prefixing by `\the` or `\number` is always working with arbitrarily many tokens. ³⁷ Attention! in the \LaTeX context a `\value{countername}` will behave ok only if it is first in the input, if not it will not get expanded, and braces around the name will be removed and chaos will ensue inside a `\numexpr`. One should enclose the whole input in `\the\numexpr...\relax` in such cases.



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

TeX dimensions

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

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

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

We thus discover that TeX (or rather here, e-TeX, but one can check that this works the same in TeX82), uses 1238/1157 as a conversion factor (and necessarily intermediate computations simulate higher precision than a priori available with integers less than 2^{31} or rather 2^{30} for dimensions). Hence the 1452/1357 ratio is irrelevant, an artefact of the rounding (or rather, as we see, truncating) for one dd to be expressed as an integral number of sp's.

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

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

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

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

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

The difference of circa 5Å is arguably tiny!

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

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

and the centered convergents of this fraction are 1/1, 15/14, 61/57, 107/100, 1238/1157, 11249/10513, 23736/22183, 296081/276709, 615898/575601, 11382245/10637527, 22148592/20699453, 1885709081/176233151, 543564351/508000000. We do recover the 1238/1157 therein!

Here is how TeX converts `abc.xyz...<unit>`. First the decimal is rounded to the nearest integral multiple of $1/65536$, say $X/65536$. The `<unit>` is associated to a ratio N/D , which represents `<unit>/pt`. For the Didot point the ratio is indeed $1238/1157$. TeX truncates the fraction XN/D to an integer M . The dimension is represented by M sp.

For more details refer to:

<http://tex.stackexchange.com/questions/338297/why-pdf-file-cannot-be-reproduced/338510#338510>.

3.8 `\ifcase`, `\ifnum`, ... constructs

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

```
\begin{enumerate}[nosep]\def\A{1}
\item \ifcase \xintSgn\A 0\or OK\else ERROR\fi
\item \ifcase \xintSgn\A\space 0\or OK\else ERROR\fi
\item \ifcase \xintSgn{\A} 0\or OK\else ERROR\fi
\end{enumerate}
```

1. ERROR
2. OK
3. OK

In order to use successfully `\if...\fi` constructions either as arguments to the `xint` bundle expandable macros, or when building up a completely expandable macro of one's own, one needs some \TeX technical expertise (see also *item 2* on page 51).

It is thus much to be recommended to use the expandable branching macros, provided by `xintfrac` such as `\xintifSgn`, `\xintifZero`, `\xintifOne`, `\xintifNotZero`, `\xintifTrueAelseB`, `\xintifCmp`, `\xintifGt`, `\xintifLt`, `\xintifEq`, `\xintifInt...` See their respective documentations. All these conditionals always have either two or three branches, and empty brace pairs `{}` for unused branches should not be forgotten.

If these tests are to be applied to standard \TeX short integers, it is more efficient to use (under \LaTeX) the equivalent conditional tests from the `etoolbox`³⁸ package.

3.9 No variable declarations are needed

There is no notion of a *declaration of a variable*.

To do a computation and assign its result to some macro `\z`, the user will employ the `\def`, `\edef`, or `\newcommand` (in \LaTeX) as usual, keeping in mind that two expansion steps are needed, thus `\edef` is initially the main tool:


```
\def\x{1729728} \def\y{352827927} \edef\z{\xintiiMul {\x}{\y}}
\meaning\z
```

macro:->610296344513856

As an alternative to `\edef` the package provides `\oodef` which expands exactly twice the replacement text, and `\fdef` which applies *f-expansion* to the replacement text during the definition.

```
\def\x{1729728} \def\y{352827927} \oodef\w {\xintiiMul \x\y} \fdef\z{\xintiiMul {\x}{\y}}
\meaning\w, \meaning\z
```

macro:->610296344513856, macro:->610296344513856

 In practice `\oodef` is slower than `\edef`, except for computations ending in very big final replacement texts (thousands of digits). On the other hand `\fdef` appears to be slightly faster than `\edef` already in the case of expansions leading to only a few dozen digits.

`xintexpr` does provide an interface to declare and assign values to identifiers which can then be used in expressions: [subsection 2.6](#).

3.10 When expandability is too much

Let's use the macros of [subsection 2.10](#) related to Fibonacci numbers. Notice that the 47th Fibonacci number is 2971215073 thus already too big for \TeX and $\varepsilon\text{-}\TeX$.

The `\FibonacciN` macro found in [subsection 2.10](#) is completely expandable, it is even *f-expandable*. We need a wrapper with `\xintthe` prefix

```
\def\theFibonacciN{\xintthe\FibonacciN}
```

³⁸ <http://www.ctan.org/pkg/etoolbox>

to print in the document or to use within `\message` (or `\TeX typeout`) to write to the log and terminal.

The `\xintthe` prefix also allows its use it as argument to the `xint` macros: for example if we are interested in knowing how many digits $F(1250)$ has, it suffices to issue `\xintLen {\theFibonacciN {1250}}` (which expands to `261`). Or if we want to check the formula $\gcd(F(1859), F(1573)) = F(\gcd(1859, 1573)) = F(143)$, we only need³⁹

```
\xintiiGCD{\theFibonacciN{1859}}{\theFibonacciN{1573}}=%
\theFibonacciN{\xintiiGCD{1859}{1573}}$
```

which produces:

```
343358302784187294870275058337 = 343358302784187294870275058337
```

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

This is very convenient but of course it repeats the complete evaluation each time it is done. In practice, it is often useful to store the result of such evaluations in macros. Any `\edef` will break expandability, but if the goal is at some point to print something to the `dvi` or `pdf` output, and not only to the `log` file, then expandability has to be broken one day or another!

Hence, in practice, if we want to print in the document some computation results, we can proceed like this and avoid having to repeat identical evaluations:

```
\begingroup
\def\A {1859} \def\B {1573}
\edef\X {\theFibonacciN\A} \edef\Y {\theFibonacciN\B}
\edef\GCDAB {\xintiiGCD\A\B}\edef\Z {\theFibonacciN\GCDAB}
\edef\GCDXY{\xintiiGCD\X\Y}
The identity $\gcd(F(\A),F(\B))=F(\gcd(\A,\B))$ can be checked via evaluation
of both sides: $\gcd(F(\A),F(\B))=\gcd(\printnumber\X,\printnumber\Y)=
\printnumber{\GCDXY} = F(\gcd(\A,\B)) = F(\GCDAB) =\printnumber\Z$. \par
% some further computations involving \A, \B, \X, \Y
\endgroup % closing the group removes assignments to \A, \B, ...
% or choose longer names less susceptible to overwrite something.
% Note: there is no LaTeX \newcommand which would be to \edef like \newcommand is to \def
```

The identity $\gcd(F(1859), F(1573)) = F(\gcd(1859, 1573))$ can be checked via evaluation of both sides: $\gcd(F(1859), F(1573)) = \gcd(14405827913044251198771689151504042869913161495023481014226268636701088272597575494722482437753529619459794869227357628882216309358018264080851775319974225695605529435028861585245173725088673642228492908228952455838894954421926557604129992902552659797113378761054522176234908415299798114131996600875176897034109975200799936107075760195202876324584695551467505894985013610208598628752325727241, 24438419251951185733282794597762619928539902481570619232605360900784013394036743212445223278959909515869581103189177976905803274125163259530761668666101372520086675409656988895101002288801683145934731013156651772159324934427986343994793711957587665447658279589092823900703131971355481220049386445313295248477472731626471511289078393) = 343358302784187294870275058337 = F(\gcd(1859, 1573)) = F(143) = 343358302784187294870275058337.$

One may legitimately ask the author: why expandability to such extremes, for things such as big fractions or floating point numbers (even continued fractions...) which anyhow can not be used directly within `TeX`'s primitives such as `\ifnum`? Why insist on a concept which is foreign to the vast majority of `TeX` users and even programmers?

I have no answer: it made definitely sense at the start of `xint` (see [subsection 3.14](#)) and once started I could not stop.

3.11 Possible syntax errors to avoid

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

³⁹ The `\xintiiGCD` macro is provided by both the `xintgcd` package (since 1.0) and by the `xint` package (since 1.3d).

- using `-` to prefix some macro: `-\xintiiSqr{35}/271`.⁴⁰
- using one pair of braces too many `\xintIrr{\xintiiPow {3}{13}}/243` (the computation goes through with no error signaled, but the result is completely wrong).
- things like `\xintiiAdd { \x}{\y}` as the space will cause `\x` to be expanded later, most probably within a `\numexpr` thus provoking possibly an arithmetic overflow.
- using `[]` and decimal points at the same time `1.5/3.5[2]`, or with a sign in the denominator `3/-5[7]`. The scientific notation has no such restriction, the two inputs `1.5/-3.5e-2` and `-1.5e2/3.5` are equivalent: `\xintRaw{1.5/-3.5e-2}=-15/35[2]`, `\xintRaw{-1.5e2/3.5}=-15/35[2]`.
- generally speaking, using in a context expecting an integer (possibly restricted to the \TeX bound) a macro or expression which returns a fraction: `\xinttheexpr 4/2\relax` outputs `4/2`, not `2`. Use `\xintNum {\xinttheexpr 4/2\relax}` or `\xinttheiexpr 4/2\relax` (which rounds the result to the nearest integer, here, the result is already an integer) or `\xinttheiiexpr 4/2\relax`. Or, divide in your head `4` by `2` and insert the result directly in the \TeX source.

3.12 Error messages

In situations such as division by zero, the \TeX run will be interrupted with some error message. The user is asked to hit the RETURN key thrice, which will display additional information. In non-interactive `nonstopmode` the \TeX run goes on uninterrupted and the error data will be found in the compilation log.

Here is an example interactive run:

```
! Undefined control sequence.
<argument> \ ! /
          DivisionByZero (hit <RET> thrice)
1.11 \xintiiDivision{123}{0}

?
! Undefined control sequence.
<argument> \ ! /
          Division of 123 by 0
1.11 \xintiiDivision{123}{0}

?
! Undefined control sequence.
<argument> \ ! /
          next: {0}{0}
1.11 \xintiiDivision{123}{0}

?
[1] (./temptest.aux)
Output written on temptest.dvi (1 page, 216 bytes).
Transcript written on temptest.log.
```

This is an experimental feature, which is in preparation for next major release.⁴¹ For the good functioning of this the macro with the weird appearance `\ ! /` (yes, this is a single control sequence) must be left undefined. I trust it will be `;-)`.⁴²

Deprecated macros also generate an (expandable) error message. Just hit the RETURN key once to proceed. Most deprecated macros at 1.20 are listed either in subsection 7.29 or subsection 8.53 or subsection 8.54. All were removed at 1.3.

 Changed at 1.3!

⁴⁰ to the contrary, this *is* allowed inside an `\xintexpr`-ession. ⁴¹ The related macros checking or resetting error flags are implemented in embryonic form but no user interface is provided with 1.21 release. ⁴² The implementation is cloned from \LaTeX 3, the `\ ! /` was chosen for its shortness.

The expression parsers are at 1.21 still using a slightly less evolved method which lets \TeX display an undefined control sequence name giving some indication of the underlying problem (we copied this method from the *bigintcalc* package). The name of the control sequence is the message.

<code>\xintError:ignored</code>	<code>\xintError:unknownfunction</code>
<code>\xintError:removed</code>	<code>\xintError:we_are_doomed</code>
<code>\xintError:inserted</code>	<code>\xintError:missing_xintthe!</code>

Some constructs in *xintexpr*-essions use delimited macros and there is thus possibility in case of an ill-formed expression to end up beyond the `\relax` end-marker. Such a situation can also occur from a non-terminated `\numexpr`:

```
\xinttheexpr 3 + \numexpr 5+4\relax followed by some LaTeX code...
```

as the `\numexpr` will swallow the `\relax` whose presence is mandatory for `\xinttheexpr`, errors will inevitably arise and may lead to very cryptic messages; but nothing unusual or especially traumatizing for the daring experienced \TeX / \LaTeX user, whose has seen zillions of un-helpful error messages already in her daily practice of \TeX / \LaTeX .⁴³

3.13 Package namespace, catcodes

The bundle packages needs that the `\space` and `\empty` control sequences are pre-defined with the identical meanings as in Plain \TeX (or $\LaTeX_2\epsilon$ which has the same macros).

Private macros of *xintkernel*, *xintcore*, *xinttools*, *xint*, *xintfrac*, *xintexpr*, *xintbinhex*, *xintgcd*, *xintseries*, and *xintcfrac* use one or more underscores `_` as private letter, to reduce the risk of getting overwritten. They almost all begin either with `\XINT_` or with `\xint_`, a handful of these private macros such as `\XINTsetupcatcodes`, `\XINTdigits` and those with names such as `\XINTinFloat...` or `\XINTinfloat...` do not have any underscore in their names (for obscure legacy reasons).

xintkernel provides `\odef`, `\oodef`, `\fdef`: if macros with these names already exist *xinttools* will not overwrite them. The same meanings are independently available under the names `\xintodef`, `\xintoodef`, etc...

Apart from `\thexintexpr`, `\thexintiexpr`, ... all other public macros from the *xint* bundle packages start with `\xint`.

For the good functioning of the macros, standard catcodes are assumed for the minus sign, the forward slash, the square brackets, the letter ``e'`. These requirements are dropped inside an `\xintexpr`-ession: spaces are gobbled, catcodes mostly do not matter, the `e` of scientific notation may be `E` (on input) ...

If a character used in the `\xintexpr` syntax is made active, this will surely cause problems; prefixing it with `\string` is one option. There is `\xintexprSafeCatcodes` and `\xintexprRestoreCatcodes` to temporarily turn off potentially active characters.

For advanced \TeX users. At loading time of the packages the catcode configuration may be arbitrary as long as it satisfies the following 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.

As pointed out in previous section the control sequence `\ ! /` must be left undefined.

3.14 Origins of the package

2013/03/28. Package *bigintcalc* by HEIKO OBERDIEK already provides expandable arithmetic operations on ``big integers'', exceeding the \TeX limits (of $2^{31} - 1$), so why another⁴⁴ one?

⁴³ not to mention the \LaTeX error messages used by Emacs AU \TeX mode also for Plain \TeX runs... ⁴⁴ this section was written before the *xintfrac* package; the author is not aware of another package allowing expandable computations with arbitrarily big fractions.

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.⁴⁵ What I had learned in this other thread thanks to interaction with ULRICH DIEZ and GL on expandable manipulations of tokens motivated me to try my hands at addition and multiplication.

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

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

2013/04/14. This initial `xint` was followed by `xintfrac` which handled exactly fractions and decimal numbers.

2013/05/25. Later came `xintexpr` and at the same time `xintfrac` got extended to handle floating point numbers.

2013/11/22. Later, `xinttools` was detached.

2014/10/28. Release 1.1 significantly extended the `xintexpr` parsers.

2015/10/10. Release 1.2 rewrote the core integer routines which had remained essentially unmodified, apart from a slight improvement of division early 2014.

This 1.2 release also got its impulse from a fast ```reversing''` macro, which I wrote after my interest got awakened again as a result of correspondance with Bruno LE FLOCH during September 2015: this new reverse uses a TeXnique which *requires* the tokens to be digits. I wrote a routine which works (expandably) in quasi-linear time, but a less fancy $O(N^2)$ variant which I developed concurrently proved to be faster all the way up to perhaps 7000 digits, thus I dropped the quasi-linear one. The less fancy variant has the advantage that `xint` can handle numbers with more than 19900 digits (but not much more than 19950). This is with the current common values of the input save stack and maximal expansion depth: 5000 and 10000 respectively.

4 Some utilities from the `xinttools` package

This is a first overview. Many examples combining these utilities with the arithmetic macros of `xint` are to be found in [section 17](#). See also [section 5](#).

4.1 Assignments

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

```
\xintAssign \xintiiDivision{\xintiiPow {2}{1000}}{\xintiiFac{100}}\A\B
give: \meaning\A: macro:->114813249641507505482278393872551066259805517784186172883663478065)
826541894704737970419535798876630484358265060061503749531707793118627774829601 and \meaning\B
B: macro:->549362945213398322513812878622391280734105004984760505953218996123132766490228838)
8132878702444582075129603152041054804964625083138567652624386837205668069376. Another example
(which uses \xintBezout from the xintgcd package):
```

```
\xintAssign \xintBezout{357}{323}\to\U\V\D
is equivalent to setting \U to -9, \V to 10, and \D to 17. And indeed  $-9 \times 357 + 10 \times 323 = 17$  is a
Bézout Identity.
```

⁴⁵ the `\ReverseOrder` could be avoided in that circumstance, but it does play a crucial rôle here.

Thus, what `\xintAssign` does is to first apply an *f-expansion* to what comes next; it then defines one after the other (using `\def`; an optional argument allows to modify the expansion type, see [subsection 17.21](#) for details), the macros found after `\to` to correspond to the successive braced contents (or single tokens) located prior to `\to`. In case the first token (after the optional parameter within brackets, cf. the [\xintAssign](#) detailed document) is not an opening brace `{`, `\xintAssign` consider that there is only one macro to define, and that its replacement text should be all that follows until the `\to`.

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

gives then `\U` with meaning `5812117166`, `\V` with meaning `-103530711951` and `\D` with meaning `3`.

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

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

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

```
% \newcount\cnta
% \newcount\cntb
\begingroup
\xintDigitsOf\xintiiPow{2}{100}\to\DIGITS
\cnta = 1
\cntb = 0
\loop
\advance \cntb \xintiiSqr{\DIGITS{\cnta}}
\ifnum \cnta < \DIGITS{0}
\advance\cnta 1
\repeat

|2^{100}| (= \xintiiPow {2}{100}) has \DIGITS{0} digits and the sum of their squares is \the\cntb.
These digits are, from the least to the most significant: \cnta = \DIGITS{0} \loop
\digint{\cnta}\ifnum \cnta > 1 \advance\cnta -1 , \repeat.\endgroup
2^{100} (=1267650600228229401496703205376) has 31 digits and the sum of their squares is 679.
These digits are, from the least to the most significant: 6, 7, 3, 5, 0, 2, 3, 0, 7, 6, 9, 4, 1, 0,
4, 9, 2, 2, 8, 2, 2, 0, 0, 6, 0, 5, 6, 7, 6, 2, 1.
```

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

4.2 Utilities for expandable manipulations

The package now has more utilities to deal expandably with 'lists of things', which were treated un-expandably in the previous section with `\xintAssign` and `\xintAssignArray`: `\xintReverseOrder` and `\xintLength` since the first release, `\xintApply` and `\xintListWithSep` since 1.04, `\xintRevWithBraces`, `\xintCSVtoList`, `\xintNthElt` since 1.06, `\xintApplyUnbraced`, since 1.06b, `\xintloop` and `\xintiloop` since 1.09g.⁴⁶

As an example the following code uses only expandable operations:

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

⁴⁶ All these utilities, as well as `\xintAssign`, `\xintAssignArray` and the `\xintFor` loops are now available from the `xinttools` package, independently of the big integers facilities of `xint`.

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

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

Expandably computing primes is done in [subsection 17.12](#).

4.3 A new kind of for loop

As part of the [utilities](#) coming with the [xinttools](#) package, there is a new kind of for loop, `\xintFor`. Check it out ([subsection 17.16](#) and also in next section).

4.4 A new kind of expandable loop

Also included in [xinttools](#), `\xintiloop` is an expandable loop giving access to an iteration index, without using count registers which would break expandability. Check it out ([subsection 17.14](#) and also in next section).

5 Additional examples using [xinttools](#) or [xintexpr](#) or both

Note: [xintexpr.sty](#) automatically loads [xinttools.sty](#).

5.1 Completely expandable prime test

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

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

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

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

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

Our integers are short, but without `\expandafter`'s with `\@firstoftwo`, or some other related techniques, direct use of `\ifnum..fi` tests is dangerous. So to make the macro more efficient we are going to use the expandable tests provided by the package [etoolbox](#)⁴⁷. The macro becomes:

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

⁴⁷ <http://ctan.org/pkg/etoolbox>

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

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

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

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

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

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

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

We did generate via the `\xintSeq` too many potential divisors though. Later sections give two variants: one with `\xintiloop` (subsection 5.2) which is still expandable and another one (subsection 5.5) which is a close variant of the `\IsPrime` code above but with the `\xintFor` loop, thus breaking expandability. The *xintiloop variant* does not first evaluate the integer square root, the *xintFor variant* still does. I did not compare their efficiencies.

Let us construct with this expandable primality test a table of the prime numbers up to 1000. We need to count how many we have in order to know how many tab stops one should add in the last row.⁴⁸ There is some subtlety for this last row. Turns out to be better to insert a `\` only when we know for sure we are starting a new row; this is how we have designed the `\OneCell` macro. And for the last row, there are many ways, we use again `\xintApplyUnbraced` but with a macro which gobbles its argument and replaces it with a tabulation character. The `\xintFor*` macro would be more elegant here.

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

⁴⁸ although a tabular row may have less tabs than in the preamble, there is a problem with the | vertical rule, if one does that.


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

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

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

There are 168 prime numbers up to 1000.

5.2 Another completely expandable prime test

The `\IsPrime` macro from [subsection 5.1](#) checked expandably if a (short) integer was prime, here is a partial rewrite using `\xintloop`. We use the `etoolbox` expandable conditionals for convenience, but not everywhere as `\xintloopindex` can not be evaluated while being braced. This is also the reason why `\xintbreaklooppando` is delimited, and the next macro `\SmallestFactor` which returns the smallest prime factor exemplifies that. One could write more efficient completely expandable routines, the aim here was only to illustrate use of the general purpose `\xintloop`. A little table giving the first values of `\SmallestFactor` follows, its coding uses `\xintFor`, which is described later; none of this uses count registers.

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

```

{\ifnumless {#1}{8}
  {#1}% 3,5,7 are primes
  {\xintilooop [3+2]
    \ifnum#1<\numexpr\xintilooopindex*\xintilooopindex\relax
      \xint_afterfi{\xintbreakilooopanddo#1.}%
    \fi
    \ifnum#1=\numexpr (#1/\xintilooopindex)*\xintilooopindex\relax
      \xint_afterfi{\xintbreakilooopanddo\xintilooopindex.}%
    \fi
    \iftrue\repeat
  }%
}% END OF THE ODD BRANCH
{2}% EVEN BRANCH
}%
\catcode\_ 8
{\centering
  \begin{tabular}{|c|*{10}c|}
    \hline
    \xintFor #1 in {0,1,2,3,4,5,6,7,8,9}\do {\&\bfseries #1}\
    \hline
    \bfseries 0&--&--&2&3&2&5&2&7&2&3\
    \xintFor #1 in {1,2,3,4,5,6,7,8,9}\do
    {\bfseries #1%
      \xintFor #2 in {0,1,2,3,4,5,6,7,8,9}\do
      {\&\SmallestFactor{#1#2}}\}%
    \hline
  \end{tabular}\par
}

```

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

5.3 Miller-Rabin Pseudo-Primality expandably

This section is based on my <http://tex.stackexchange.com/a/165008> post.

At the time of writing, the code at the link above is still the version from April 2016 and it needed some hacks to get recursive (pseudo)-functions defined. Since 1.2h of 2016/11/20 there is `\xintNewFunction` which allows us here to avoid such internal hacking.

And since 1.3 of 2018/03/01, it is possible to use `\xintdefiifunc` also for recursive definitions, so we use it here, but we can benefit from it only for modular exponentiation as the rest of the code uses `iter` or `break` statements which are not yet compatible with `\xintdefiifunc`.

The `isPseudoPrime(n)` is usable in `\xintiexpr`-essions and establishes if its (positive) argument is a Miller-Rabin PseudoPrime to the bases 2, 3, 5, 7, 11, 13, 17. If this is true and $n < 341550071728321$ (which has 15 digits) then n really is a prime number.

Similarly $n = 3825123056546413051$ (19 digits) is the smallest composite number which is a strong pseudo prime for bases 2, 3, 5, 7, 11, 13, 17, 19 and 23. It is easy to extend the code below to include these additional tests (we could make the list of tested bases an argument too, now that I think

about it.)

For more information see

https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Deterministic_variants_of_the_test
and

http://primes.utm.edu/prove/prove2_3.html

In particular, according to JÄESCHKE *On strong pseudoprimes to several bases*, Math. Comp., 61 (1993) 915–926, if $n < 4,759,123,141$ it is enough to establish Rabin-Miller pseudo-primality to bases $a = 2, 7, 61$ to prove that n is prime. This range is enough for \TeX numbers and we could then write a very fast expandable primality test for such numbers using only `\numexpr`. Left as an exercise...

```
% I ----- Modular Exponentiation
% Computes x^m modulo n (with m non negative).
% We will always use it with 1 < x < n

\xintdefiifunc powmod_a(x, m, n) :=
  ifone(m,
    % m=1, return x modulo n
    x /: n,
    % m > 1 test if odd or even and do recursive call
    if(odd(m), (x*sqr(powmod_a(x, m//2, n))) /: n,
      sqr(powmod_a(x, m//2, n)) /: n
    )
  );
\xintdefiifunc powmod(x, m, n) := if(m, powmod_a(x, m, n), 1);

% See http://tex.stackexchange.com/a/165008 for macros written directly by a
% human.

% For comparison here are the underlying support macros defined by
% \xintdefiifunc from the code above (since 1.3a): (with linebreaks added by
% TeX when writing to the log)

% Function powmod_a for \xintiexpr parser associated to \XINT_iexpr_userfunc_
% c_powmod_a with meaning macro:#1#2#3->\xintiifOne {#2}{\xintiiMod {#1}{#3}}{\x
% intiifNotZero {\xintiiOdd {#2}}{\xintiiMod {\xintiiMul {#1}{\xintiiSqr {\xintE
% xpandArgs {XINT_iexpr_userfunc_powmod_a}{#1}{\xintiiDivFloor {#2}{2}}{#3}}}}
% {#3}}{\xintiiMod {\xintiiSqr {\xintExpandArgs {XINT_iexpr_userfunc_powmod_a}{
% #1}{\xintiiDivFloor {#2}{2}}{#3}}}{#3}}

% Function powmod for \xintiexpr parser associated to \XINT_iexpr_userfunc_
% powmod with meaning macro:#1#2#3->\xintiifNotZero {#2}{\xintExpandArgs {XINT_i
% iexpr_userfunc_powmod_a}{#1}{#2}{#3}}{1}

% II ----- Miller-Rabin compositeness witness

% n=2^k m + 1 with m odd and k at least 1

% Choose 1<x<n.
% compute y=x^m modulo n
% if equals 1 we can't say anything
% if equals n-1 we can't say anything
% else put j=1, and
% compute repeatedly the square, incrementing j by 1 each time,
% thus always we have y^{2^{j-1}}
% -> if at some point n-1 mod n found, we can't say anything and break out
% -> if however we never find n-1 mod n before reaching
```

```

%      z=y^{2^{k-1}} with j=k
%      we then have z^2=x^{n-1}.
%      % Suppose z is not -1 mod n. If z^2 is 1 mod n, then n can be prime only if
%      % z is 1 mod n, and we can go back up, until initial y, and we have already
%      % excluded y=1. Thus if z is not -1 mod n and z^2 is 1 then n is not prime.
%      % But if z^2 is not 1, then n is not prime by Fermat. Hence (z not -1 mod n)
%      % implies (n is composite). (Miller test)

% let's use again xintexpr indecipherable (except to author) syntax. Of course
% doing it with macros only would be faster.

% Here \xintdefiifunc is not usable because not compatible with iter, break, ...
% but \xintNewFunction comes to the rescue.

\xintNewFunction{isCompositeWitness}[4]{% x=#1, n=#2, m=#3, k=#4
  subs((y==1)?{0}
    {iter(y;(j=#4)?{break!(@==#2-1)}
      {(@==#2-1)?{break(0)}{sqr(@)/:#2}},j=1++)}
    ,y=powmod(#1,#3,#2))}

% added note (2018/03/07) it is possible in the above that m=#3 is never
% zero, so we should rather call powmod_a for a small gain, but I don't
% have time to re-read the code comments and settle this.

% III ----- Strong Pseudo Primes

% cf
% http://oeis.org/A014233
% <http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html>
% <http://mathworld.wolfram.com/StrongPseudoprime.html>

% check if positive integer <49 si a prime.
% 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
\def\IsVerySmallPrime #1%
  {\ifnum#1=1 \xintdothis0\fi
  \ifnum#1=2 \xintdothis1\fi
  \ifnum#1=3 \xintdothis1\fi
  \ifnum#1=5 \xintdothis1\fi
  \ifnum#1=\numexpr (#1/2)*2\relax\xintdothis0\fi
  \ifnum#1=\numexpr (#1/3)*3\relax\xintdothis0\fi
  \ifnum#1=\numexpr (#1/5)*5\relax\xintdothis0\fi
  \xintorthat 1}

\xintNewFunction{isPseudoPrime}[1]{% n = #1
  (#1<49)?% use ? syntax to evaluate only what is needed
  {\IsVerySmallPrime{\xintthe#1}}% macro needs to be fed with #1 unlocked.
  {(even(#1))?
  {0}
  {subs(%
    % L expands to two values m, k hence isCompositeWitness does get
    % its four variables x, n, m, k
    isCompositeWitness(2, #1, L)?
    {0}%
    {isCompositeWitness(3, #1, L)?
    {0}%
    {isCompositeWitness(5, #1, L)?
    {0}%
  }
  }
  }
  }
  }
  }

```

```

        {isCompositeWitness(7, #1, L)?
        {0}%
% above enough for N<3215031751 hence all TeX numbers
        {isCompositeWitness(11, #1, L)?
        {0}%
% above enough for N<2152302898747, hence all 12-digits numbers
        {isCompositeWitness(13, #1, L)?
        {0}%
% above enough for N<3474749660383
        {isCompositeWitness(17, #1, L)?
        {0}%
% above enough for N<341550071728321
        {1}%
        }% not needed to comment-out end of lines spaces inside
        }% \xintexpr but this is too much of a habit for me with TeX!
        }% I left some after the ? characters.
        }%
        }%
        }% this computes (m, k) such that n = 2^k m + 1, m odd, k>=1
        , L=iter(#1//2;(even(@))?{@//2}{break(@,k)},k=1++)%
        }%
    }%
}

% if needed:
\def\IsPseudoPrime #1{\xinttheiexpr isPseudoPrime(#1)\relax}

\noindent The smallest prime number at least equal to 3141592653589 is
\xinttheiexpr
    seq(isPseudoPrime(3141592653589+n)?
        {break(3141592653589+n)}{omit}, n=0++)\relax.
% we could not use 3141592653589++ syntax because it works only with TeX numbers
\par

```

The smallest prime number at least equal to 3141592653589 is 3141592653601.

5.4 A table of factorizations

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

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

The code next generates a `table` which has been made into a float appearing on page 71. Here is now the code for factorization; the conditionals use the package provided `\xint_firstoftwo` and `\xint_secondoftwo`, one could have employed rather \TeX 's own `\@firstoftwo` and `\@secondoftwo`, or, simpler still in \TeX context, the `\ifnumequal`, `\ifnumless` . . . , utilities from the package `etoolbox` which do exactly that under the hood. Only \TeX acceptable numbers are treated here, but it would be easy to make a translation and use the `xint` macros, thus extending the scope to big numbers; naturally up to a cost in speed.

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

```

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

```

```

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

\def\factorize_b #1.#2.{\ifnum#1=1 \abortfactorize\fi
  \ifnum\numexpr #1-(#2-1)*#2<#2
    #1\abortfactorize
  \fi
  \ifnum \numexpr #1-#2=\numexpr ((#1/#2)-1)*#2\relax
    \expandafter\xint_firstoftwo
  \else\expandafter\xint_secondoftwo
  \fi
  {#2&\expandafter\factorize_b\the\numexpr#1/#2.#2.}%
  {\expandafter\factorize_b\the\numexpr #1\expandafter.%
    \the\numexpr #2+2.}}%

\catcode\_ 8
\begin{figure*}[ht!]
\centering\phantomsection\label{floatfactorize}\normalcolor
\tabskip1ex
\centeredline{\vbox{\halign {\hfil\strut#\hfil&&\hfil#\hfil\cr\noalign{\hrule}
  \xintilop ["7FFFFFFE0+1]
  \expandafter\bfseries\xintilopindex &
  \ifnum\xintilopindex="7FFFFFFED
    \number"7FFFFFFED\cr\noalign{\hrule}
  \expandafter\xintilopskiptonext
  \fi
  \expandafter\factorize\xintilopindex.\cr\noalign{\hrule}
  \ifnum\xintilopindex<"7FFFFFFE
  \repeat
  \bfseries \number"7FFFFFFF&\number "7FFFFFFF\cr\noalign{\hrule}
}}}
\centeredline{A table of factorizations}
\end{figure*}

```

5.5 Another table of primes

As a further example, let us dynamically generate a tabular with the first 50 prime numbers after 12345. First we need a macro to test if a (short) number is prime. Such a completely expandable macro was given in [subsection 5.1](#), here we consider a variant which will be slightly more efficient. This new `\IsPrime` has two parameters. The first one is a macro which it redefines to expand to the result of the primality test applied to the second argument. For convenience we use the `etoolbox` wrappers to various `\ifnum` tests, although here there isn't anymore the constraint of complete expandability (but using explicit `\if.. \fi` in tabulars has its quirks); equivalent tests are provided by `xint`, but they have some overhead as they are able to deal with arbitrarily big integers.

```

\def\IsPrime #1#2% #1=\Result, #2=tested number (assumed >0).
{\edef\TheNumber {\the\numexpr #2}% hence #2 may be a count or \numexpr.
  \ifnumodd {\TheNumber}
  {\ifnumgreater {\TheNumber}{1}
  {\edef\ItsSquareRoot{\xintiiSqrt \TheNumber}%
  \xintFor ##1 in {\xintintegers [3+2]}\do
  {\ifnumgreater {##1}{\ItsSquareRoot} % ##1 is a \numexpr.

```

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

A table of factorizations

```

        {\def#1{1}\xintBreakFor}
        {}%
    \ifnumequal {\TheNumber}{(\TheNumber/##1)*##1}
        {\def#1{0}\xintBreakFor }
        {}%
    }}
    {\def#1{0}}% 1 is not prime
    {\ifnumequal {\TheNumber}{2}{\def#1{1}}{\def#1{0}}}%
}

```

As we used `\xintFor` inside a macro we had to double the # in its #1 parameter. Here is now the code which creates the prime table (the table has been put in a `float`, which should be found on page 72):

```

\newcounter{primecount}
\newcounter{cellcount}
\begin{figure*}[ht!]
  \centering
  \begin{tabular}{|*{7}c|}
  \hline

```



```

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

```

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

5.6 Factorizing again

Here is an *f-expandable* macro which computes the factors of an integer. It uses the *xint* macros only.

```

\catcode \@ 11
\let\factorize\relax
\newcommand\Factorize [1]
{\romannumeral0\expandafter\factorize\expandafter{\romannumeral-\`0#1}}%
\newcommand\factorize [1]{\xintiifOne{#1}{ 1}{\factors@a #1.#1;}}%
\def\factors@a #1.{\xintiifOdd{#1}
{\factors@c 3.#1.%
{\expandafter\factors@b \expandafter1\expandafter.\romannumeral0\xinthalff{#1}.}}%
\def\factors@b #1.#2.{\xintiifOne{#2}
{\factors@end {2, #1}}%
{\xintiifOdd{#2}{\factors@c 3.#2.{2, #1}}%
{\expandafter\factors@b \the\numexpr #1+\@ne\expandafter.%
\romannumeral0\xinthalff{#2}.}}%
}%
\def\factors@c #1.#2.{%
\expandafter\factors@d\romannumeral0\xintiidivision {#2}{#1}{#1}{#2}%
}%
\def\factors@d #1#2#3#4{\xintiifNotZero{#2}
{\xintiifGt{#3}{#1}
{\factors@end {#4, 1}}% ultimate quotient is a prime with power 1
{\expandafter\factors@c\the\numexpr #3+\tw@.#4.}}%
{\factors@e 1.#3.#1.%
}%

```

```

\def\factorse #1.#2.#3.{\xintiifOne{#3}
  {\factors@end {#2, #1}}%
  {\expandafter\factors@f\romannumeral0\xintiidivision {#3}{#2}{#1}{#2}{#3}}%
}%
\def\factorsof #1#2#3#4#5{\xintiifNotZero{#2}
  {\expandafter\factors@c\the\numexpr #4+\tw@.#5.{#4, #3}}%
  {\expandafter\factors@e\the\numexpr #3+\@ne.#4.#1.}%
}%
\def\factorsofend #1;{\xintlistwithsep{, }{\xintRevWithBraces {#1}}}%
\catcode`@ 12

```

The macro will be acceptably efficient only with numbers having somewhat small prime factors.

```
\Factorize{16246355912554185673266068721806243461403654781833}
```

16246355912554185673266068721806243461403654781833, 13, 5, 17, 8, 29, 5, 37, 6, 41, 4, 59, 6

It puts a little stress on the input save stack in order not be bothered with previously gathered things.⁴⁹

Its output is a comma separated list with the number first, then its prime factors with multiplicity. Let's produce something prettier:

```

\catcode`_ 11
\def\ShowFactors #1{\expandafter\ShowFactors_a\romannumeral-\0\Factorize{#1},\relax,\relax,}
\def\ShowFactors_a #1,{#1=\ShowFactors_b}
\def\ShowFactors_b #1,#2,{\if\relax#1\else#1^{#2}\expandafter\ShowFactors_b\fi}
\catcode`_ 8
$$\ShowFactors{16246355912554185673266068721806243461403654781833}$$

```

16246355912554185673266068721806243461403654781833 = 13⁵17⁸29⁵37⁶41⁴59⁶

If we only considered small integers, we could write pure `\numexpr` methods which would be very much faster (especially if we had a table of small primes prepared first) but still ridiculously slow compared to any non expandable implementation, not to mention use of programming languages directly accessing the CPU registers. . .

5.7 The Quick Sort algorithm illustrated

First a completely expandable macro which sorts a comma separated list of numbers.⁵⁰

The `\QSx` macro expands its list argument, which may thus be a macro; its comma separated items must expand to integers or decimal numbers or fractions or scientific notation as acceptable to `xintfrac`, but if an item is itself some (expandable) macro, this macro will be expanded each time the item is considered in a comparison test! This is actually good if the macro expands in one step to the digits, and there are many many digits, but bad if the macro needs to do many computations. Thus `\QSx` should be used with either explicit numbers or with items being macros expanding in one step to the numbers (particularly if these numbers are very big).

If the interest is only in \TeX integers, then one should replace the `\xintifCmp` macro with a suitable conditional, possibly helped by tools such as `\ifnumgreater`, `\ifnumequal` and `\ifnumless` from `etoolbox` (\LaTeX only; I didn't see a direct equivalent to `\xintifCmp`.) Or, if we are dealing with decimal numbers with at most four+four digits, then one should use suitable `\ifdim` tests. Naturally this will boost consequently the speed, from having skipped all the overhead in parsing fractions and scientific numbers as are acceptable by `xintfrac` macros, and subsequent treatment.

```

% THE QUICK SORT ALGORITHM EXPANDABLY
% \usepackage{xintfrac} in the preamble (latex)

```

⁴⁹ 2015/11/18 I have not revisited this code for a long time, and perhaps I could improve it now with some new techniques.

⁵⁰ The code in earlier versions of this manual handled inputs composed of braced items. I have switched to comma separated inputs on the occasion of <http://tex.stackexchange.com/a/273084>. The version here is like code 3 on <http://tex.stackexchange.com> (which is about 3x faster than the earlier code it replaced in this manual) with a modification to make it more efficient if the data has many repeated values. A faster routine (for sorting hundreds of values) is provided as code 6 at the link mentioned in the footnote, it is based on Merge Sort, but limited to inputs which one can handle as \TeX dimensions. This code 6 could be extended to handle more general numbers, as acceptable by `xintfrac`. I have also written a non expandable version, which is even faster, but this matters really only when handling hundreds or rather thousands of values.

```

\makeatletter
% use extra safe delimiters
\catcode`! 3 \catcode`? 3
\def\Qsx {\romannumeral0\qsx }%
% first we check if empty list (else \qsx@finish will not find a comma)
\def\qsx #1{\expandafter\qsx@a\romannumeral-`0#1!,?}%
\def\qsx@a #1{\ifx,#1\expandafter\qsx@abort\else
\expandafter\qsx@start\fi #1}%
\def\qsx@abort #1?{ }%
\def\qsx@start {\expandafter\qsx@finish\romannumeral0\qsx@b,}%
\def\qsx@finish ,#1{ #1}%
%
% we check if empty of single and if not pick up the first as Pivot:
\def\qsx@b ,#1#2,#3{\ifx?#3\xintdothis\qsx@empty\fi
\ifx!#3\xintdothis\qsx@single\fi
\xintorthat\qsx@separate {#1#2}{#1#2#3}%
\def\qsx@empty #1#2#3#4#5{ }%
\def\qsx@single #1#2#3#4#5?{, #4}%
\def\qsx@separate #1#2#3#4#5#6,%
{%
\ifx!#5\expandafter\qsx@separate@done\fi
\xintifCmp {#5#6}{#4}%
\qsx@separate@appendtosmaller
\qsx@separate@appendtoequal
\qsx@separate@appendtogreater {#5#6}{#1}{#2}{#3}{#4}%
}%
%
\def\qsx@separate@appendtoequal #1#2{\qsx@separate {#2,#1}}%
\def\qsx@separate@appendtogreater #1#2#3{\qsx@separate {#2}{#3,#1}}%
\def\qsx@separate@appendtosmaller #1#2#3#4{\qsx@separate {#2}{#3}{#4,#1}}%
%
\def\qsx@separate@done\xintifCmp #1%
\qsx@separate@appendtosmaller
\qsx@separate@appendtoequal
\qsx@separate@appendtogreater #2#3#4#5#6#7?%
}%
\expandafter\qsx@f\expandafter {\romannumeral0\qsx@b #4!,?}{\qsx@b #5!,?}{#3}%
}%
%
\def\qsx@f #1#2#3{#2, #3#1}%
%
\catcode`! 12 \catcode`? 12
\makeatother

% EXAMPLE
\begingroup
\edef\z {\Qsx {1.0, 0.5, 0.3, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
1.3, 1.1, 0.7, 1.6, 0.6, 0.9, 0.8, 0.2, 0.1, 1.9}}
\meaning\z

\def\a {3.123456789123456789}\def\b {3.123456789123456788}
\def\c {3.123456789123456790}\def\d {3.123456789123456787}
\oodef\z {\Qsx { \a, \b, \c, \d}}%
% The space before \a to let it not be expanded during the conversion from CSV
% values to List. The \oodef expands exactly twice (via a bunch of \expandafter's)
\meaning\z
\endgroup

```

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

macro:->\d , \b , \a , \c (the spaces after \d, etc... come from the use of the \meaning primitive.)

The choice of pivot as first element is bad if the list is already almost sorted. Let's add a variant which will pick up the pivot index randomly. The previous routine worked also internally with comma separated lists, but for a change this one will use internally lists of braced items (the initial conversion via \xintCSVtoList handles all potential spurious space problems).

```
% QuickSort expandably on comma separated values with random choice of pivots
% =====> Requires availability of \pdfuniformdeviate <=====
% \usepackage{xintfrac, xinttools} in preamble
\makeatletter
\def\QSx {\romannumeral0\qsx }% This is a f-expandable macro.
% This converts from comma separated values on input and back on output.
% **** NOTE: these steps (and the other ones too, actually) are costly if input
%           has thousands of items.
\def\qsx #1{\xintlistwithsep{, }%
    {\expandafter\qsx@sort@a\expandafter{\romannumeral0\xintcsvtolist{#1}}}%
%
% we check if empty or single or double and if not pick up the first as Pivot:
\def\qsx@sort@a #1%
    {\expandafter\qsx@sort@b\expandafter{\romannumeral0\xintlength{#1}}{#1}}%
\def\qsx@sort@b #1{\ifcase #1
    \expandafter\qsx@sort@empty
    \or\expandafter\qsx@sort@single
    \or\expandafter\qsx@sort@double
    \else\expandafter\qsx@sort@c\fi {#1}}%
\def\qsx@sort@empty #1#2{ }%
\def\qsx@sort@single #1#2{#2}%
\catcode`_ 11
\def\qsx@sort@double #1#2{\xintifGt #2{\xint_exchangetwo_keepbraces}{#2}}%
\catcode`_ 8
\def\qsx@sort@c #1#2{%
    \expandafter\qsx@sort@sep@a\expandafter
        {\romannumeral0\xintnthelt{\pdfuniformdeviate #1+\@ne}{#2}}{#2}}%
\def\qsx@sort@sep@a #1{\qsx@sort@sep@loop {}{}{#1}}%
\def\qsx@sort@sep@loop #1#2#3#4#5%
{%
    \ifx?#5\expandafter\qsx@sort@sep@done\fi
    \xintifCmp {#5}{#4}%
        \qsx@sort@sep@appendtosmaller
        \qsx@sort@sep@appendtoequal
        \qsx@sort@sep@appendtogreater {#5}{#1}{#2}{#3}{#4}%
}%
%
\def\qsx@sort@sep@appendtoequal #1#2{\qsx@sort@sep@loop {#2}{#1}}%
\def\qsx@sort@sep@appendtogreater #1#2#3{\qsx@sort@sep@loop {#2}{#3}{#1}}%
\def\qsx@sort@sep@appendtosmaller #1#2#3#4{\qsx@sort@sep@loop {#2}{#3}{#4}{#1}}%
%
\def\qsx@sort@sep@done\xintifCmp #1%
    \qsx@sort@sep@appendtosmaller
    \qsx@sort@sep@appendtoequal
    \qsx@sort@sep@appendtogreater #2#3#4#5#6%
{%
    \expandafter\qsx@sort@recurse\expandafter
        {\romannumeral0\qsx@sort@a {#4}}{\qsx@sort@a {#5}}{#3}}%
}%
```

```

%
\def\qsx@sort@recurse #1#2#3{#2#3#1}%
%
\makeatother

% EXAMPLES
\begingroup
\edef\z {\QSx {1.0, 0.5, 0.3, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
1.3, 1.1, 0.7, 1.6, 0.6, 0.9, 0.8, 0.2, 0.1, 1.9}}
\meaning\z

\def\ a {3.123456789123456789}\def\ b {3.123456789123456788}
\def\ c {3.123456789123456790}\def\ d {3.123456789123456787}
\oodef\z {\QSx { \ a, \ b, \ c, \ d}}%
% The space before \ a to let it not be expanded during the conversion from CSV
% values to List. The \oodef expands exactly twice (via a bunch of \expandafter's)
\meaning\z

\def\somenumbers{%
3997.6421, 8809.9358, 1805.4976, 5673.6478, 3179.1328, 1425.4503, 4417.7691,
2166.9040, 9279.7159, 3797.6992, 8057.1926, 2971.9166, 9372.2699, 9128.4052,
1228.0931, 3859.5459, 8561.7670, 2949.6929, 3512.1873, 1698.3952, 5282.9359,
1055.2154, 8760.8428, 7543.6015, 4934.4302, 7526.2729, 6246.0052, 9512.4667,
7423.1124, 5601.8436, 4433.5361, 9970.4849, 1519.3302, 7944.4953, 4910.7662,
3679.1515, 8167.6824, 2644.4325, 8239.4799, 4595.1908, 1560.2458, 6098.9677,
3116.3850, 9130.5298, 3236.2895, 3177.6830, 5373.1193, 5118.4922, 2743.8513,
8008.5975, 4189.2614, 1883.2764, 9090.9641, 2625.5400, 2899.3257, 9157.1094,
8048.4216, 3875.6233, 5684.3375, 8399.4277, 4528.5308, 6926.7729, 6941.6278,
9745.4137, 1875.1205, 2755.0443, 9161.1524, 9491.1593, 8857.3519, 4290.0451,
2382.4218, 3678.2963, 5647.0379, 1528.7301, 2627.8957, 9007.9860, 1988.5417,
2405.1911, 5065.8063, 5856.2141, 8989.8105, 9349.7840, 9970.3013, 8105.4062,
3041.7779, 5058.0480, 8165.0721, 9637.7196, 1795.0894, 7275.3838, 5997.0429,
7562.6481, 8084.0163, 3481.6319, 8078.8512, 2983.7624, 3925.4026, 4931.5812,
1323.1517, 6253.0945}%

\oodef\z {\QSx \somenumbers}% produced as a comma+space separated list
% black magic as workaround to the shrinkability of spaces in last line...
\hsize 87\fontcharwd\font`0
\lcode`~ =32
\lowercase{\def~}{\discretionary}{~}{\kern\fontcharwd\font`0}}\catcode32 13
\noindent\phantom{000}\scantokens\expandafter{\meaning\z}\par
\endgroup
macro:->0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8,
1.9, 2.0
macro:->\d , \b , \ a , \ c
macro:->1055.2154, 1228.0931, 1323.1517, 1425.4503, 1519.3302, 1528.7301, 1560.2458,
1698.3952, 1795.0894, 1805.4976, 1875.1205, 1883.2764, 1988.5417, 2166.9040, 2382.4218,
2405.1911, 2625.5400, 2627.8957, 2644.4325, 2743.8513, 2755.0443, 2899.3257, 2949.6929,
2971.9166, 2983.7624, 3041.7779, 3116.3850, 3177.6830, 3179.1328, 3236.2895, 3481.6319,
3512.1873, 3678.2963, 3679.1515, 3797.6992, 3859.5459, 3875.6233, 3925.4026, 3997.6421,
4189.2614, 4290.0451, 4417.7691, 4433.5361, 4528.5308, 4595.1908, 4910.7662, 4931.5812,
4934.4302, 5058.0480, 5065.8063, 5118.4922, 5282.9359, 5373.1193, 5601.8436, 5647.0379,
5673.6478, 5684.3375, 5856.2141, 5997.0429, 6098.9677, 6246.0052, 6253.0945, 6926.7729,
6941.6278, 7275.3838, 7423.1124, 7526.2729, 7543.6015, 7562.6481, 7944.4953, 8008.5975,
8048.4216, 8057.1926, 8078.8512, 8084.0163, 8105.4062, 8165.0721, 8167.6824, 8239.4799,
8399.4277, 8561.7670, 8760.8428, 8809.9358, 8857.3519, 8989.8105, 9007.9860, 9090.9641,

```

9128.4052, 9130.5298, 9157.1094, 9161.1524, 9279.7159, 9349.7840, 9372.2699, 9491.1593, 9512.4667, 9637.7196, 9745.4137, 9970.3013, 9970.4849

All the previous examples were with numbers which could have been handled via `\ifdim` tests rather than the `\xintifCmp` macro from `xintfrac`; using `\ifdim` tests would naturally be faster. Even faster routine is code 6 at <http://tex.stackexchange.com/a/273084> which uses `\pdfescapestring` and a Merge Sort algorithm.

We then turn to a graphical illustration of the algorithm.⁵¹ For simplicity the pivot is always chosen as the first list item. Then we also give a variant which picks up the last item as pivot.

```
% in LaTeX preamble:
% \usepackage{xintfrac, xinttools}
% \usepackage{color}
% or, when using Plain TeX:
% \input xintfrac.sty \input xinttools.sty
% \input color.tex
%
% Color definitions
\definecolor{LEFT}{RGB}{216,195,88}
\definecolor{RIGHT}{RGB}{208,231,153}
\definecolor{INERT}{RGB}{199,200,194}
\definecolor{INERTpiv}{RGB}{237,237,237}
\definecolor{PIVOT}{RGB}{109,8,57}
% Start of macro definitions
\makeatletter
% \catcode`? 3 % a bit too paranoid. Normal ? will do.
%
% argument will never be empty
\def\QS@cmp@a #1{\QS@cmp@b #1??}%
\def\QS@cmp@b #1{\noexpand\QS@sep@A\@ne{#1}\QS@cmp@d {#1}}%
\def\QS@cmp@d #1#2{\ifx ?#2\expandafter\QS@cmp@done\fi
\xintifCmp {#1}{#2}\tw@\@ne\z@{#2}\QS@cmp@d {#1}}%
\def\QS@cmp@done #1{??}%
%
\def\QS@sep@A #1?{\QSLr\QS@sep@L #1\thr@@?#1\thr@@?#1\thr@@?}%
\def\QS@sep@L #1#2{\ifcase #1{#2}\or\or\else\expandafter\QS@sep@I@start\fi \QS@sep@L}%
\def\QS@sep@I@start\QS@sep@L {\noexpand\empty?\QSIr\QS@sep@I}%
\def\QS@sep@I #1#2{\ifcase#1\or{#2}\or\else\expandafter\QS@sep@R@start\fi\QS@sep@I}%
\def\QS@sep@R@start\QS@sep@I {\noexpand\empty?\QSRr\QS@sep@R}%
\def\QS@sep@R #1#2{\ifcase#1\or\or{#2}\else\expandafter\QS@sep@done\fi\QS@sep@R}%
\def\QS@sep@done\QS@sep@R {\noexpand\empty?}%
%
\def\QS@loop {%
\xintloop
% pivot phase
\def\QS@pivotcount{0}%
\let\QSLr\DecoLEFTwithPivot \let\QSIr \DecoINERT
\let\QSRr\DecoRIGHTwithPivot \let\QSIrr\DecoINERT
\centerline{\QS@list}%
% sorting phase
\ifnum\QS@pivotcount>\z@
\def\QSLr {\QS@cmp@a}\def\QSRr {\QS@cmp@a}%
\def\QSIr {\QSIrr}\let\QSIrr\relax
\edef\QS@list{\QS@list}% compare
```

⁵¹ I have rewritten (2015/11/21) the routine to do only once (and not thrice) the needed calls to `\xintifCmp`, up to the price of one additional `\edef`, although due to the context execution time on our side is not an issue and moreover is anyhow overwhelmed by the TikZ's activities. Simultaneously I have updated the code <http://tex.stackexchange.com/a/142634/4686>. The variant with the choice of pivot on the right has more overhead: the reason is simply that we do not convert the data into an array, but maintain a list of tokens with self-reorganizing delimiters.

```

\let\QSLr\relax\let\QSRr\relax\let\QSIr\relax
\edef\QS@list{\QS@list}% separate
\def\QSLr ##1##2?{\ifx\empty##1\else\noexpand \QSLr {##1}##2}\fi}%
\def\QSIr ##1##2?{\ifx\empty##1\else\noexpand \QSIr {##1}##2}\fi}%
\def\QSRr ##1##2?{\ifx\empty##1\else\noexpand \QSRr {##1}##2}\fi}%
\edef\QS@list{\QS@list}% gather
\let\QSLr\DecoLEFT \let\QSRr\DecoRIGHT
\let\QSIr\DecoINERTwithPivot \let\QSIrr\DecoINERT
\centerline{\QS@list}%
\repeat }%
%
% \xintFor* loops handle gracefully empty lists.
\def\DecoLEFT #1{\xintFor* ##1 in {#1} \do {\colorbox{LEFT}{##1}}}%
\def\DecoINERT #1{\xintFor* ##1 in {#1} \do {\colorbox{INERT}{##1}}}%
\def\DecoRIGHT #1{\xintFor* ##1 in {#1} \do {\colorbox{RIGHT}{##1}}}%
\def\DecoPivot #1{\begingroup\color{PIVOT}\advance\fbboxsep-\fbboxrule\fbbox{#1}\endgroup}%
%
\def\DecoLEFTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
\xintFor* ##1 in {#1} \do
{\xintifForFirst {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}%
\def\DecoINERTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
\xintFor* ##1 in {#1} \do
{\xintifForFirst {\colorbox{INERTpiv}{##1}}{\colorbox{INERT}{##1}}}%
\def\DecoRIGHTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
\xintFor* ##1 in {#1} \do
{\xintifForFirst {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}%
%
\def\QuickSort #1{% warning: not compatible with empty #1.
% initialize, doing conversion from comma separated values to a list of braced items
\edef\QS@list{\noexpand\QSRr{\xintCSVtoList{#1}}}% many \edef's are to follow anyhow
% earlier I did a first drawing of the list, here with the color of RIGHT elements,
% but the color should have been for example white, anyway I drop this first line
\let\QSRr\DecoRIGHT
%\par\centerline{\QS@list}%
%
% loop as many times as needed
\QS@loop }%
%
% \catcode`? 12 % in case we had used a funny ? as delimiter.
\makeatother
%% End of macro definitions.
%% Start of Example
\begingroup\offinterlineskip
\small
% \QuickSort {1.0, 0.5, 0.3, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
%             1.3, 1.1, 0.7, 1.6, 0.6, 0.9, 0.8, 0.2, 0.1, 1.9}
% \medskip
% with repeated values
\QuickSort {1.0, 0.5, 0.3, 0.8, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
            1.3, 1.1, 0.7, 0.3, 1.6, 0.6, 0.3, 0.8, 0.2, 0.8, 0.7, 1.2}
\endgroup

```

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

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

Here is the variant which always picks the pivot as the rightmost element.

```

\makeatletter
%
\def\QS@cmp@a #1{\noexpand\QS@sep@a\expandafter\QS@cmp@d\expandafter
    {\romannumeral0\xintnthelt{-1}{#1}}#1??}%
%
\def\DecoLEFTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
    \xintFor* ##1 in {#1} \do
        {\xintifForLast {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}}
\def\DecoINERTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
    \xintFor* ##1 in {#1} \do
        {\xintifForLast {\colorbox{INERTpiv}{##1}}{\colorbox{INERT}{##1}}}}
\def\DecoRIGHTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
    \xintFor* ##1 in {#1} \do
        {\xintifForLast {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}}
\def\QuickSort #1{%
    % initialize, doing conversion from comma separated values to a list of braced items
    \edef\QS@list{\noexpand\QSLr {\xintCSVtoList{#1}}}% many \edef's are to follow anyhow
    %
    % loop as many times as needed
    \QS@loop }%
\makeatother
\beginngroup\offinterlineskip
\small
% \QuickSort {1.0, 0.5, 0.3, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
%             1.3, 1.1, 0.7, 1.6, 0.6, 0.9, 0.8, 0.2, 0.1, 1.9}
% \medskip
% with repeated values
\QuickSort {1.0, 0.5, 0.3, 0.8, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
            1.3, 1.1, 0.7, 0.3, 1.6, 0.6, 0.3, 0.8, 0.2, 0.8, 0.7, 1.2}
\endngroup

```

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

The choice of the first or last item as pivot is not a good one as nearly ordered lists will take quadratic time. But for explaining the algorithm via a graphical interpretation, it is not that bad. If one wanted to pick up the pivot randomly, the routine would have to be substantially rewritten: in particular the `\Deco.withPivot` macros need to know where the pivot is, and currently this is implemented by using either `\xintifForFirst` or `\xintifForLast`.

6 Macros of the [xintkernel](#) package

.1	<code>\odef</code> , <code>\oodef</code> , <code>\fdef</code>	80	.5	<code>\xintreplicate</code>	81
.2	<code>\xintReverseOrder</code>	80	.6	<code>\xintgobble</code>	81
.3	<code>\xintLength</code>	80	.7	(WIP) <code>\xintUniformDeviante</code>	81
.4	<code>\xintLastItem</code>	81			

The [xintkernel](#) package contains mainly the common code base for handling the load-order of the bundle packages, the management of catcodes at loading time, definition of common constants and macro utilities which are used throughout the code etc ... it is automatically loaded by all packages of the bundle.

It provides a few macros possibly useful in other contexts.

6.1 `\odef`, `\oodef`, `\fdef`

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

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

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

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

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

They can be prefixed with `\global`. It appears that `\fdef` is generally a bit faster than `\edef` when expanding macros from the `xint` bundle, when the result has a few dozens of digits. `\oodef` needs thousands of digits it seems to become competitive.

6.2 `\xintReverseOrder`

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

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

6.3 `\xintLength`

n ★ `\xintLength{⟨list⟩}` counts how many tokens (or braced items) there are (possibly none). It does no expansion of its argument, so to use it to count things in the replacement text of a macro `\x` one should do `\expandafter\xintLength\expandafter{x}`. Blanks between items are not counted. See also `\xintNthElt{0}` (from `xinttools`) which first *f-expands* its argument and then applies the same code.

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

6.4 `\xintLastItem`

n ★ `\xintLastItem{⟨list⟩}` returns the last item (unbraced) of its argument. If the list has no items the output is empty.

It does no expansion, which should be obtained via suitable `\expandafter`'s. See also `\xintNthElt{-1}` from [xinttools](#) which obtains the same result (but with another code) after having however *f-expanded* its argument first.

6.5 `\xintreplicate`

$\frac{\text{num}}{x}$ n ★ `\romannumeral\xintreplicate{x}{⟨stuff⟩}` is simply copied over from \TeX 's `\prg_replicate:nn` with some minor changes.⁵² It does not do any expansion of its second argument but inserts it in the upcoming token stream precisely x times. Using it with a negative x raises no error and does nothing.⁵³

Note that expansion must be triggered by a `\romannumeral`.

6.6 `\xintgobble`

$\frac{\text{num}}{x}$ ★ `\romannumeral\xintgobble{x}` is a Gobbling macro written in the spirit of \TeX 's `\prg_replicate:nn` (which I cloned as `\xintreplicate`.) It gobbles x tokens upstream, with x allowed to be as large as 531440. Don't use it with $x < 0$.

Note that expansion must be triggered by a `\romannumeral`.

`\xintgobble` looks as if it must be related to `\xintTrim` from [xinttools](#), but the latter uses different code (using directly `\xintgobble` is not possible because one must make sure not to gobble more than the number of available items; and counting available items first is an overhead which `\xintTrim` avoids.) It is rather `\xintKeep` with a negative first argument which hands over to `\xintgobble` (because in that case it is needed to count anyhow beforehand the number of items, hence `\xintgobble` can then be used safely.)

I wrote an `\xintcount` in the same spirit as `\xintreplicate` and `\xintgobble`. But it needs to be counting hundreds of tokens to be worth its salt compared to `\xintLength`.

6.7 (WIP) `\xintUniformDeviate`

$\frac{\text{num}}{x}$ ★ `\xintUniformDeviate{x}` is a wrapper of engine `\pdfuniformdeviate` (or `\uniformdeviate`).⁵⁴ The implementation is to be considered experimental for the time being.

The argument is expanded in `\numexpr` and the macro itself needs two expansion steps. It produces like the engine primitive an integer (digit tokens) with minimal value 0 and maximal one $x-1$ if x is positive, or minimal value $x+1$ and maximal value 0 if x is negative. For the discussion next, x is supposed positive as this avoids having to insert absolute values in formulas.

The underlying engine Random Number Generator works with an array of 55 28bits integers. To produce a « uniform » random integer in a given range $0..x-1$ it produces next pseudo-random y (supposedly uniformly distributed, i.e. non-uniformity can be neglected) such that $0 \leq y < 2^{28}$ and the output is the rounding of $x * (y/2^{28})$, with upper bound x remapped to 0. This has following corollaries:

1. with $x=2^{29}$ or $x=2^{30}$ the engine primitive produces only even numbers,
2. with $x=3*2^{26}$ the integers produced by the RNG when taken modulo three obey the proportion 1:1:2, not 1:1:1,
3. with $x=3*2^{14}$ there is analogous although weaker non-uniformity of the random integers when taken modulo 3,

⁵² I started with the code from Joseph WRIGHT's answer to <http://tex.stackexchange.com/questions/16189/repeat-command-n-times>.

⁵³ This behaviour may change in future. ⁵⁴ The `\uniformdeviate` primitive has been added to \TeX and will be available with \TeX Live 2019 release.

- generally speaking pure powers of two should generate uniform random integers, but when the range is divisible by large powers of two, the non-uniformity may be amplified in surprising ways by modulo operations.

These observations are not to be construed as criticism of the engine primitive itself, which comes from MetaPost, as the code comments and more generally the whole of *The Art of Computer Programming, Vol. 2* stresses that it should rather be seen as producing random fractions (the unit fraction being 2^{28}). Using it as a generator for integers is a bit of an abuse.

The first goal of `\xintUniformDeviate` is to guarantee a better uniformity for the distribution of random integers in any given range x .

If the probability to obtain a given y in $0..x-1$ is $(1+e(y))/x$, the 'relative non-uniformity' for that value y is $|e(y)|$.

The engine primitive guarantees only $x/2^{28}$ relative non-uniformity, and `\xintUniformDeviate` (in its current implementation) improves this by a factor $2^{28}=268435456$: the non-uniformity is guaranteed to be bounded by $x/2^{56}$.⁵⁵ With such a small non-uniformity, modulo phenomena as mentioned earlier are not observable in reasonable computing time.⁵⁶

```
%\xintdefiifunc mod3(x):= x 'mod' 3;
\xintNewIIExpr\ModThree[1]{#1 'mod' 3}

\pdfsetrandomseed 87654321
\xintdefiivar BadDigits:=qraw(
  \romannumeral\xintreplicate{503}{\ModThree{\pdfuniformdeviate "C0000000}},%
  \ModThree{\pdfuniformdeviate "C0000000}%
);% 504=503+1

\pdfsetrandomseed 87654321
\xintdefiivar GoodDigits:=qraw(
  \romannumeral\xintreplicate{503}{\ModThree{\xintUniformDeviate{"C0000000}},}%
  \ModThree{\xintUniformDeviate{"C0000000}}%
);% 504=503+1

These 504 digits generated from \string\pdfuniformdeviate:
\xinttheiexpr BadDigits\relax\hfill\break
contain these respective amounts of 0, 1, and 2:
% (this is definitely not the fastest way to count, but it is fun - and expandable)
\xinttheiexpr iter(0,0,0;(i=0)?{[0][0]+1,[0][1],[0][2]}
  {(i=1)?{[0][0],[0][1]+1,[0][2]}
    {[0][0],[0][1],[0][2]+1}},
  i=BadDigits)\relax\par

These 504 digits generated from \string\xintUniformDeviate:
\xinttheiexpr GoodDigits\relax\hfill\break
contain these respective amounts of 0, 1, and 2:
\xinttheiexpr iter(0,0,0;(i=0)?{[0][0]+1,[0][1],[0][2]}
  {(i=1)?{[0][0],[0][1]+1,[0][2]}
    {[0][0],[0][1],[0][2]+1}},
  i=GoodDigits)\relax\par
%% output to data file for double-check with python
```

⁵⁵ These estimates assume that the engine RNG underlying stream of 28-bits integers can be considered uniform; it is known that the parity bits of these 28-bits integers have a period of $55(2^{55}-1)$ and that after that many draws the count of 1s has only an excess of 55 compared to the count of 0s, so the scale seems to be an intrinsic non-uniformity of 2^{55} but it is not obvious if it applies to much shorter ranges. At any rate we assumed that the non-uniformity for x a power of two less than 2^{28} is negligible in comparison to 2^{-28} . Bigger powers of 2 produce only even integers because the output is rescaled by factor $x/2^{28}$! ⁵⁶ The function `qraw()` is used here not so much to speed up the loop expansion, but in relation to [subsection 10.5](#) to avoid too much usage of `\csname ... \endcsname` storage. Besides, if we had used `mod3` function we would have needed `\xinttheiexpr mod3(\pdfun ...)\relax` wrapping for each individual macro. So we decided rather to use a macro `\ModThree`; of course we could have used `\numexpr`, with the technical problem to expand only once `\pdfuniformdeviate`, which would have led to some extra manoeuvre, so ok for `\ModThree`.

It would be nice if the engine provided some user interface for letting its RNG execute a given number of iterations without the overhead of replicated executions of `\pdfuniformdeviate`. This could help gain entropy and would reduce correlations across series from distinct seeds.

The description above summarizes parts of discussions held with Bruno Le Floch in May 2018 on occasion of his LaTeX3 contributions related to this.

TeXhackers note: currently the implementation of `\xintUniformDeviate` consumes exactly 5 calls to the engine primitive at each execution; the improved $x/2^{56}$ non-uniformity could be obtained with only 2 calls, but paranoia about the phenomenon of seeds with common bits has led me to accept the overhead of using the 7 high bits of 4 random 28bits integers, rather than one single 28bits integer, or two, or three.

Timings indicate that one `\xintUniformDeviate` has a time cost about 13 times the one for one call to the engine primitive (and not only 5, as the extra arithmetic expressions add overhead which is more costly than the primitive itself). Except if the code using the pseudo-random number is very short, this time penalty will prove in practice much less severe (and this is one important reason why we opted for obtaining 28bits via the 7 high bits of 4 successive pseudo random numbers from the engine primitive). For example let's raise 100 times a random integer to the tenth power: ⁵⁷

```
\pdfsetrandomseed 12345678
\pdfresettimer\romannumeral\xintreplicate
  {100}{\fdef\foo{\xintiiPow{\xintUniformDeviate{100000000}}{10}}}%
\the\dimexpr\pdfelapsedtime sp\relax\space (with \string\xintUniformDeviate)\newline
(last result: \foo)\newline
\pdfsetrandomseed 12345678
\pdfresettimer\romannumeral\xintreplicate
  {100}{\fdef\foo{\xintiiPow{\pdfuniformdeviate 100000000}{10}}}%
\the\dimexpr\pdfelapsedtime sp\relax\space (with \string\pdfuniformdeviate)\newline
(last result: \foo)\par
```

0.05302pt (with `\xintUniformDeviate`)

(last result: 3613350655737796214900713110162983015171362952221525157584613965918429029376)

0.05066pt (with `\pdfuniformdeviate`)

(last result: 6625445910227856739067071963066183205761325943682291308182364687926259765625)

TeXhackers note: the macros `\xintRandomDigits` or `\xintiiRandRange`, and their variants, as well as the supporting macros for `random()` generate random decimal digits eight by eight as if using `\xintUniformDeviate{100000000}`, but via a direct optimized call made possibly by the range being a power of 10.

⁵⁷ This is done on a 2.4GHz processor. Hmm... or on a 2.8GHz one, I should add some automatic recognition to the build process...

7 Macros of the `xintcore` package

.1	<code>\xintiNum</code>	86	.16	<code>\xintiiSub</code>	87
.2	<code>\xintDouble</code>	86	.17	<code>\xintiiMul</code>	87
.3	<code>\xintHalf</code>	86	.18	<code>\xintiiSqr</code>	87
.4	<code>\xintInc</code>	86	.19	<code>\xintiiPow</code>	87
.5	<code>\xintDec</code>	86	.20	<code>\xintiiFac</code>	87
.6	<code>\xintDSL</code>	86	.21	<code>\xintiiDivision</code>	88
.7	<code>\xintDSR</code>	86	.22	<code>\xintiiQuo</code>	88
.8	<code>\xintDSRr</code>	86	.23	<code>\xintiiRem</code>	88
.9	<code>\xintFDg</code>	86	.24	<code>\xintiiDivRound</code>	88
.10	<code>\xintLDg</code>	86	.25	<code>\xintiiDivTrunc</code>	88
.11	<code>\xintiiSgn</code>	86	.26	<code>\xintiiDivFloor</code>	89
.12	<code>\xintiiOpp</code>	87	.27	<code>\xintiiMod</code>	89
.13	<code>\xintiiAbs</code>	87	.28	<code>\xintNum</code>	89
.14	<code>\xintiiAdd</code>	87	.29	Removed macros.....	89
.15	<code>\xintiiCmp</code>	87			

Package `xintcore` is automatically loaded by `xint`.

`xintcore` provides for big integers the four basic arithmetic operations (addition, subtraction, multiplication, division), as well as powers and factorials.

In the descriptions of the macros `{N}` and `{M}` stand for (big) integers or macros *f-expanding* to such big integers in strict format as described in subsection 3.4.

All macros require strict integer format on input and produce strict integer format on output, except:



- `\xintiNum` which converts to strict integer format an input in *extended* integer format, i.e. admitting multiple leading plus or minus signs, then possibly leading zeroes, then digits,
- and `\xintNum` which is an alias for the former, which gets redefined by `xintfrac` to accept more generally also decimal numbers or fractions as input and which truncates them to integers.

Most removed macros listed in subsection 7.29 were by design applying `\xintNum` to their inputs. Typically these macros had a single *i* in their names, for example `\xintiAdd` was such a companion to `\xintiiAdd`. `xintfrac` redefined `\xintNum` to be the macro accepting general fractional input and truncating it to an integer. Hence a macro such as `\xintiAdd` was compatible with the output format of `xintfrac` macros, contrarily to `\xintiiAdd` which handles only strict integer format for its inputs. Of course, `xintfrac` defined also its own `\xintAdd` which did the addition of its arguments without truncating them to integers... but whose output format is the `A/B[N]` format explained in subsection 3.5, hence even if representing a small integer it can not be used directly in a \TeX context such as `\ifnum`, contrarily to `\xintiAdd` or to `\xintiiAdd`.

This situation was the result of some early-on design decisions which now appear misguided and impede further development. Hence, at 1.20 it has been decided to deprecate *all* such *i*-macros. And they got removed from the package at 1.3.

Changed at 1.3!

The *ii* in the names of the macros such as `\xintiiAdd` serves to stress that they accept only strict integers as input (this is signaled by the margin annotation *f*), or macros *f-expanding* to such strict format (big) integers and that they produce strict integers as output.

Other macros, such as `\xintDouble`, lack the *ii*, but this is only a legacy of the history of the package and they have the same requirements for input and format of output as the *ii*-macros.⁵⁸

⁵⁸ Regarding `\xintFDg` and `\xintLDg`, this is a breaking change because formerly they used `\xintNum`.

The letter x (with margin annotation $\overset{\text{num}}{x}$) stands for an argument which will be handled embedded in `\numexpr..relax`. It will thus be completely expanded and must give an integer obeying the TeX bounds. See also [subsection 3.6](#). This is the case for the argument of `\xintiiFac` or the exponent argument of `\xintiiPow`.

The *★*'s in the margin are there to remind of the complete expandability, even *f-expandability* of the macros, as discussed in [subsubsection 3.3.1](#).

7.1 `\xintiNum`

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

```
\xintiNum{+-----+---000000000367941789479}
-367941789479
```

7.2 `\xintDouble`

f ★ `\xintDouble{N}` computes $2N$.

7.3 `\xintHalf`

f ★ `\xintHalf{N}` computes $N/2$ truncated towards zero.

7.4 `\xintInc`

f ★ `\xintInc{N}` evaluates $N+1$.

7.5 `\xintDec`

f ★ `\xintDec{N}` evaluates $N-1$.

7.6 `\xintDSL`

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

7.7 `\xintDSR`

f ★ `\xintDSR{N}` is truncated decimal shift right, *i.e.* it is the truncation of $N/10$ towards zero.

7.8 `\xintDSRr`

f ★ `\xintDSRr{N}` is rounded decimal shift right, *i.e.* it is the rounding of $N/10$ away from zero. It is needed in xintcore for use by `\xintiiDivRound`.

7.9 `\xintFDg`

f ★ `\xintFDg{N}` outputs the first digit (most significant) of the number.

7.10 `\xintLDg`

f ★ `\xintLDg{N}` outputs the least significant digit. When the number is positive, this is the same as the remainder in the Euclidean division by ten.

7.11 `\xintiiSgn`

f ★ `\xintiiSgn{N}` returns 1 if the number is positive, 0 if it is zero and -1 if it is negative.

7.12 `\xintiiOpp`

f ★ `\xintiiOpp{N}` outputs the opposite $-N$ of the number N .

Important note: an input such as `-\foo` is not legal, generally speaking, as argument to the macros of the `xint` bundle (except, naturally in `\xintexpr`-essions). The reason is that the minus sign stops the *f*-expansion done during parsing of the inputs. One must use the syntax `\xintiiOpp{\foo}` if one wants to pass `-\foo` as argument to other macros.

7.13 `\xintiiAbs`

f ★ `\xintiiAbs{N}` outputs the absolute value of the number.

7.14 `\xintiiAdd`

ff ★ `\xintiiAdd{N}{M}` computes the sum of the two (big) integers.

7.15 `\xintiiCmp`

ff ★ `\xintiiCmp{N}{M}` produces `1` if $N > M$, `0` if $N = M$, and `-1` if $N < M$.

At 1.21 this macro was moved from package `xint` to `xintcore`.

7.16 `\xintiiSub`

ff ★ `\xintiiSub{N}{M}` computes the difference $N - M$.

7.17 `\xintiiMul`

ff ★ `\xintiiMul{N}{M}` computes the product of two (big) integers.

7.18 `\xintiiSqr`

f ★ `\xintiiSqr{N}` produces the square.

7.19 `\xintiiPow`

f $\overset{\text{num}}{x}$ ★ `\xintiiPow{N}{x}` computes N^x . For $x=0$, this is 1. For $N=0$ and $x < 0$, or if $|N| > 1$ and $x < 0$, an error is raised. There will also be an error if x exceeds the maximal ε -TeX number `2147483647`, but the real limit for exponents comes from either the computation time or the settings of some TeX memory parameters.

Indeed, the maximal power of 2 which `xint` is able to compute explicitly is $2^{(2^{17})} = 2^{131072}$ which has `39457` digits. This exceeds the maximal size on input for the `xintcore` multiplication, hence any 2^N with a higher N will fail. On the other hand $2^{(2^{16})}$ has `19729` digits, thus it can be squared once to obtain $2^{(2^{17})}$ or multiplied by anything smaller, thus all exponents up to and including 2^{17} are allowed (because the power operation works by squaring things and making products).

7.20 `\xintiiFac`

f $\overset{\text{num}}{x}$ ★ `\xintiiFac{x}` computes the factorial.

7.26 `\xintiiDivFloor`

ff ★ `\xintiiDivFloor{M}{N}` computes $\text{floor}(M/N)$. For positive divisor $N > 0$ and arbitrary dividend M it is the same as the Euclidean quotient `\xintiiQuo`.

```
\xintiiQuo{1000}{57} (Euclidean), \xintiiDivFloor{1000}{57} (floored)\newline
\xintiiQuo{-1000}{57}, \xintiiDivFloor{-1000}{57}\newline
\xintiiQuo{1000}{-57}, \xintiiDivFloor{1000}{-57}\newline
\xintiiQuo{-1000}{-57}, \xintiiDivFloor{-1000}{-57}\par
17 (Euclidean), 17 (floored)
-18, -18
-17, -18
18, 17
```

7.27 `\xintiiMod`

ff ★ `\xintiiMod{M}{N}` computes $M - N * \text{floor}(M/N)$. For positive divisor $N > 0$ and arbitrary dividend M it is the same as the Euclidean remainder `\xintiiRem`.

Formerly, this macro computed $M - N * \text{trunc}(M/N)$. The former meaning is retained as `\xintiiMod-Trunc`.

```
\xintiiRem {1000}{57} (Euclidean), \xintiiMod {1000}{57} (floored),
\xintiiModTrunc {1000}{57} (truncated)\newline
\xintiiRem {-1000}{57}, \xintiiMod {-1000}{57}, \xintiiModTrunc {-1000}{57}\newline
\xintiiRem {1000}{-57}, \xintiiMod {1000}{-57}, \xintiiModTrunc {1000}{-57}\newline
\xintiiRem {-1000}{-57}, \xintiiMod {-1000}{-57}, \xintiiModTrunc {-1000}{-57}\par
31 (Euclidean), 31 (floored), 31 (truncated)
26, 26, -31
31, -26, 31
26, -31, -31
```

7.28 `\xintNum`

f ★ `\xintNum` is originally an alias for `\xintiNum`. But with `xintfrac` loaded its meaning is modified to accept more general inputs. It then becomes an alias to `\xintTTrunc` which truncates the general input to an integer in strict format.

7.29 Removed macros

Changed at 1.3! These macros were deprecated at 1.2o and removed at 1.3. `\xintiiFDg` (renamed to `\xintFDg`), `\xintiiLDg` (renamed to `\xintLDg`), `\xintiOpp`, `\xintiAbs`, `\xintiAdd`, `\xintCmp` (it gets defined by `xintfrac`, so deprecation will usually not be seen; the macro with this name from former `xintcore` should have been called `\xintiCmp` actually), `\xintSgn` (it also gets its proper definition from `xintfrac`), `\xintiSub`, `\xintiMul`, `\xintiDivision`, `\xintiQuo`, `\xintiRem`, `\xintiDivRound`, `\xintiDivTrunc`, `\xintiMod`, `\xintiSqr`, `\xintiPow`, `\xintiFac`.

8 Macros of the [xint](#) package

This package loads automatically [xintcore](#) (and [xintkernel](#)) hence all macros described in [section 7](#) are still available.

.1	\xintLen	91	.30	\xintiiifGt	94
.2	\xintReverseDigits	91	.31	\xintiiifLt	94
.3	\xintiiGCD	91	.32	\xintiiifOdd	94
.4	\xintiiLCM	91	.33	\xintiiSum	94
.5	\xintDecSplit	91	.34	\xintiiPrd	95
.6	\xintDecSplitL , \xintDecSplitR	92	.35	\xintiiSquareRoot	95
.7	\xintiiE	92	.36	\xintiiSqrt , \xintiiSqrtR	95
.8	\xintDSH	92	.37	\xintiiBinomial	95
.9	\xintDSHr , \xintDSx	92	.38	\xintiiPFactorial	96
.10	\xintiiEq	92	.39	\xintiiMax	97
.11	\xintiiNotEq	92	.40	\xintiiMin	97
.12	\xintiiGeq	93	.41	\xintiiMaxof	97
.13	\xintiiGt	93	.42	\xintiiMinof	97
.14	\xintiiLt	93	.43	\xintifTrueAelseB	97
.15	\xintiiGtorEq	93	.44	\xintifFalseAelseB	97
.16	\xintiiLtorEq	93	.45	\xintNOT	97
.17	\xintiiIsZero	93	.46	\xintAND	97
.18	\xintiiIsNotZero	93	.47	\xintOR	98
.19	\xintiiIsOne	93	.48	\xintXOR	98
.20	\xintiiOdd	93	.49	\xintANDof	98
.21	\xintiiEven	93	.50	\xintORof	98
.22	\xintiiMON	93	.51	\xintXORof	98
.23	\xintiiMMON	93	.52	\xintLen	98
.24	\xintiiifSgn	93	.53	Removed macros (they require xintfrac) ..	98
.25	\xintiiifZero	94	.54	Removed macros (they used \xintNum) ...	98
.26	\xintiiifNotZero	94	.55	(WIP) \xintRandomDigits	98
.27	\xintiiifOne	94	.56	(WIP) \xintXRandomDigits	99
.28	\xintiiifCmp	94	.57	(WIP) \xintiiRandRange	99
.29	\xintiiifEq	94	.58	(WIP) \xintiiRandRangeAtoB	100

This is 1.3e of 2019/04/05.

Version 1.0 was released 2013/03/28. Since 1.1 2014/10/28 the core arithmetic macros have been moved to a separate package [xintcore](#), which is automatically loaded by [xint](#). Only the [\xintiiSum](#), [\xintiiPrd](#), [\xintiiSquareRoot](#), [\xintiiPFactorial](#), [\xintiiBinomial](#) genuinely add to the arithmetic macros from [xintcore](#). ([\xintiiFac](#) which computes factorials is already in [xintcore](#).)

With the exception of [\xintLen](#), of the «Boolean logic macros» (see next paragraphs) all macros require inputs being integers in strict format, see [subsection 3.4](#).⁵⁹ The *ii* in the macro names is here as a reminder of that fact. The output is an integer in strict format, or a pair of two braced such integers for [\xintiiSquareRoot](#), with the exception of [\xintiiE](#) which may produce strings of zero's if its first argument is zero.

Macros [\xintDecSplit](#) and [\xintReverseDigits](#) are non-arithmetic and have their own specific rules.

For all macros described here for which it makes sense, package [xintfrac](#) defines a similar one without *ii* in its name. This will handle more general inputs: decimal, scientific numbers, fractions. The *ii* macros provided here by [xint](#) can be nested inside macros of [xintfrac](#) but the opposite does not apply, because the output format of the [xintfrac](#) macros, even for representing

⁵⁹ of course for conditionals such as [\xintiiifCmp](#) this constraint applies only to the first two arguments.

integers, is not understood by the `ii` macros. The «Boolean macros» `\xintAND` etc... are exceptions though, they work fine if served as inputs some *xintfrac* output, despite doing only *f*-expansion. Prior to 1.2o, these macros did apply the `\xintNum` or the more general *xintfrac* general parsing, but this overhead was deemed superfluous as it serves only to handle hand-written input and is not needed if the input is obtained as a nested chain of *xintfrac* macros for example.

Changed at 1.3!

Prior to release 1.2o, `xint` defined additional macros which applied `\xintNum` to their input arguments. All these macros were deprecated at 1.2o and have been removed at 1.3.

At 1.3d macros `\xintiigcd` and `\xintiilcm` from package *xintgcd* are also available from loading `xint` only. They are support macros for the (multi-arguments) functions `gcd()` and `lcm()` in `\xintiexpr`.

See [subsubsection 3.3.1](#) for the significance of the ^{Num}`f`, `f`, ^{num}`x` and `★` margin annotations.

8.1 `\xintilen`

^{Num}`f` `★` `\xintilen{N}` returns the length of the number, after its parsing via `\xintinum`. The count does not include the sign.

```
\xintilen{-12345678901234567890123456789}
```

29

Prior to 1.2o, the package defined only `\xintlen`, which is extended by *xintfrac* to fractions or decimal numbers, hence acquires a bit more overhead then.

8.2 `\xintReverseDigits`

3.60004pt, 8.39996pt, 12.0pt

`f` `★` `\xintReverseDigits{N}` will reverse the order of the digits of the number. `\xintRev` is the former denomination and is kept as an alias. Leading zeroes resulting from the operation are not removed. Contrarily to `\xintReverseOrder` this macro *f*-expands its argument; it is only usable with digit tokens. It does *not* apply `\xintNum` to its argument (so this must be done explicitly if the argument is an integer produced from some *xintfrac* macros). It does accept a leading minus sign which will be left upfront in the output.

```
\oodef\x{\xintReverseDigits
{98765432109876543210987654321098765432109876543210}}\meaning\x\par
\noindent\oodef\x{\xintReverseDigits {\xintReverseDigits
{98765432109876543210987654321098765432109876543210}}}\meaning\x\par
```

macro:->01234567890123456789012345678901234567890123456789

macro:->98765432109876543210987654321098765432109876543210

8.3 `\xintiigcd`

This is the same as the `\xintiigcd` from package *xintgcd*.

8.4 `\xintiilcm`

This is the same as the `\xintiilcm` from package *xintgcd*.

8.5 `\xintDecSplit`

^{num}`x` `f` `★` `\xintDecSplit{x}{N}` cuts the *N* (a list of digits) into two pieces *L* and *R*: it outputs `{L}{R}` where the original *N* is the concatenation *LR*. These two pieces are decided according to *x*:

- for *x*>0, *R* coincides with the *x* least significant digits. If *x* equals or exceeds the length of *N* the first piece *L* will thus be *empty*,
- for *x*=0, *R* is *empty*, and *L* is all of *N*,

8.12 `\xintiiGeq`

ff ★ `\xintiiGeq{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.
Important: the macro compares *absolute values*.

8.13 `\xintiiGt`

ff ★ `\xintiiGt{N}{M}` returns 1 if $N > M$, 0 otherwise.

8.14 `\xintiiLt`

ff ★ `\xintiiLt{N}{M}` returns 1 if $N < M$, 0 otherwise.

8.15 `\xintiiGtorEq`

ff ★ `\xintiiGtorEq{N}{M}` returns 1 if $N \geq M$, 0 otherwise. Extended by *xintfrac* to fractions.

8.16 `\xintiiLtorEq`

ff ★ `\xintiiLtorEq{N}{M}` returns 1 if $N \leq M$, 0 otherwise.

8.17 `\xintiiIsZero`

f ★ `\xintiiIsZero{N}` returns 1 if $N = 0$, 0 otherwise.

8.18 `\xintiiIsNotZero`

f ★ `\xintiiIsNotZero{N}` returns 1 if $N \neq 0$, 0 otherwise.

8.19 `\xintiiIsOne`

f ★ `\xintiiIsOne{N}` returns 1 if $N = 1$, 0 otherwise.

8.20 `\xintiiOdd`

f ★ `\xintiiOdd{N}` is 1 if the number is odd and 0 otherwise.

8.21 `\xintiiEven`

f ★ `\xintiiEven{N}` is 1 if the number is even and 0 otherwise.

8.22 `\xintiiMON`

f ★ `\xintiiMON{N}` computes $(-1)^N$.
`\xintiiMON {-280914019374101929}`

-1

8.23 `\xintiiMMON`

f ★ `\xintiiMMON{N}` computes $(-1)^{N-1}$.
`\xintiiMMON {280914019374101929}`

1

8.24 `\xintiiifSgn`

fnnn ★ `\xintiiifSgn{N}{A}{B}{C}` executes either the $\langle A \rangle$, $\langle B \rangle$ or $\langle C \rangle$ code, depending on its first argument being respectively negative, zero, or positive.

8.25 `\xintiiifZero`

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

8.26 `\xintiiifNotZero`

fnn ★ `\xintiiifNotZero{⟨N⟩}{⟨IsNotZero⟩}{⟨IsZero⟩}` expandably checks if the first mandatory argument *N* is not zero or is zero. It then either executes the first or the second branch.
Beware that both branches must be present.

8.27 `\xintiiifOne`

fnn ★ `\xintiiifOne{⟨N⟩}{⟨IsOne⟩}{⟨IsNotOne⟩}` expandably checks if the first mandatory argument *N* is one or not one. It then either executes the first or the second branch. Beware that both branches must be present.

8.28 `\xintiiifCmp`

ffnn ★ `\xintiiifCmp{⟨A⟩}{⟨B⟩}{⟨A<B⟩}{⟨A=B⟩}{⟨A>B⟩}` compares its first two arguments and chooses accordingly the correct branch.

8.29 `\xintiiifEq`

ffnn ★ `\xintiiifEq{⟨A⟩}{⟨B⟩}{⟨A=B⟩}{⟨not(A=B)⟩}` checks equality of its two first arguments and executes the corresponding branch.

8.30 `\xintiiifGt`

ffnn ★ `\xintiiifGt{⟨A⟩}{⟨B⟩}{⟨A>B⟩}{⟨not(A>B)⟩}` checks if $A > B$ and executes the corresponding branch.

8.31 `\xintiiifLt`

ffnn ★ `\xintiiifLt{⟨A⟩}{⟨B⟩}{⟨A<B⟩}{⟨not(A<B)⟩}` checks if $A < B$ and executes the corresponding branch.

8.32 `\xintiiifOdd`

fnn ★ `\xintiiifOdd{⟨A⟩}{⟨A odd⟩}{⟨A even⟩}` checks if *A* is an odd integer and executes the corresponding branch.

8.33 `\xintiiSum`

**f* ★ `\xintiiSum{⟨braced things⟩}` after expanding its argument expects to find a sequence of tokens (or braced material). Each is *f*-expanded, and the sum of all these numbers is returned.

```
\xintiiSum{{123}{-98763450}{\xintiiFac{7}}{\xintiiMul{3347}{591}}}\newline
\xintiiSum{1234567890}\newline
\xintiiSum{1234}\newline
\xintiiSum{}
```

-96780210

45

10

0

If $x < 0$ an out-of-range error is raised. Else, if $y < 0$ or if $x < y$ the macro evaluates to 0.

The allowable range is $0 \leq x \leq 99999999$. But this theoretical range includes binomial coefficients with more than the roughly 19950 digits that the arithmetics of [xint](#) can handle. In such cases, the computation will end up in a low-level TeX error after a long time.

It turns out that $\binom{65000}{32500}$ has 19565 digits and $\binom{64000}{32000}$ has 19264 digits. The latter can be evaluated (this takes a long long time) but presumably not the former (I didn't try). Reasonable feasible evaluations are with binomial coefficients not exceeding about one thousand digits.

The `binomial` function is available in the `xintexpr` parsers.

```
\xinttheiexpr seq(binomial(100,i), i=47..53)\relax
84413487283064039501507937600, 93206558875049876949581681100, 98913082887808032681188722800,
100891344545564193334812497256, 98913082887808032681188722800, 93206558875049876949581681100,
84413487283064039501507937600
```

See `\xintFloatBinomial` from package `xintfrac` for the float variant, used in `\xintfloatexpr`.

In order to evaluate binomial coefficients $\binom{x}{y}$ with $x > 99999999$, or even $x \geq 2^{31}$, but y is not too large, one may use an ad hoc function definition such as:

```
\xintdeffunc mybigbinomial(x,y):='*(x-y+1..[1]..x)//y!;%
%
% without [1], x would have been limited to < 2^31
\printnumber{\xinttheexpr mybigbinomial(98765432109876543210,10)\relax}
24338098741940755592729533173058146177070669479669793038510211146784065843698581878582323710
27360575372715482389633359878460739973726786576925067784100587971261422326652270975592667517
4871960261
```

To get this functionality in macro form, one can do:

```
\xintNewIIExpr\MyBigBinomial [2]{`*(#1-#2+1..[1]..#1)//#2!}
\printnumber{\MyBigBinomial {98765432109876543210}{10}}
24338098741940755592729533173058146177070669479669793038510211146784065843698581878582323710
27360575372715482389633359878460739973726786576925067784100587971261422326652270975592667517
4871960261
```

As we used `\xintNewIIExpr`, this macro will only accept strict integers. Had we used `\xintNewExpr` the `\MyBigBinomial` would have accepted general fractions or decimal numbers, and computed the product at the numerator without truncating them to integers; but the factorial at the denominator would truncate its argument.

8.38 `\xintiiPFactorial`

$\frac{num}{x} \frac{num}{x} \star$ `\xintiiPFactorial{a}{b}` computes the partial factorial $(a+1)(a+2)\dots b$. For $a=b$ the product is considered empty hence returns 1.

The allowed range is $-100000000 \leq a, b \leq 99999999$. The rule is to interpret the formula as the product of the j 's such that $a < j \leq b$, hence in particular if $a \geq b$ the product is empty and the macro evaluates to 1.

Only for $0 \leq a \leq b$ is the behaviour to be considered stable. For $a > b$ or negative arguments, the definitive rules have not yet been fixed.

```
\xintiiPFactorial {100}{130}
69293021885203871012298422845822803287591970060789350400000000
```

This theoretical range allows computations whose result values would have more than the roughly 19950 digits that the arithmetics of [xint](#) can handle. In such cases, the computation will end up in a low-level TeX error after a long time.

The `pfactorial` function is available in the `xintexpr` parsers.

```
\xinttheiexpr pfactorial(100,130)\relax
69293021885203871012298422845822803287591970060789350400000000
```

See `\xintFloatPFactorial` from package `xintfrac` for the float variant, used in `\xintfloatexpr`.

In case values are needed with $b > 99999999$, or even $b \geq 2^{31}$, but $b - a$ is not too large, one may use an ad hoc function definition such as:

```
\xintdeffunc mybigpfac(a,b):='*(a+1..[1]..b);%
%
% without [1], b would have been limited to < 2^31
```

```
\printnumber{\xinttheexpr mybigpfac(98765432100,98765432120)\relax}
780008550175675280672981073130237784386530020290496474672081960281164994340505876568704893222
996306044822368535664039125614499125874046078441040781214726754618154427340986762834500699332
322948600573016997034009566576640000
```

8.39 `\xintiiMax`

ff ★ `\xintiiMax{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):
`\xintiiMax {-5}{-6}=-5`.

8.40 `\xintiiMin`

ff ★ `\xintiiMin{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): `\xintiiMin {-5}{-6}=-6`.

8.41 `\xintiiMaxof`

f → **f* ★ `\xintiiMaxof{{a}{b}{c}...}` returns the maximum. The list argument may be a macro, it is *f*-expanded first.

8.42 `\xintiiMinof`

f → **f* ★ `\xintiiMinof{{a}{b}{c}...}` returns the minimum. The list argument may be a macro, it is *f*-expanded first.

8.43 `\xintifTrueAelseB`

fnn ★ `\xintifTrueAelseB{f}{true branch}{false branch}` is a synonym for `\xintiiifNotZero`.
`\xintiiifnotzero` is lowercase companion macro.

Note 1: as it does only *f*-expansion on its argument it fails with inputs such as `--0`. But with `xintfrac` loaded, it does work fine if nested with other `xintfrac` macros, because the output format of such macros is fine as input to `\xintiiifNotZero`. This remark applies to all other «Boolean logic» macros next.

Note 2: prior to 1.20 this macro was using `\xintifNotZero` which applies `\xintNum` to its argument (or gets redefined by `xintfrac` to handle general decimal numbers or fractions). Hence it would have worked with input such as `--0`. But it was decided at 1.20 that the overhead was not worth it. The same remark applies to the other «Boolean logic» type macros next.

8.44 `\xintifFalseAelseB`

fnn ★ `\xintifFalseAelseB{f}{false branch}{true branch}` is a synonym for `\xintiiifZero`.
`\xintiiifzero` is lowercase companion macro.

8.45 `\xintNOT`

f ★ `\xintNOT` is a synonym for `\xintiiIsZero`.
`\xintiiiszero` serves as lowercase companion macro.

8.46 `\xintAND`

ff ★ `\xintAND{f}{g}` returns 1 if `f!=0` and `g!=0` and 0 otherwise.

8.47 `\xintOR`

ff ★ `\xintOR{f}{g}` returns 1 if $f \neq 0$ or $g \neq 0$ and 0 otherwise.

8.48 `\xintXOR`

ff ★ `\xintXOR{f}{g}` returns 1 if exactly one of f or g is true (i.e. non-zero), else 0.

8.49 `\xintANDof`

f → **f* ★ `\xintANDof{a}{b}{c}...` returns 1 if all are true (i.e. non zero) and 0 otherwise. The list argument may be a macro, it (or rather its first token) is *f-expanded* first to deliver its items.

8.50 `\xintORof`

f → **f* ★ `\xintORof{a}{b}{c}...` returns 1 if at least one is true (i.e. does not vanish), else it produces 0. The list argument may be a macro, it is *f-expanded* first.

8.51 `\xintXORof`

f → **f* ★ `\xintXORof{a}{b}{c}...` returns 1 if an odd number of them are true (i.e. do not vanish), else it produces 0. The list argument may be a macro, it is *f-expanded* first.

8.52 `\xintLen`

^{Num}
f ★ `\xintLen` is originally an alias for `\xintiLen`. But with *xintfrac* loaded its meaning is modified to accept more general inputs.

8.53 Removed macros (they require *xintfrac*)

Changed at 1.3! These macros now require *xintfrac*. They have been removed from xint at 1.3. `\xintEq`, `\xintNeq`, `\xintGeq`, `\xintGt`, `\xintLt`, `\xintGtorEq`, `\xintLtorEq`, `\xintIsZero`, `\xintIsNotZero`, `\xintIsOne`, `\xintOdd`, `\xintEven`, `\xintifSgn`, `\xintifCmp`, `\xintifEq`, `\xintifGt`, `\xintifLt`, `\xintifZero`, `\xintifNotZero`, `\xintifOne`, `\xintifOdd`.

With the exception of `\xintNeq` which was renamed to `\xintNotEq`, the above listed macros all belong to *xintfrac*.

8.54 Removed macros (they used `\xintNum`)

Changed at 1.3! These macros filtered their arguments via `\xintNum`. They got deprecated at 1.20 and removed at 1.3: `\xintMON`, `\xintMMON`, `\xintiMax`, `\xintiMin`, `\xintiMaxof`, `\xintiMinof`, `\xintiSquareRoot`, `\xintiSqrt`, `\xintiSqrtR`, `\xintiBinomial`, `\xintiPFactorial`.

All randomness related macros are Work-In-Progress: implementation and user interface may change. They work only if the TeX engine provides the `\uniformdeviate` or `\pdfuniformdeviate` primitive. See `\xintUniformDeviate` for additional information.

8.55 (WIP) `\xintRandomDigits`

^{num}
x ★ `\xintRandomDigits{N}` expands in two steps to N random decimal digits. The argument must be non-negative and is limited by TeX memory parameters. On TeXLive 2018 with input save stack size at 5000 the maximal allowed N is at most 19984 (tested within a `\write` to an auxiliary file, the macro context may cause a reduced maximum).

9 Macros of the xintfrac package

First version of this package was in release 1.03 (2013/04/14) of the *xint* bundle.

Changed
at 1.3!

At release 1.3 (2018/02/28) the behaviour of `\xintAdd` (and of `\xintSub`) was modified: when adding a/b and c/d they will use always the least common multiple of the denominators. This helps limit the build-up of denominators, but the author still hesitates if the fraction should be reduced to smallest terms. The current method allows (for example when multiplying two polynomials) to keep a well-predictable denominator among various terms, even though some may be reducible.

.1 <code>\xintNum</code> 102	.44 <code>\xintifGt</code> 111
.2 <code>\xintRaw</code> 102	.45 <code>\xintifLt</code> 111
.3 <code>\xintNumerator</code> 102	.46 <code>\xintifInt</code> 111
.4 <code>\xintDenominator</code> 103	.47 <code>\xintSgn</code> 111
.5 <code>\xintRawWithZeros</code> 103	.48 <code>\xintOpp</code> 111
.6 <code>\xintREZ</code> 103	.49 <code>\xintAbs</code> 111
.7 <code>\xintIrr</code> 103	.50 <code>\xintAdd</code> 112
.8 <code>\xintPIrr</code> 103	.51 <code>\xintSub</code> 112
.9 <code>\xintJrr</code> 104	.52 <code>\xintMul</code> 112
.10 <code>\xintPRaw</code> 104	.53 <code>\xintDiv</code> 112
.11 <code>\xintDecToString</code> 104	.54 <code>\xintDivFloor</code> 112
.12 <code>\xintTrunc</code> 104	.55 <code>\xintMod</code> 112
.13 <code>\xintXTrunc</code> 105	.56 <code>\xintDivMod</code> 112
.14 <code>\xintTFrac</code> 108	.57 <code>\xintDivTrunc</code> 112
.15 <code>\xintRound</code> 108	.58 <code>\xintModTrunc</code> 112
.16 <code>\xintFloor</code> 108	.59 <code>\xintDivRound</code> 113
.17 <code>\xintCeil</code> 108	.60 <code>\xintSqr</code> 113
.18 <code>\xintiTrunc</code> 108	.61 <code>\xintPow</code> 113
.19 <code>\xintTTrunc</code> 109	.62 <code>\xintFac</code> 113
.20 <code>\xintiRound</code> 109	.63 <code>\xintBinomial</code> 113
.21 <code>\xintiFloor</code> 109	.64 <code>\xintPFactorial</code> 113
.22 <code>\xintiCeil</code> 109	.65 <code>\xintMax</code> 113
.23 <code>\xintE</code> 109	.66 <code>\xintMin</code> 113
.24 <code>\xintCmp</code> 109	.67 <code>\xintMaxof</code> 114
.25 <code>\xintEq</code> 109	.68 <code>\xintMinof</code> 114
.26 <code>\xintNotEq</code> 109	.69 <code>\xintSum</code> 114
.27 <code>\xintGeq</code> 110	.70 <code>\xintPrd</code> 114
.28 <code>\xintGt</code> 110	.71 <code>\xintDigits</code> , <code>\xinttheDigits</code> 114
.29 <code>\xintLt</code> 110	.72 <code>\xintFloat</code> 114
.30 <code>\xintGtorEq</code> 110	.73 <code>\xintPFloat</code> 116
.31 <code>\xintLtorEq</code> 110	.74 <code>\xintFloatE</code> 117
.32 <code>\xintIsZero</code> 110	.75 <code>\xintFloatAdd</code> 117
.33 <code>\xintIsNotZero</code> 110	.76 <code>\xintFloatSub</code> 117
.34 <code>\xintIsOne</code> 110	.77 <code>\xintFloatMul</code> 117
.35 <code>\xintOdd</code> 110	.78 <code>\xintFloatDiv</code> 117
.36 <code>\xintEven</code> 110	.79 <code>\xintFloatPow</code> 117
.37 <code>\xintifSgn</code> 110	.80 <code>\xintFloatPower</code> 118
.38 <code>\xintifZero</code> 110	.81 <code>\xintFloatSqrt</code> 118
.39 <code>\xintifNotZero</code> 111	.82 <code>\xintFloatFac</code> 119
.40 <code>\xintifOne</code> 111	.83 <code>\xintFloatBinomial</code> 119
.41 <code>\xintifOdd</code> 111	.84 <code>\xintFloatPFactorial</code> 120
.42 <code>\xintifCmp</code> 111	.85 <code>\xintFrac</code> 120
.43 <code>\xintifEq</code> 111	.86 <code>\xintSignedFrac</code> 120

.87 \xintFwOver..... 120	.89 \xintLen..... 121
.88 \xintSignedFwOver..... 120	

`xintfrac` loads automatically *xintcore* and *xint* and inherits their macro definitions. Only these two are redefined: `\xintNum` and `\xintLen`. As explained in [subsection 3.4](#) and [subsection 3.5](#) the interchange format for the `xintfrac` macros, i.e. $A/B[N]$, is not understood by the *ii*-named macros of *xintcore*/*xint* which expect the so-called strict integer format. Hence, to use such an *ii*-macro with an output from an `xintfrac` macro, an extra `\xintNum` wrapper is required. But macros already defined by `xintfrac` cover most use cases hence this should be a rarely needed.

Frac *f* In the macro descriptions, the variable *f* and the margin indicator stand for the `xintfrac` input format for integers, scientific numbers, and fractions as described in [subsection 3.4](#).

num *x* As in the *xint.sty* documentation, *x* stands for something which internally will be handled in a `\numexpr`. It may thus be an expression as understood by `\numexpr` but its evaluation and intermediate steps must obey the \TeX bound.

The output format for most macros is the $A/B[N]$ format but naturally the float macros use the scientific notation on output. And some macros are special, for example `\xintTrunc` produces decimal numbers, `\xintIrr` produces an A/B with no $[N]$, `\xintiTrunc` and `\xintiRound` produce integers without trailing $[N]$ either, etc...

1.3a belatedly adds documentation for some macros such as `\xintDivFloor` which had been defined long ago, but did not make it to the user manual for various reasons, one being that it is thought few users will use directly the `xintfrac` macros, the `\xintexpr` interface being more convenient. For complete documentation refer to [sourcexint.pdf](#).

9.1 `\xintNum`

Frac *f* ★ The original `\xintNum` from *xint* is made a synonym to `\xintTTrunc` (whose description is to be found farther in this section).

Attention that for example `\xintNum{1e100000}` expands to the needed 100001 digits...

The original `\xintNum` from *xintcore* which does not understand the fraction slash or the scientific notation is still available under the name `\xintiNum`.

9.2 `\xintRaw`

Frac *f* ★ This macro 'prints' the fraction *f* as it is received by the package after its parsing and expansion, in a form $A/B[N]$ equivalent to the internal representation: the denominator *B* is always strictly positive and is printed even if it has value 1.

```
\xintRaw{\the\numexpr 571*987\relax.123e-10/\the\numexpr-201+59\relax e-7}
-563577123/142[-6]
No simplification is done, not even of common zeroes between numerator and denominator:
\xintRaw {178000/25600000}
178000/25600000[0]
```

9.3 `\xintNumerator`

Frac *f* ★ The input data is parsed as if by `\xintRaw` into $A/B[N]$ format and the macro outputs *A* if $N \leq 0$, or *A* extended by *N* zeroes if $N > 0$.

```
\xintNumerator {178000/25600000[17]}\newline
\xintNumerator {312.289001/20198.27}\newline
\xintNumerator {178000e-3/256e5}\newline
\xintNumerator {178.000/25600000}
17800000000000000000000000
312289001
178000
178000
```

9.4 `\xintDenominator`

Frac
f ★ The input data is parsed as if by `\xintRaw` into $A/B[N]$ format and the macro outputs B if $N>0$, or B extended by $|N|$ zeroes if $N\leq 0$.

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

```
25600000
20198270000
25600000000
25600000000
```

9.5 `\xintRawWithZeros`

Frac
f ★ This macro parses the input and outputs A/B , with A as would be returned by `\xintNumerator{f}` and B as would be returned by `\xintDenominator{f}`.

```
\xintRawWithZeros{178000/25600000[17]}\newline
\xintRawWithZeros{312.289001/20198.27}\newline
\xintRawWithZeros{178000e-3/256e5}\newline
\xintRawWithZeros{178.000/25600000}\newline
\xintRawWithZeros{\the\numexpr 571*987\relax.123e-10/\the\numexpr-201+59\relax e-7}
```

```
1780000000000000000000/25600000
312289001/20198270000
178000/25600000000
178000/25600000000
-563577123/142000000
```

9.6 `\xintREZ`

Frac
f ★ The input is first parsed into $A/B[N]$ as by `\xintRaw`, then trailing zeroes of A and B are suppressed and N is accordingly adjusted.

```
\xintREZ {178000/25600000[17]}
```

```
178/256[15]
```

This macro is used internally by various other constructs; its implementation was redone entirely at 1.3a, and it got faster on long inputs.

9.7 `\xintIrr`

Frac
f ★ This puts the fraction into its unique irreducible form:

```
\xintIrr {178.256/256.1780}, \xintIrr {178000/25600000[17]}
```

```
6856/9853, 695312500000000/1
```

The current implementation does not cleverly first factor powers of 2 and 5, and `\xintIrr {2/3}[100]` will execute the Euclidean division of $2 \cdot 10^{100}$ by 3, which is a bit stupid as it could have known that the 100 trailing zeros can not bring any divisibility by 3.

Starting with release 1.08, `\xintIrr` does not remove the trailing /1 when the output is an integer. This was deemed better for various (questionable?) reasons, anyway the output format is since always A/B with $B>0$, even in cases where it turns out that $B=1$. Use `\xintPraw` on top of `\xintIrr` if it is needed to get rid of such a trailing /1.

9.8 `\xintPIrr`

Frac
f ★ This puts the fraction into irreducible form, *keeping as is the decimal part [N] from raw internal $A/B[N]$ format.* (P stands here for *Partial*)

1.3 `\xintPIrr {178.256/256.1780}, \xintPIrr {178000/25600000[17]}`

3428/49265[1], 89/12800[17]

Notice that the output always has the ending [N], which is exactly the opposite of `\xintIrr`'s behaviour. The interest of this macro is mainly in handling fractions which somehow acquired a big [N] (perhaps from input in scientific notation) and for which the reduced fraction would have a very large number of digits. This large number of digits can considerably slow-down computations done afterwards.

For example package `poexpr` uses `\xintPIrr` when differentiating a polynomial, or in setting up a Sturm chain for localization of the real roots of a polynomial. This is relevant to polynomials whose coefficients were input in decimal notation, as this automatically creates internally some [N]. Keeping and combining those [N]'s during computations significantly increases their speed.

9.9 `\xintJrr`

$\frac{f}{f}$ ★ This also puts the fraction into its unique irreducible form:
`\xintJrr {178.256/256.178}`
 6856/9853

This is (supposedly, not tested for ages) faster than `\xintIrr` for fractions having some big common factor in the numerator and the denominator.

`\xintJrr {\xintiiPow{\xintiiFac {15}}{3}}/%`
`\xintiiPrd{\xintiiFac{10}}{\xintiiFac{30}}{\xintiiFac{5}}}`
 1001/51705840

But to notice the difference one would need computations with much bigger numbers than in this example. As `\xintIrr`, `\xintJrr` does not remove the trailing /1 from a fraction reduced to an integer.

9.10 `\xintPRaw`

$\frac{f}{f}$ ★ `PRaw` stands for ``pretty raw''. It does like `\xintRaw` apart from removing the [N] part if `N=0` and removing the B if `B=1`.
`\xintPRaw {123e10/321e10}, \xintPRaw {123e9/321e10}, \xintPRaw {\xintIrr{861/123}}`
 123/321, 123/321[-1], 7

9.11 `\xintDecToString`

$\frac{f}{f}$ ★ This is a macro tailored for printing decimal numbers. It does not trim trailing zeros, use `\xintDecToString{\xintREZ{<foo>}}` for that.

New with 1.3
`\xintDecToString {123456789e5}\newline`
`\xintDecToString {123456789e-5}\newline`
`\xintDecToString {12345e-10}\newline`
`\xintDecToString {12345e-10/123}\par % just leave denominator as is`
 12345678900000
 1234.56789
 0.0000012345
 0.0000012345/123

Consider it an unstable macro, what it does exactly is yet to be decided. It is a backport from `poexpr`'s `\PolDecToString`, which has now been made an alias to it.

9.12 `\xintTrunc`

$\frac{\text{num}}{x} \frac{f}{f}$ ★ `\xintTrunc{x}{f}` returns the integral part, a dot (standing for the decimal mark), and then the first x digits of the decimal expansion of the fraction f, except when the fraction is (or evaluates to) zero, then it simply prints 0 (with no dot).

The argument `x` must be non-negative, the behaviour is currently undefined when `x<0` and will provoke errors.

Except when the input is (or evaluates to) exactly zero, the output contains exactly `x` digits after the decimal mark, thus the output may be `0.00...0` or `-0.00...0`, indicating that the original fraction was positive, respectively negative.

Warning: *it is not yet decided is this behaviour is definitive.*

Currently `xintfrac` has no notion of a positive zero or a negative zero. Hence transitivity of `\xintTrunc` is broken for the case where the first truncation gives on output `0.00...0` or `-0.00...0`: a second truncation to less digits will then output `0`, whereas if it had been applied directly to the initial input it would have produced `0.00...0` or respectively `-0.00...0` (with less zeros).

If `xintfrac` distinguished zero, positive zero, and negative zero it would be possible to maintain transitivity.

The problem would also be fixed, even without distinguishing a negative zero on input, if `\xintTrunc` always produced `0.00...0` (with no sign) when the mathematical result is zero, discarding the information on original input being positive, zero, or negative.

I have multiple times hesitated about what to do and must postpone again final decision.

```
\xintTrunc {16}{-803.2028/20905.298}\newline
\xintTrunc {20}{-803.2028/20905.298}\newline
\xintTrunc {10}{\xintPow {-11}{-11}}\newline
\xintTrunc {12}{\xintPow {-11}{-11}}\newline
\xintTrunc {50}{\xintPow {-11}{-11}}\newline
\xintTrunc {12}{\xintAdd {-1/3}{3/9}}\par
-0.0384210165289200
-0.03842101652892008523
-0.0000000000
-0.000000000003
-0.00000000000350493899481392497604003313162598556370
0
```

The digits printed are exact up to and including the last one.

9.13 `\xintXTrunc`

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

`\xintXTrunc{x}{f}` is similar to `\xintTrunc` with the following important differences:

- it is completely expandable but not *f*-expandable, as is indicated by the hollow star in the margin,
- hence it can not be used as argument to the other package macros, but as it *f*-expands its `{f}` argument, it accepts arguments expressed with other `xintfrac` macros,
- it requires `x>0`,
- contrarily to `\xintTrunc` the number of digits on output is not limited to about 19950 and may go well beyond 100000 (this is mainly useful for outputting a decimal expansion to a file),
- when the mathematical result is zero, it always prints it as `0.00...0` or `-0.00...0` with `x` zeros after the decimal mark.

Warning: transitivity is broken too (see discussion of `\xintTrunc`), due to the sign in the last item. Hence *the definitive policy is yet to be fixed*.

Transitivity is here in the sense of using a first `\edef` and then a second one, because it is not possible to nest `\xintXTrunc` directly as argument to itself. Besides, although the number of digits on output isn't limited, nevertheless `x` should be less than about 19970 when the number of

digits of the input (assuming it is expressed as a decimal number) is even bigger: `\xintXTrunc{30000}{Z}` after `\edef Z{\xintXTrunc{60000}{1/66049}}` raises an error in contrast with a direct `\xintXTrunc{30000}{1/66049}`. But `\xintXTrunc{30000}{123.456789}` works, because here the number of digits originally present is smaller than what is asked for, thus the routine only has to add trailing zeros, and this has no limitation (apart from TeX main memory).

`\xintXTrunc` will expand fully in an `\edef` or a `\write` (`\message`, `\wlog`, ...) or in an `\xintexpr`-expression, or as list argument to `\xintFor`.

Here is an example session where the user checks that the decimal expansion of $1/66049 = 1/257^2$ has the maximal period length $257 * 256 = 65792$ (this period length must be a divisor of $\phi(66049)$ and to check it is the maximal one it is enough to show that neither 32896 nor 256 are periods.)

```
$ rlwrap etex -jobname worksheet-66049
This is pdfTeX, Version 3.14159265-2.6-1.40.17 (TeX Live 2016) (preloaded format=etex)
restricted \writel8 enabled.
**xintfrac.sty
entering extended mode
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xintfrac.sty
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xint.sty
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xintcore.sty
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xintkernel.sty)))
%% we load xinttools for \xintKeep, etc... \xintXTrunc itself has no more

%% any dependency on xinttools.sty since 1.2i

*\input xinttools.sty
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xinttools.sty)
*\def\m#1;{\message{#1}}

*\m \the\numexpr 257*257\relax;
66049
*\m \the\numexpr 257*256\relax;
65792
%% Thus 1/66049 will have a period length dividing 65792.

%% Let us first check it is indeed periodical.

*\edef Z{\xintXTrunc{66000}{1/66049}}

%% Let's display the first decimal digits.

*\m \xintXTrunc{208}{Z};

0.00001514027464458205271843631243470756559523989765174340262532362337052794137
6856576178291874214598252812306015231116292449545034746930309315810988811337037
6538630410755651107511090251177156353616254598858423
%% let's now fetch the trailing digits

*\m \xintKeep{65792-66000}{Z};% 208 trailing digits

0000151402746445820527184363124347075655952398976517434026253236233705279413768
5657617829187421459825281230601523111629244954503474693030931581098881133703765
38630410755651107511090251177156353616254598858423
%% yes they match! we now check that 65792/2 and 65792/257=256 aren't periods.

*\m \xintXTrunc{256}{Z};

0.00001514027464458205271843631243470756559523989765174340262532362337052794137
```



```

6856576178291874214598252812306015231116292449545034746930309315810988811337037
6538630410755651107511090251177156353616254598858423291798513225029902042423049
554118911717058547442
*\m \xintXTrunc{256+256}{\Z};

0.00001514027464458205271843631243470756559523989765174340262532362337052794137
6856576178291874214598252812306015231116292449545034746930309315810988811337037
6538630410755651107511090251177156353616254598858423291798513225029902042423049
5541189117170585474420505987978621932201850141561567926842192917379521264515738
3154930430438008145467758785144362518736089872670290239064936637950612424109373
3440324607488379839210283274538600130206361943405653378552286938485064119063119
8049932625777831609865402958409665551333
% now with 65792/2=32896. Problem: we can't do \xintXTrunc{32896+100}{\Z}

% but only direct \xintXTrunc{32896+100}{1/66049}. Anyway we want to nest it

% hence let's do it all with (slower) \xintKeep, \xintKeepUnbraced.

*\m \xintKeep {-100}{\xintKeepUnbraced{2+65792/2+100}{\Z}};

9999848597253554179472815636875652924344047601023482565973746763766294720586231
434238217081257854017
% This confirms 32896 isn't a period length.

% To conclude let's write the 66000 digits to the log.

*\wlog{\Z}

% We want always more digits:

*\wlog{\xintXTrunc{150000}{1/66049}}

*\bye

```

The acute observer will have noticed that there is something funny when one compares the first digits with those after the middle-period:

```

0000151402746445820527184363124347075655952398976517434026253236233705279413768...
9999848597253554179472815636875652924344047601023482565973746763766294720586231...

```

Mathematical exercise: can you explain why the two indeed add to 9999...9999?

You can try your hands at this simpler one:

```

1/49=\xintTrunc{42+5}{1/49}...\newline
\xintTrim{2}{\xintTrunc{21}{1/49}}\newline
\xintKeep{-21}{\xintTrunc{42}{1/49}}

```

1/49=0.02040816326530612244897959183673469387755102040...

020408163265306122448

979591836734693877551

This was again an example of the type $1/N$ with N the square of a prime. One can also find counter-examples within this class: $1/31^2$ and $1/37^2$ have an odd period length (465 and respectively 111) hence they can not exhibit the symmetry.

Mathematical challenge: prove generally that if the period length of the decimal expansion of $1/p^r$ (with p a prime distinct from 2 and 5 and r a positive exponent) is even, then the previously observed symmetry about the two halves of the period adding to a string of nine's applies.

9.14 `\xintTFrac`

$\frac{\text{Frac}}{f}$ ★ `\xintTFrac{f}` returns the fractional part, $f = \text{trunc}(f) + \text{frac}(f)$. Thus if $f < 0$, then $-1 < \text{frac}(f) \leq 0$ and if $f > 0$ one has $0 \leq \text{frac}(f) < 1$. The T stands for 'Trunc', and there should exist also similar macros associated respectively with 'Round', 'Floor', and 'Ceil', each type of rounding to an integer deserving arguably to be associated with a fractional 'modulo'. By sheer laziness, the package currently implements only the 'modulo' associated with 'Truncation'. Other types of modulo may be obtained more clumsily via a combination of the rounding with a subsequent subtraction from f .

Notice that the result is filtered through `\xintREZ`, and will thus be of the form $A/B[N]$, where neither A nor B has trailing zeros. But the output fraction is not reduced to smallest terms.

The function call in expressions (`\xintexpr`, `\xintfloatexpr`) is `frac`. Inside `\xintexpr.\relax` x , the function `frac` is mapped to `\xintTFrac`. Inside `\xintfloatexpr.\relax`, `frac` first applies `\xintTFrac` to its argument (which may be an exact fraction with more digits than the floating point precision) and only in a second stage makes the conversion to a floating point number with the precision as set by `\xintDigits` (default is 16).

```
\xintTFrac {1235/97}, \xintTFrac {-1235/97}\newline
\xintTFrac {1235.973}, \xintTFrac {-1235.973}\newline
\xintTFrac {1.122435727e5}\par
```

```
71/97[0], -71/97[0]
973/1[-3], -973/1[-3]
5727/1[-4]
```

9.15 `\xintRound`

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

```
\xintRound {16}{-803.2028/20905.298}\newline
\xintRound {20}{-803.2028/20905.298}\newline
\xintRound {10}{\xintPow {-11}{-11}}\newline
\xintRound {12}{\xintPow {-11}{-11}}\newline
\xintRound {12}{\xintAdd {-1/3}{3/9}}\par
```

```
-0.0384210165289201
-0.03842101652892008523
-0.0000000000
-0.000000000004
0
```

9.16 `\xintFloor`

$\frac{\text{Frac}}{f}$ ★ `\xintFloor {f}` returns the largest relative integer N with $N \leq f$.

```
\xintFloor {-2.13}, \xintFloor {-2}, \xintFloor {2.13}
```

```
-3/1[0], -2/1[0], 2/1[0] Note the trailing [0], see \xintiFloor if it is not desired.
```

9.17 `\xintCeil`

$\frac{\text{Frac}}{f}$ ★ `\xintCeil {f}` returns the smallest relative integer N with $N > f$.

```
\xintCeil {-2.13}, \xintCeil {-2}, \xintCeil {2.13}
```

```
-2/1[0], -2/1[0], 3/1[0]
```

9.18 `\xintiTrunc`

$\frac{\text{num}}{x} \frac{\text{Frac}}{f}$ ★ `\xintiTrunc{x}{f}` returns the integer equal to 10^x times what `\xintTrunc{x}{f}` would produce.

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

In particular `\xintiTrunc{0}{f}`'s output is in strict integer format contrarily to `\xintTrunc{0}{f}` which produces an output with a decimal mark, except if `f` turns out to be zero.

9.19 `\xintTTrunc`

$\frac{\text{Frac}}{f}$ ★ `\xintTTrunc{f}` truncates to an integer (truncation towards zero). This is the same as `\xintiTrunc{0}{f}` and also the same as `\xintNum`.

9.20 `\xintiRound`

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

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

In particular `\xintiRound{0}{f}`'s output is in strict integer format contrarily to `\xintRound{0}{f}` which produces an output with a decimal mark, except if `f` turns out to be zero.

9.21 `\xintiFloor`

$\frac{\text{Frac}}{f}$ ★ `\xintiFloor {f}` does the same as `\xintFloor` but without the trailing `/1[0]`.

```
\xintiFloor {-2.13}, \xintiFloor {-2}, \xintiFloor {2.13}
-3, -2, 2
```

9.22 `\xintiCeil`

$\frac{\text{Frac}}{f}$ ★ `\xintiCeil {f}` does the same as `\xintCeil` but its output is without the `/1[0]`.

```
\xintiCeil {-2.13}, \xintiCeil {-2}, \xintiCeil {2.13}
-2, -2, 3
```

9.23 `\xintE`

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

```
\count 255 123456789 \xintE {10}{\count 255}
10/1[123456789] Don't feed this example to \xintNum!
```

9.24 `\xintCmp`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★ This compares two fractions `F` and `G` and produces `-1`, `0`, or `1` according to `F<G`, `F=G`, `F>G`. For choosing branches according to the result of comparing `f` and `g`, see `\xintifCmp`.

9.25 `\xintEq`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★ `\xintEq{f}{g}` returns 1 if `f=g`, 0 otherwise.

9.26 `\xintNotEq`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★ `\xintNotEq{f}{g}` returns 0 if `f=g`, 1 otherwise.

9.27 `\xintGeq`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ This compares the *absolute values* of two fractions. `\xintGeq{f}{g}` outputs 1 if $|f| \geq |g|$ and 0 if not.
Important: the macro compares *absolute values*.

9.28 `\xintGt`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ `\xintGt{f}{g}` returns 1 if $f > g$, 0 otherwise.

9.29 `\xintLt`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ `\xintLt{f}{g}` returns 1 if $f < g$, 0 otherwise.

9.30 `\xintGtorEq`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ `\xintGtorEq{f}{g}` returns 1 if $f \geq g$, 0 otherwise. Extended by `xintfrac` to fractions.

9.31 `\xintLtorEq`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ `\xintLtorEq{f}{g}` returns 1 if $f \leq g$, 0 otherwise.

9.32 `\xintIsZero`

f ★ `\xintIsZero{f}` returns 1 if $f=0$, 0 otherwise.

9.33 `\xintIsNotZero`

f ★ `\xintIsNotZero{f}` returns 1 if $f \neq 0$, 0 otherwise.

9.34 `\xintIsOne`

f ★ `\xintIsOne{f}` returns 1 if $f=1$, 0 otherwise.

9.35 `\xintOdd`

f ★ `\xintOdd{f}` returns 1 if the integer obtained by truncation is odd, and 0 otherwise.

9.36 `\xintEven`

f ★ `\xintEven{f}` returns 1 if the integer obtained by truncation is even, and 0 otherwise.

9.37 `\xintifSgn`

$\frac{Frac}{f} nnn$ ★ `\xintifSgn{f}{A}{B}{C}` executes either the $\langle A \rangle$, $\langle B \rangle$ or $\langle C \rangle$ code, depending on its first argument being respectively negative, zero, or positive.

9.38 `\xintifZero`

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

9.39 `\xintifNotZero`

$\frac{\text{Frac}}{f} nn \star$ `\xintifNotZero{⟨N⟩}{⟨IsNotZero⟩}{⟨IsZero⟩}` expandably checks if the first mandatory argument `f` is not zero or is zero. It then either executes the first or the second branch.
Beware that both branches must be present.

9.40 `\xintifOne`

$\frac{\text{Frac}}{f} nn \star$ `\xintifOne{⟨N⟩}{⟨IsOne⟩}{⟨IsNotOne⟩}` expandably checks if the first mandatory argument `f` is one or not one. It then either executes the first or the second branch. Beware that both branches must be present.

9.41 `\xintifOdd`

$\frac{\text{Frac}}{f} nn \star$ `\xintifOdd{⟨N⟩}{⟨odd⟩}{⟨not odd⟩}` expandably checks if the first mandatory argument `f`, after truncation to an integer, is odd or even. It then executes accordingly the first or the second branch. Beware that both branches must be present.

9.42 `\xintifCmp`

$\frac{\text{rac}}{f} \frac{\text{Frac}}{f} nnn \star$ `\xintifCmp{⟨f⟩}{⟨g⟩}{⟨if f<g⟩}{⟨if f=g⟩}{⟨if f>g⟩}` compares its first two arguments and chooses accordingly the correct branch.

9.43 `\xintifEq`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f} nn \star$ `\xintifEq{⟨f⟩}{⟨g⟩}{⟨YES⟩}{⟨NO⟩}` checks equality of its two first arguments and executes accordingly the `YES` or the `NO` branch.

9.44 `\xintifGt`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f} nn \star$ `\xintifGt{⟨f⟩}{⟨g⟩}{⟨YES⟩}{⟨NO⟩}` checks if `f > g` and in that case executes the `YES` branch.

9.45 `\xintifLt`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f} nn \star$ `\xintifLt{⟨f⟩}{⟨g⟩}{⟨YES⟩}{⟨NO⟩}` checks if `f < g` and in that case executes the `YES` branch.

9.46 `\xintifInt`

$\frac{\text{Frac}}{f} nn \star$ `\xintifInt{f}{YES branch}{NO branch}` expandably chooses the `YES` branch if `f` reveals itself after expansion and simplification to be an integer.

9.47 `\xintSgn`

$\frac{\text{Frac}}{f} \star$ The sign of a fraction.

9.48 `\xintOpp`

$\frac{\text{Frac}}{f} \star$ The opposite of a fraction. Note that `\xintOpp {3}` produces `-3/1[0]` whereas `\xintiiOpp {3}` produces `-3`.

9.49 `\xintAbs`

$\frac{\text{Frac}}{f} \star$ The absolute value. Note that `\xintAbs {-2}=2/1[0]` where `\xintiiAbs {-2}` outputs `=2`.

9.50 `\xintAdd`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ Computes the addition of two fractions.
Changed at 1.3! Since 1.3 always uses the least common multiple of the denominators.

9.51 `\xintSub`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ Computes the difference of two fractions (`\xintSub{F}{G}` computes $F-G$).
Changed at 1.3! Since 1.3 always uses the least common multiple of the denominators.

9.52 `\xintMul`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ Computes the product of two fractions.
 Output is not reduced to smallest terms.

9.53 `\xintDiv`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ Computes the quotient of two fractions. (`\xintDiv{F}{G}` computes F/G).
 Output is not reduced to smallest terms.

9.54 `\xintDivFloor`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ Computes the quotient of two arguments then apply floor function to get an integer (in strict format). This macro was defined at 1.1 (but was left not documented until 1.3a...) and changed at 1.2p, formerly it appended `/1[0]` to output.
`\xintDivFloor{-170/3}{23/2}`
-5

9.55 `\xintMod`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ Computes the remainder associated to the floored division `\xintDivFloor`. Prior to 1.2p the meaning was the one of `\xintModTrunc`. Was left undocumented until 1.3a.
`\xintMod{-170/3}{23/2}`
5/6[0]
Changed at 1.3! Modified at 1.3 to use a l.c.m. for the denominator of the result.

9.56 `\xintDivMod`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ Computes both the floored division and the remainder `\xintDivFloor`. New at 1.2p and documented at 1.3a.
`\oodef\foo{\xintDivMod{-170/3}{23/2}}\meaning\foo`
macro:->{-5}{5/6[0]}

9.57 `\xintDivTrunc`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ Computes the quotient of two arguments then truncates to an integer (in strict format).
`\xintDivTrunc{-170/3}{23/2}`
-4

9.58 `\xintModTrunc`

$\frac{Frac}{f} \frac{Frac}{f}$ ★ Computes the remainder associated with the truncated division of two arguments. Prior to 1.2p it was named `\xintMod`, but the latter then got associated with floored division.
`\xintModTrunc{-170/3}{23/2}`
-64/6[0]
Changed at 1.3! Modified at 1.3 to use a l.c.m. for the denominator of the result.

9.59 `\xintDivRound`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★ Computes the quotient of the two arguments then rounds to an integer (in strict format).
`\xintDivRound{-170/3}{23/2}`
 -5

9.60 `\xintSqr`

$\frac{\text{Frac}}{f}$ ★ Computes the square of one fraction.

9.61 `\xintPow`

$\frac{\text{Frac}}{f} \frac{\text{Num}}{f}$ ★ `\xintPow{f}{x}`: computes f^x with f a fraction and x possibly also, but x will first get truncated to a (positive or negative) integer.

The exponent x must obey the TeX-bound, but this limit is theoretical, as TeX's memory or expansion settings get saturated quite earlier: it is explained in the documentation of `\xintiiPow` that the maximal power of 2 computable by `xint` is 2^{131072} which has 39457 digits. Actually, the practical range is even smaller due to execution times.

The output will always be in the form $A/B[n]$ (even if the exponent vanishes: `\xintPow {2/3}{0}` $=1/1[0]$).

Within an `\xintiiexpr`..`\relax` the infix operator \wedge is mapped to `\xintiiPow`; within an `\xintexpr`-ession it is mapped to `\xintPow`.

9.62 `\xintFac`

$\frac{\text{Num}}{f}$ ★ This is a convenience variant of `\xintiiFac` which applies `\xintNum` to its argument. Notice however that the output will have a trailing $[0]$ according to the `xintfrac` format for integers.

9.63 `\xintBinomial`

$\frac{\text{Num}}{f} \frac{\text{Num}}{f}$ ★ This is a convenience variant of `\xintiiBinomial` which applies `\xintNum` to its arguments. Notice however that the output will have a trailing $[0]$ according to the `xintfrac` format for integers.

9.64 `\xintPFactorial`

$\frac{\text{Num}}{f} \frac{\text{Num}}{f}$ ★ This is a convenience variant of `\xintiiPFactorial` which applies `\xintNum` to its arguments. Notice however that the output will have a trailing $[0]$ according to the `xintfrac` format for integers.

9.65 `\xintMax`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★ The maximum of two fractions. Beware that `\xintMax {2}{3}` produces $3/1[0]$. The original, for use with integers only with no need of normalization, is available as `\xintiiMax`: `\xintiiMax {2}{3}` $=3$.
 ff ★ `\xintMax {2.5}{7.2}`
 $72/1[-1]$

9.66 `\xintMin`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$ ★ The minimum of two fractions. Beware that `\xintMin {2}{3}` produces $2/1[0]$. The original, for use with integers only with no need of normalization, is available as `\xintiiMin`: `\xintiiMin {2}{3}` $=2$.
 ff ★ `\xintMin {2.5}{7.2}`
 $25/1[-1]$

9.67 `\xintMaxof`

$f \rightarrow * \overset{\text{Frac}}{f} \star$ The maximum of any number of fractions, each within braces, and the whole thing within braces.
`\xintMaxof {{1.23}{1.2299}{1.2301}}` and `\xintMaxof {{-1.23}{-1.2299}{-1.2301}}`
`12301/1[-4]` and `-12299/1[-4]`

9.68 `\xintMinof`

$f \rightarrow * \overset{\text{Frac}}{f} \star$ The minimum of any number of fractions, each within braces, and the whole thing within braces.
`\xintMinof {{1.23}{1.2299}{1.2301}}` and `\xintMinof {{-1.23}{-1.2299}{-1.2301}}`
`12299/1[-4]` and `-12301/1[-4]`

9.69 `\xintSum`

$f \rightarrow * \overset{\text{Frac}}{f} \star$ This computes the sum of fractions. The output will now always be in the form `A/B[n]`. The original, for big integers only (in strict format), is available as `\xintiiSum`.
`\xintSum {{1282/2196921}{-281710/291927}{4028/28612}}`
`-5037928302100692/6116678670072468[0]`
 No simplification attempted.

9.70 `\xintPrd`

$f \rightarrow * \overset{\text{Frac}}{f} \star$ This computes the product of fractions. The output will now always be in the form `A/B[n]`. The original, for big integers only (in strict format), is available as `\xintiiPrd`.
`\xintPrd {{1282/2196921}{-281710/291927}{4028/28612}}`
`-1454721142160/18350036010217404[0]`
 No simplification attempted.
`\xintIsOne {21921379213/21921379213}\neq\xintIsOne {1.00000000000000000000000000000001}$`
`1 ≠ 0`

9.71 `\xintDigits`, `\xinttheDigits`

The syntax `\xintDigits := D;` (where spaces do not matter) assigns the value of `D` to the number of digits to be used by floating point operations. The default is `16`. The maximal value is `32767`. The \star macro `\xinttheDigits` serves to print the current value.

9.72 `\xintFloat`

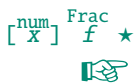
$\left[\overset{\text{num}}{x} \right] \overset{\text{Frac}}{f} \star$ The macro `\xintFloat [P]{f}` has an optional argument `P` which replaces the current value of `\xinttheDigits`. The fraction `f` is then printed in scientific notation with a rounding to `P` digits.
 That is, on output: the first digit is from `1` to `9`, it is possibly prefixed by a minus sign and is followed by a dot and `P-1` digits, then a lower case `e` and an exponent `N`. The trailing zeroes are not trimmed.

There is currently one exceptional case: the zero value, which gets output as `0.e0`. It is yet to be decided what the final policy will be.

Starting with `1.2k`, when the input is a fraction `AeN/BeM` the output always is the *correct rounding* to `P` digits. Formerly, this was guaranteed only when `A` and `B` had at most `P+2` digits, or when `B` was `1` and `A` was arbitrary, but in other cases it was only guaranteed that the difference between the original fraction and the rounding was at most `0.6` unit in the last place (of the output), hence the output could differ in the last digit (and earlier ones in case of chains of zeros or nines) from the correct rounding.

5.000000000000001e-16
 5.00000000000000500000000000001e-16
 5.00000000000000500000000000005000000000001e-16

9.73 `\xintPFloat`



The macro `\xintPFloat [P]{f}` is like `\xintFloat` but ``pretty-prints'' the output. Its behaviour has changed with release 1.2f: there is only one simplification rule now which is that decimal notation (with possibly needed extra zeros) is used in place of scientific notation when the exponent would end up being between -5 and 5 inclusive.

If the input vanishes the output will be `0.` with a a decimal mark.⁶¹

`\xintthefloatexpr` applies this macro to its output (or each of its outputs, if comma separated). Currently trailing zeros are not trimmed.

```
\begingroup\def\test #1{#1$\}\to{$}\xintPFloat{#1}}%
\string\xintDigits\ at \xinttheDigits
\begin{itemize}[nosep]
\item \test {0}
\item \test {1.23456789e-7}
\item \test {1.23456789e-6}
\item \test {1.23456789e-5}
\item \test {1.23456789e-4}
\item \test {1.23456789e-3}
\item \test {1.23456789e-2}
\item \test {1.23456789e-1}
\item \test {1.23456789e0}
\item \test {1.23456789e1}
\item \test {1.23456789e2}
\item \test {1.23456789e3}
\item \test {1.23456789e4}
\item \test {1.23456789e5}
\item \test {1.23456789e6}
\item \test {1.23456789e7}
\end{itemize}
\endgroup
```

`\xintDigits` at 16

- `0` → `0.`
- `1.23456789e-7` → `1.234567890000000e-7`
- `1.23456789e-6` → `1.234567890000000e-6`
- `1.23456789e-5` → `0.0000123456789000000`
- `1.23456789e-4` → `0.0001234567890000000`
- `1.23456789e-3` → `0.0012345678900000000`
- `1.23456789e-2` → `0.01234567890000000`
- `1.23456789e-1` → `0.12345678900000000`
- `1.23456789e0` → `1.234567890000000`
- `1.23456789e1` → `12.345678900000000`
- `1.23456789e2` → `123.4567890000000`
- `1.23456789e3` → `1234.5678900000000`
- `1.23456789e4` → `12345.67890000000`
- `1.23456789e5` → `123456.78900000000`
- `1.23456789e6` → `1.234567890000000e6`
- `1.23456789e7` → `1.234567890000000e7`

⁶¹ Currently there are no subnormal numbers, and no underflow because the exponent is only limited by the maximal T_EX number; thus underflow situations would manifest themselves via low-level arithmetic overflow errors.

9.74 `\xintFloatE`

$\left[\begin{smallmatrix} \text{num} \\ \text{X} \end{smallmatrix} \right] \frac{\text{Frac}}{f} \frac{\text{num}}{X} \star$ `\xintFloatE [P]{f}{x}` multiplies the input `f` by 10^x , and converts it to float format according to the optional first argument or current value of `\xinttheDigits`.
`\xintFloatE {1.23e37}{53}`
 1.2300000000000000e90

9.75 `\xintFloatAdd`

$\left[\begin{smallmatrix} \text{num} \\ \text{X} \end{smallmatrix} \right] \frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$ `\xintFloatAdd [P]{f}{g}` first replaces `f` and `g` with their float approximations `f'` and `g'` to `P` significant places or to the precision from `\xinttheDigits`. It then produces the sum `f'+g'`, correctly rounded to nearest with the same number of significant places.

9.76 `\xintFloatSub`

$\left[\begin{smallmatrix} \text{num} \\ \text{X} \end{smallmatrix} \right] \frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$ `\xintFloatSub [P]{f}{g}` first replaces `f` and `g` with their float approximations `f'` and `g'` to `P` significant places or to the precision from `\xinttheDigits`. It then produces the difference `f'-g'` correctly rounded to nearest `P`-float.

9.77 `\xintFloatMul`

$\left[\begin{smallmatrix} \text{num} \\ \text{X} \end{smallmatrix} \right] \frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$ `\xintFloatMul [P]{f}{g}` first replaces `f` and `g` with their float approximations `f'` and `g'` to `P` (or `\xinttheDigits`) significant places. It then correctly rounds the product `f'*g'` to nearest `P`-float. See [subsection 3.2](#) for more.

It is obviously much needed that the author improves its algorithms to avoid going through the exact $2P$ or $2P-1$ digits before throwing to the waste-bin half of those digits !

9.78 `\xintFloatDiv`

$\left[\begin{smallmatrix} \text{num} \\ \text{X} \end{smallmatrix} \right] \frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$ `\xintFloatDiv [P]{f}{g}` first replaces `f` and `g` with their float approximations `f'` and `g'` to `P` (or `\xinttheDigits`) significant places. It then correctly rounds the fraction `f'/g'` to nearest `P`-float.
 See [subsection 3.2](#) for more.

Notice in the special situation with `f` and `g` integers that `\xintFloatDiv [P]{f}{g}` will not necessarily give the correct rounding of the exact fraction `f/g`. Indeed the macro arguments are each first individually rounded to `P` digits of precision. The correct syntax to get the correctly rounded integer fraction `f/g` is `\xintFloat[P]{f/g}`.

9.79 `\xintFloatPow`

$\left[\begin{smallmatrix} \text{num} \\ \text{X} \end{smallmatrix} \right] \frac{\text{Frac}}{f} \frac{\text{num}}{X} \star$ `\xintFloatPow [P]{f}{x}` uses either the optional argument `P` or in its absence the value of `\xinttheDigits`. It computes a floating approximation to f^x .

The exponent `x` will be handed over to a `\numexpr`, hence count registers are accepted on input for this `x`. And the absolute value $|x|$ must obey the $\text{T}_{\text{E}}\text{X}$ bound.

The argument `f` is first rounded to `P` significant places to give `f'`. The output `Z` is such that the exact f'^x differs from `Z` by an absolute error less than $0.52 \text{ ulp}(Z)$.

`\xintFloatPow [8]{3.1415}{1234567890}`
 1.6122066e613749456

9.80 `\xintFloatPower`

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

`\xintFloatPower[P]{f}{g}` computes a floating point value f^g where the exponent g is not constrained to be at most the \TeX bound 2147483647. It may even be a fraction A/B but must simplify to a (possibly big) integer. The exponent of the *output* however *must* at any rate obey the \TeX bound.

The argument f is first rounded to P significant places to give f' . The output Z is then such that the exact f'^g differs from Z by an absolute error less than $0.52 \text{ ulp}(Z)$.

This is the macro which is used for the \wedge (or $**$) infix operators in `\xintthefloatexpr... \relax`. In this context (but not directly with the macro,) half-integer exponents are allowed. This is handled via an integer power followed by a square-root extraction. The exponent is first rounded to nearest integer or half-integer so that the computation never raises errors (except naturally for negative exponent and zero f .) The $0.52 \text{ ulp}(Z)$ bound applies with half-integer exponents too.

Notice that this is a bound on the distance from f^g to Z , as f always gets rounded to P or `\xinttheDigits` digits. The distance from f^g to Z can be much worse if g is very large. Roughly, when g is negligible compared to 10^P , we get an extra difference of up to about $50g \text{ ulp}(Z)$ which completely dwarfs the $0.52 \text{ ulp}(Z)$. Thus, if f has strictly more than P digits, then the computation must be done with an elevated working precision P' . For example with $g=1000$ we should use $P'=P+6$ to achieve a total error at worst slightly bigger than $0.55 \text{ ulp}(Z)$ after the final rounding from P' to P digits to get Z .

Examples:⁶²

```
\np{\xintFloatPower [8]{3.1415}{3e9}}\newline% Notice that 3e9>2^31
\np{\xintFloatPower [48]{1.1547}{\xintiiPow {2}{35}}}\newline
1.431,772,9 × 101,491,411,192
2.785,837,382,571,371,438,495,789,880,733,698,213,205,183,990,48 × 102,146,424,193
235 = 34359738368 exceeds  $\TeX$ 's bound, but what counts is the exponent of the result which, while
dangerously close to 231 is not quite there yet.
```

With expressions:

```
{\xintDigits:=48;\np{\xintthefloatexpr 1.1547^(2^35)\relax}}
```

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

There is a subtlety here that the 2^{35} will be evaluated as a floating point number but fortunately it only has 11 digits, hence the final evaluation is done with a correct exponent. It would have been safer, and also more efficient to code the above rather as:

```
\xintthefloatexpr 1.1547^{\xintiiexpr 2^35\relax}\relax
Here is an example with 12^16 as exponent, which has 18 digits (=184884258895036416).
{\xintDigits:=12;\np{\xintthefloatexpr (1+1e-8)^{\xintiiexpr 12^16\relax}\relax}}\newline
\np{\xintthefloatexpr (1+1e-8)^{\xintiiexpr 12^16\relax}\relax}\relax\newline
{\xintDigits:=27;\np{\xintthefloatexpr (1+1e-8)^(12^16)\relax}}\newline
{\xintDigits:=48;\np{\xintthefloatexpr (1+1e-8)^(12^16)\relax}}
```

1.879,985,676,69 × 10^{802,942,130}
1.879,985,676,694,948 × 10^{802,942,130}
1.879,985,676,694,948,388,381,844,07 × 10^{802,942,130}
1.879,985,676,694,948,388,381,844,074,802,295,996,746,413,609,97 × 10^{802,942,130}

There is an important difference between `\xintFloatPower[Q]{X}{Y}` and `\xintthefloatexpr[Q] X^Y \relax`: in the former case the computation is done with Q digits or precision,⁶³ whereas with `\xintthefloatexpr[Q]` the evaluation of the expression proceeds with `\xinttheDigits` digits of precision, and the final result is then rounded to Q digits: thus this makes real sense only if used with $Q < \text{\xinttheDigits}$.

9.81 `\xintFloatSqrt`

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

`\xintFloatSqrt[P]{f}` computes a floating point approximation of \sqrt{f} , either using the optional precision P or the value of `\xinttheDigits`.

⁶² `\np` is formatting macro from the <http://ctan.org/pkg/numprint> package. ⁶³ if X and Y themselves stand for some floating point macros with arguments, their respective evaluations obey the precision `\xinttheDigits` or as set optionally in the macro calls themselves.

0.6 ulp(Y).

```
\xintFloatBinomial [24]{3000}{1500}  

 $\binom{3000}{1500} \approx 1.79196793754756005073269e901$ 
```

The associated function in `\xintfloatexpr` is `binomial()`:

```
\xintthefloatexpr binomial(3000,1500)\relax
```

```
1.791967937547560e901
```

The computation is based on the formula $(x-y+1)\dots x/y!$ (here one arranges $y \leq x-y$ naturally).

9.84 `\xintFloatPFactorial`

$\xintFloatPFactorial[P]{x}{y}$ computes the product $(x+1)\dots y$.

The arguments must be integers (they are expanded inside `\numexpr`) and the allowed range is $-100000000 \leq x, y \leq 99999999$. If $x \geq y$ the product is considered empty hence returns one (as a floating point value). See also `\xintiiPFactorial`.

The exact theoretical value differs from the calculated one Y by an absolute error strictly less than 0.6 ulp(Y).

The associated function in `\xintfloatexpr` is `pfactorial()`:

```
\xintthefloatexpr pfactorial(2500,5000)\relax
```

```
2.595989917947957e8914
```

9.85 `\xintFrac`

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

9.86 `\xintSignedFrac`

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

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

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

9.87 `\xintFwOver`

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

9.88 `\xintSignedFwOver`

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

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


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

9.89 `\xintLen`

Frac
f ★ The original `\xintLen` macro is extended to accept a fraction on input: the length of `A/B[n]` is the length of `A` plus the length of `B` plus the absolute value of `n` and minus one (an integer input as `N` is internally represented in a form equivalent to `N/1[0]` so the minus one means that the extended `\xintLen` behaves the same as the original for integers).

```
\xintLen{201710/298219}=\xintLen{201710}+\xintLen{298219}-1\newline
\xintLen{1234/1}=\xintLen{1234}=\xintLen{1234[0]}=\xintiLen{1234}\newline
\xintLen{-1e3/5.425} (\xintRaw {-1e3/5.425})\par
```

11=6+6-1

4=4=4=4

10 (-1/5425[6])

The length is computed on the `A/B[n]` which would have been returned by `\xintRaw`, as illustrated by the last example above.

`\xintLen` is only for use with such (scientific) numbers or fractions. See also `\xintNthElt` from *xinttools*. See also `\xintLength` (which however does not expand its argument) from *xintkernel* for counting more general tokens (or rather braced items).

10 Macros of the **xintexpr** package

.1	The \xintexpr expressions	122			
.2	\numexpr or \dimexpr expressions, count and dimension registers and variables	125	.14	\xintifsgnexpr, \xintifsgnfloatexpr, \xintifsgniexpr	132
.3	Catcodes and spaces	125	.15	The \xintNewExpr macro	132
.4	Expandability, \xintepro	126	.16	\xintNewFloatExpr	137
.5	Memory considerations	127	.17	\xintNewIExpr	138
.6	\xintiexpr, \xinttheiexpr	127	.18	\xintNewIIExpr	138
.7	\xintiexpr, \xinttheiexpr	128	.19	\xintNewBoolExpr	138
.8	\xintboolexpr, \xinttheboolexpr	129	.20	The \xintdefvar, \xintdefiivar, \xintdeffloatvar macros	138
.9	\xintfloatexpr, \xintthefloatexpr	129	.21	The \xintdeffunc, \xintdefiifunc, \xintdeffloatfunc macros	138
.10	\xinteval, \xintieval, \xintiieval, \xintfloateval	130	.22	The \xintNewFunction macro	138
.11	Using an expression parser within another one	130	.23	Technicalities	138
.12	The \xintthecoords macro	131	.24	Acknowledgements (2013/05/25)	139
.13	\xintifboolexpr, \xintifboolfloatexpr,				

The **xintexpr** package was first released with version 1.07 (2013/05/25) of the **xint** bundle. It was substantially enhanced with release 1.1 from 2014/10/28.

The package loads automatically **xintfrac** and **xinttools**.

New with 1.3d

- 1.3d adds `\xinteval`, `\xintieval`, `\xintiieval`, `\xintfloateval`.
- for the `gcd()` and `lcm()` functions, it is NOT necessary anymore to load package **xintgcd**. And they now work not only with integers:

```
\xinttheiexpr lcm (2^5*7*13^10*17^5,2^3*13^15*19^3,7^3*13*23^2)\relax\newline
```

2894379441338000036761046087608864

but also with fractions:

```
\xinttheexpr lcm(7/300, 11/150, 13/60), gcd(7/300, 11/150, 13/60)\relax\par
```

1001/30, 1/300

New with 1.3d

- for allowing hexadecimal (uppercase letters only) input, it is necessary to load package **xintbinhex**.

```
\xinttheexpr "A*B*C*D*D*F, "FF.FF, reduce("FF.FFF + 16^-3)\relax
```

3346200, 25599609375[-8], 256

Please refer to [section 2](#) for a more detailed description of some syntax elements.

10.1 The \xintexpr expressions

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

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

- x ★ `\thexintexpr<expandable_expression>\relax`, or
- x ★ `\xinttheexpr<expandable_expression>\relax`, or
- x ★ `\xintthe\xintexpr<expandable_expression>\relax`.

The computations are done *exactly*, and with no simplification of the result. See `\xintfloatexpr` for a similar parser which rounds each operation inside the expression to `\xinttheDigits` digits of precision.

As an alternative and equivalent syntax to

```
\xintexpr round(<expression>, D)\relax
```

there is⁶⁴

```
\xintiexpr [D] <expression> \relax
```

The parameter `D` must be zero or positive.⁶⁵ Perhaps some future version will give a meaning to using a negative `D`.⁶⁶

- the expression may contain arbitrarily many levels of nested parenthesized sub-expressions,
- the expression may contain explicitly or from a macro expansion a sub-expression `\xintexpr. .\relax`, which itself may contain a sub-expressions etc. . .
- to let sub-contents evaluate as a sub-unit it should thus be either
 1. parenthesized,
 2. or a sub-expression `\xintexpr... \relax`.
- to use an expression as argument to the other package macros, or more generally to macros which expand their arguments, one must use the `\xinttheexpr... \relax` or `\xintthe\xintexpr... \relax` forms.
- similarly, printing the result itself must be done with these forms.
- one should not use `\xinttheexpr... \relax` as a sub-constituent of an `\xintexpr... \relax` but only the `\xintexpr... \relax` form which is more efficient in this context.
- each xintexpr session, whether prefixed or not with `\xintthe`, is completely expandable and obtains its result in two expansion steps.

See [section 2](#) for the primary information on built-in operators and functions. This section now adds some complementary information.

- An expression is built the standard way with opening and closing parentheses, infix operators, and (big) numbers, with possibly a fractional part, and/or scientific notation (except for `\xintiexpr` which only admits big integers). All variants work with comma separated expressions. On output each comma will be followed by a space. A decimal number must have digits either before or after the decimal mark.
- As everything gets expanded, the characters `., +, -, *, /, ^, !, &, |, ?, :, <, >, =, (,), ",]`, `[, @` and the comma `,` should not (if used in the expression) be active. For example, the French language in *Babel* system, for `pdfLATEX`, activates `!, ?, ;` and `..`. Turn off the activity before expressions using such characters.

Alternatively the macro `\xintexprSafeCatcodes` resets all characters potentially needed by `\xintexpr` to their standard catcodes and `\xintexprRestoreCatcodes` restores the former status.

- Count registers and `\numexpr`-essions are accepted (LaTeX's counters can be inserted using `\value`) natively without `\the` or `\number` as prefix. Also dimen registers and control sequences, skip registers and control sequences (LaTeX's lengths), `\dimexpr`-essions, `\glueexpr`-essions are automatically unpacked using `\number`, discarding the stretch and shrink components and giving the dimension value in `sp` units (1/65536th of a TeX point). Furthermore, tacit multiplication

⁶⁴ For truncation rather than rounding, one uses `\xintexpr trunc(<expression>, D)\relax`. ⁶⁵ `D=0` corresponds to using `round(<expression>)` not `round(<expression>,0)` which would leave a trailing dot. Same for `trunc`. There is also function `float` for floating point rounding to `\xinttheDigits` or the given number of significant digits as second argument. ⁶⁶ Thanks to KT for this suggestion. Sorry for the delay in implementing it... matter of formatting the output and corresponding choice of user interface are still in need of some additional thinking.



is implied, when the (count or dimen or glue) register or variable, or the (`\numexpr` or `\dimexpr` or `\glueexpr`) expression is immediately prefixed by a (decimal) number. See [subsection 2.4](#) for the complete rules of tacit multiplication.

- With a macro `\x` defined like this:

```
\def\x {\xintexpr \a + \b \relax} or \edef\x {\xintexpr \a+\b\relax}
```

one may then do `\xintthe\x`, either for printing the result on the page or to use it in some other macros expanding their arguments. The `\edef` does the computation immediately but keeps it in an internal private format. Naturally, the `\edef` is only possible if `\a` and `\b` are already defined. With both approaches the `\x` can be inserted in other expressions, as for example (assuming naturally as we use an `\edef` that in the 'yet-to-be computed' case the `\a` and `\b` now have some suitable meaning):

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

- There is also `\xintboolexpr ... \relax` and `\xinttheboolexpr ... \relax`. Same as `\xintexpr` with the final result converted to 1 if it is not zero.
- See also `\xintifboolexpr` ([subsection 10.13](#)) and the `bool()` and `toql()` functions in [section 10](#). Here is an example:

```
\xintNewBoolExpr \AssertionA[3]{ #1 && (#2||#3) }
\xintNewBoolExpr \AssertionB[3]{ #1 || (#2&&#3) }
\xintNewBoolExpr \AssertionC[3]{ xor(#1,#2,#3) }
{\centering\normalcolor\xintFor #1 in {0,1} \do {%
\xintFor #2 in {0,1} \do {%
\xintFor #3 in {0,1} \do {%
#1 AND (#2 OR #3) is \textcolor[named]{OrangeRed}{\AssertionA {#1}{#2}{#3}}\hfil
#1 OR (#2 AND #3) is \textcolor[named]{OrangeRed}{\AssertionB {#1}{#2}{#3}}\hfil
#1 XOR #2 XOR #3 is \textcolor[named]{OrangeRed}{\AssertionC {#1}{#2}{#3}}\}}}
0 AND (0 OR 0) is 0          0 OR (0 AND 0) is 0          0 XOR 0 XOR 0 is 0
0 AND (0 OR 1) is 0          0 OR (0 AND 1) is 0          0 XOR 0 XOR 1 is 1
0 AND (1 OR 0) is 0          0 OR (1 AND 0) is 0          0 XOR 1 XOR 0 is 1
0 AND (1 OR 1) is 0          0 OR (1 AND 1) is 1          0 XOR 1 XOR 1 is 0
1 AND (0 OR 0) is 0          1 OR (0 AND 0) is 1          1 XOR 0 XOR 0 is 1
1 AND (0 OR 1) is 1          1 OR (0 AND 1) is 1          1 XOR 0 XOR 1 is 0
1 AND (1 OR 0) is 1          1 OR (1 AND 0) is 1          1 XOR 1 XOR 0 is 0
1 AND (1 OR 1) is 1          1 OR (1 AND 1) is 1          1 XOR 1 XOR 1 is 1
```

This example used for efficiency `\xintNewBoolExpr`. See also the [subsection 10.15](#).

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

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

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

```
\xinttheexpr sqrt(2,60)\relax\newline
Here the [60] is to avoid truncation to |\xinttheDigits| of precision on output.\newline
\printnumber{\xintthefloatexpr [60] sqrt(2,60)\relax}
```

```
141421356237309504880168872420969807856967187537694807317668[-59]
```

Here the [60] is to avoid truncation to `\xinttheDigits` of precision on output.

```
1.41421356237309504880168872420969807856967187537694807317668
```

Floats are quickly indispensable when using the power function, as exact results will easily have hundreds, if not thousands, of digits.

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

```
9.99002093014384507944032764330033590980429139054e30102
```

Only integer and (in `\xintfloatexpr... \relax`) half-integer exponents are allowed.

- if one uses *macros* within `\xintexpr...\relax` one should obviously take into account that the parser will *not* see the macro arguments, hence once cannot use the syntax there, except if the arguments are themselves wrapped as `\xinttheexpr...\relax` and assuming the macro *f-expands* these arguments.

10.2 `\numexpr` or `\dimexpr` expressions, count and dimension registers and variables

Count registers, count control sequences, dimen registers, dimen control sequences (like `\parindent`), skips and skip control sequences, `\numexpr`, `\dimexpr`, `\glueexpr`, `\fontdimen` can be inserted directly, they will be unpacked using `\number` which gives the internal value in terms of scaled points for the dimensional variables: `1pt = 65536sp` (stretch and shrink components are thus discarded).

Tacit multiplication (see [subsection 2.4](#)) is implied, when a number or decimal number prefixes such a register or control sequence. \TeX lengths are skip control sequences and \TeX counters should be inserted using `\value`.

Release 1.2 of the `\xintexpr` parser also recognizes and prefixes with `\number` the `\ht`, `\dp`, and `\wd` \TeX primitives as well as the `\fontcharht`, `\fontcharwd`, `\fontchardp` and `\fontcharic` ϵ - \TeX primitives.

In the case of numbered registers like `\count255` or `\dimen0` (or `\ht0`), the resulting digits will be re-parsed, so for example `\count255 0` is like `100` if `\the\count255` would give `10`. The same happens with inputs such as `\fontdimen6\font`. And `\numexpr 35+52\relax` will be exactly as if `87` as been encountered by the parser, thus more digits may follow: `\numexpr 35+52\relax 000` is like `87000`. If a new `\numexpr` follows, it is treated as what would happen when `\xintexpr` scans a number and finds a non-digit: it does a tacit multiplication.

```
\xinttheexpr \numexpr 351+877\relax\numexpr 1000-125\relax\relax{} is the same
as \xinttheexpr 1228*875\relax.
```

1074500 is the same as 1074500.

Control sequences however (such as `\parindent`) are picked up as a whole by `\xintexpr`, and the numbers they define cannot be extended extra digits, a syntax error is raised if the parser finds digits rather than a legal operation after such a control sequence.

A token list variable must be prefixed by `\the`, it will not be unpacked automatically (the parser will actually try `\number`, and thus fail). Do not use `\the` but only `\number` with a dimen or skip, as the `\xintexpr` parser doesn't understand `pt` and its presence is a syntax error. To use a dimension expressed in terms of points or other \TeX recognized units, incorporate it in `\dimexpr...\relax`.

Regarding how dimensional expressions are converted by \TeX into scaled points see also [subsection 3.7](#).

10.3 Catcodes and spaces

Active characters may (and will) break the functioning of `\xintexpr`. Inside an expression one may prefix, for example a `:` with `\string`. Or, for a more radical way, there is `\xintexprSafeCatcodes`. This is a non-expandable step as it changes catcodes.

10.3.1 `\xintexprSafeCatcodes`

This macro sets the catcodes of many characters to safe values. This is used internally by `\xintNewExpr` (restoring the catcodes on exit), hence it does not have to be protected against active characters when used at top-level.

Also `\xintdefvar`, `\xintdeffunc`, ..., use it before fetching their semi-colon delimited arguments, so they can be used (also in the document body) for example with Babel+French which makes the semi-colon active in the (\mathTeX) document body.

Changed at 1.3c!

As `\xintNewExpr` and `\xintdeffunc` and variants use internally some `\scantokens`, they will (reasonably) succeed in sanitizing catcodes in the expressions, even if all is from the replacement text of some macro whose definition was done under some special catcode regime.

But `\xintdeffunc`, `\xintdefvar` and variants need the (catcode other) semi-colon as delimiter. Thus make sure the semi-colon has its normal catcode when using `\xintdeffunc` inside some macro definition.

The macros `\xintdeffunc` and variants ignore completely the colon in `:=` (which furthermore is optional) so it can have any (reasonable) frozen catcode.

The macros `\xintdefvar` and variants are also compatible with any reasonable frozen catcode of the colon `:` in `:=`, and the colon presence is only optional.

New with
1.3c



It is important to ALWAYS shortly let `\xintexprSafeCatcodes` be followed by `\xintexprRestoreCatcodes`. If one uses twice `\xintexprSafeCatcodes` then the next `\xintexprRestoreCatcodes` will restore the ancien catcode regime at time of the first one.

10.3.2 `\xintexprRestoreCatcodes`

Restores the catcodes to the earlier state. More precisely, `\xintexprSafeCatcodes` sets a toggle (with local scope). If the toggle is set already it does not restore the current catcodes. The next `\xintexprRestoreCatcodes` unsets the toggle. So, in case of nesting, the catcodes are restored to what they were when the *first* un-paired `\xintexprSafeCatcodes` got executed.

Changed
at 1.3c!

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

`\xintexpr` and `\xinttheexpr` are for the most part agnostic regarding catcodes: (unbraced) digits, binary operators, minus and plus signs as prefixes, dot as decimal mark, parentheses, may be indifferently of catcode letter or other or subscript or superscript, ..., it doesn't matter.⁶⁷

The characters `+`, `-`, `*`, `/`, `^`, `!`, `&`, `|`, `?`, `:`, `<`, `>`, `=`, `(`, `)`, `"`, `[`, `]`, `;`, the dot and the comma should not be active if in the expression, as everything is expanded along the way. If one of them is active, it should be prefixed with `\string`.

The exclamation mark `!` should have its standard catcode: with catcode letter it is used internally and hence will confuse the parsers if it comes from the expression.

Digits, slash, square brackets, minus sign, in the output from an `\xinttheexpr` are all of catcode 12. For `\xintthefloatexpr` the ``e'` in the output has its standard catcode ``letter'`.

A macro with arguments will expand and grab its arguments before the parser may get a chance to see them, so the situation with catcodes and spaces is not the same within such macro arguments.

10.4 Expandability, `\xintexpr`

As is the case with all other package macros `\xintexpr f-expands` (in two steps) to its final (non-printable) result; and `\xinttheexpr f-expands` (in two steps) to the chain of digits (and possibly minus sign `-`, decimal mark `.`, fraction slash `/`, scientific `e`, square brackets `[,]`) representing the result.

Starting with 1.09j, an `\xintexpr...\relax` can be inserted without `\xintthe` prefix inside an `\edef`, or a `\write`. It expands to a private more compact representation (five tokens) than `\xinttheexpr` or `\xintthe\xintexpr`.

The material between `\xintexpr` and `\relax` should contain only expandable material.

The once expanded `\xintexpr` is `\romannumeral0\xintexpr`.

ATTENTION! Prior to 1.3d the `\xintexpr` macro was named `\xinteval`.

Changed
at 1.3d!

⁶⁷ Furthermore, although `\xintexpr` uses `\string`, it is escape-char agnostic. It should work with any `\escapechar` setting including `-1`.

But `\xinteval` is now something else. And there is similarly `\xintiexpr` (formerly `\xintieval`), `\xintiexpr` (formerly `\xintiieval`), and `\xintfloatexpr` (formerly `\xintfloateval`). For an example see [subsection 2.10](#).

An expression can only be legally finished by a `\relax` token, which will be absorbed.

It is quite possible to nest expressions among themselves; for example, if one needs inside an `\xintiexpr... \relax` to do some computations with fractions, rounding the final result to an integer, one just has to insert `\xintiexpr... \relax`. The functioning of the infix operators will not be in the least affected from the fact that the surrounding ``environment'' is the `\xintiexpr` one.

10.5 Memory considerations

The parser creates an undefined control sequence for each intermediate computation evaluation: addition, subtraction, etc. . . Thus, a moderately sized expression might create 10, or 20 such control sequences. On my \TeX installation, the memory available for such things is of circa 200, 000 multi-letter control words. So this means that a document containing hundreds, perhaps even thousands of expressions will compile with no problem.

Besides the hash table, also \TeX main memory is impacted. Thus, if xintexpr is used for computing plots⁶⁸, this may cause a problem. In my testing and with current TL2015 memory settings, I ran into problems after doing about *ten thousand* evaluations (for example $(\#1+\#2)*\#3-\#1*\#3-\#2*\#3$) each with number having *hundreds* of digits. Typical error message can be:

```
./testaleatoires.tex:243: TeX capacity exceeded, sorry [pool size=6134970].
<argument> ...19140037877484848545931233090884903
There is a (partial) solution.69
```

A document can possibly do tens of thousands of evaluations only if some identical formulae are being used repeatedly, with varying arguments (from previous computations possibly) or coming from data being fetched from a file. Most certainly, there will be a few dozens formulae at most, but they will be used again and again with varying inputs.

With the `\xintNewExpr` macro, it is possible to convert once and for all an expression containing parameters into an expandable macro with parameters. Only this initial definition of this macro actually activates the `\xintexpr` parser and will (very moderately) impact the hash-table: once this unique parsing is done, a macro with parameters is produced which is built-up recursively from the `\xintAdd`, `\xintMul`, etc... macros, exactly as it would be necessary to do without the facilities of the xintexpr package.

Notice that since 1.2c the `\xintdeffunc` construct allows an alternative to `\xintNewExpr` whose syntax uses arbitrary letters rather than macro parameters #1, #2, . . . , #9. The declared function must still be used inside an expression, but its use will need only as many `\csname`'s as were needed for the function arguments plus one more for encapsulating the function result.

10.6 `\xintiexpr`, `\xinttheiexpr`

- x ★ Equivalent to doing `\xintexpr round(...)\relax` (more precisely, `round` is applied to each one of the evaluated values, if the expression was comma separated). Thus, only the *final result value* is rounded to an integer. Half integers are rounded towards $+\infty$ for positive numbers and towards $-\infty$ for negative ones.

An optional parameter `d>0` within brackets, immediately after `\xintiexpr` is allowed: it instructs the expression to do its final rounding to the nearest value with that many digits after the decimal mark, *i.e.*, `\xintiexpr [d] <expression>\relax` is equivalent (in case of a single expression) to `\xintexpr round(<expression>, d)\relax`.

`\xintiexpr [0] ...` is the same as `\xintiexpr ...`⁷⁰

⁶⁸ this is not very probable as so far `xint` does not include a mathematical library with floating point calculations, but provides only the basic operations of algebra. ⁶⁹ which convinced me that I could stick with the parser implementation despite its potential impact on the hash-table and other parts of \TeX 's memory. ⁷⁰ Incidentally using `round(...,0)` in place of `round(...)` in `\xintexpr` would leave a trailing dot in the produced value.

If truncation rather than rounding is needed use (in case of a single expression, naturally) `\xintexpr trunc(...)\relax` for truncation to an integer or `\xintexpr trunc(...,d)\relax` for truncation to a decimal number with $d > 0$ digits after the decimal mark.

Perhaps in the future some meaning will be given to using negative value for the optional parameter d .⁷¹

`\thexintexpr` is synonym to `\xinttheiexpr`.

10.7 `\xintiexpr`, `\xinttheiexpr`

- x ★ This variant does not know fractions. It deals almost only with long integers. Comma separated lists of expressions are allowed.

It maps `/` to the *rounded* quotient. The operator `//` is, like in `\xintexpr...\relax`, mapped to *truncated* division. The Euclidean quotient (which for positive operands is like the truncated quotient) was, prior to release 1.1, associated to `/`. The function `quo(a,b)` can still be employed.

The `\xintiexpr`-essions use the ``ii'` macros for addition, subtraction, multiplication, power, square, sums, products, Euclidean quotient and remainder.

The `round`, `trunc`, `floor`, `ceil` functions are still available, and are about the only places where fractions can be used, but `/` within, if not somehow hidden will be executed as integer rounded division. To avoid this one can wrap the input in `qfrac`: this means however that none of the normal expression parsing will be executed on the argument.

To understand the illustrative examples, recall that `round` and `trunc` have a second (non negative) optional argument. In a normal `\xintexpr`-essions, `round` and `trunc` are mapped to `\xintRound` and `\xintTrunc`, in `\xintiexpr`-essions, they are mapped to `\xintiRound` and `\xintiTrunc`.

```
\xinttheiexpr 5/3, round(5/3,3), trunc(5/3,3), trunc(\xintDiv {5}{3},3),
trunc(\xintRaw {5/3},3)\relax{} are problematic, but
%
\xinttheiexpr 5/3, round(qfrac(5/3),3), trunc(qfrac(5/3),3), floor(qfrac(5/3)),
ceil(qfrac(5/3))\relax{} work!
```

2, 2000, 2000, 2000, 2000 are problematic, but 2, 1667, 1666, 1, 2 work!

On the other hand decimal numbers and scientific numbers can be used directly as arguments to the `num`, `round`, or any function producing an integer.

Scientific numbers will be represented with as many zeroes as necessary, thus one does not want to insert `num(1e100000)` for example in an `\xintiexpression` !

```
\xinttheiexpr num(13.4567e3)+num(10000123e-3)\relax % should (num truncates) compute 13456+10000
23456
```

The `reduce` function is not available and will raise an error. The `frac` function also. The `sqr` function is mapped to `\xintiSqrt` which gives a truncated square root. The `sqrtr` function is mapped to `\xintiSqrtR` which gives a rounded square root.

```
\xinttheiexpr \xintFloatSqrt [20]{2}, \xintFloatSqrt [20]{3}\relax % no operations
```

`\noindent` The next example requires the `|round|`, and one could not put the `|+|` inside it:

```
\xinttheiexpr round(\xintFloatSqrt [20]{2},19)+round(\xintFloatSqrt [20]{3},19)\relax
```

(the second argument of `|round|` and `|trunc|` tells how many digits from after the

⁷¹ Thanks to KT for this suggestion.

decimal mark one should keep.)

```
14142135623730950488[-19], 17320508075688772935[-19]
```

The next example requires the `round`, and one could not put the `+` inside it:

```
31462643699419723423
```

(the second argument of `round` and `trunc` tells how many digits from after the decimal mark one should keep.)

The whole point of `\xintiexpr` is to gain some speed in *integer-only* algorithms, and the above explanations related to how to nevertheless use fractions therein are a bit peripheral. We observed (2013/12/18) of the order of 30% speed gain when dealing with numbers with circa one hundred digits (1.2: this info may be obsolete).

`\thexintiexpr` is synonym to `\xinttheiexpr`.

10.8 `\xintboolexpr`, `\xinttheboolexpr`

- x ★ Equivalent to doing `\xintexpr ...\relax` and returning 1 if the result does not vanish, and 0 if the result is zero. As `\xintexpr`, this can be used on comma separated lists of expressions, and will return a comma separated list of 0's and 1's.

`\thexintboolexpr` is synonym to `\xinttheboolexpr`.

There is slight quirk in case it is used as a sub-expression: the boolean expression needs at least one logic operation else the value is not standardized to 1 or 0, for example we get from

```
\xinttheexpr \xintboolexpr 1.23\relax\relax\newline
```

```
123[-2]
```

which is to be compared with

```
\xinttheboolexpr 1.23\relax
```

```
1
```

A related issue existed with `\xinttheexpr \xintiexpr 1.23\relax\relax`, which was fixed with 1.2 release, and I decided back then not to add the needed overhead also to the `\xintboolexpr` context, as one only needs to use `?(1.23)` for example or involve the 1.23 in any logic operation like 1.23 'and' 3.45, or involve the `\xintboolexpr ...\relax` itself with any logical operation, contrarily to the sub-`\xintiexpr` case where `\xinttheexpr 1+\xintiexpr 1.23\relax\relax` did behave contrarily to expectations until 1.1.

10.9 `\xintfloatexpr`, `\xintthefloatexpr`

- x ★ `\xintfloatexpr...relax` is exactly like `\xintexpr...relax` but with the four binary operations and the power function are mapped to `\xintFloatAdd`, `\xintFloatSub`, `\xintFloatMul`, `\xintFloatDiv` and `\xintFloatPower`, respectively.⁷²

The target precision for the computation is from the current setting of `\xintDigits`. Comma separated lists of expressions are allowed.

An optional parameter within brackets is allowed:

- if positive it instructs the macro to round the result to that many digits of precision. It thus makes sense to employ it only if this parameter is less than the `\xinttheDigits` precision.
- if negative it means to trim off that many digits (of course, rounding the value). Don't use it to trim all digits (or more than all)!

New with
1.3e

Since 1.2f all float operations first round their arguments; a parsed number is not rounded prior to its use as operand to such a float operation.

`\thexintfloatexpr` is synonym to `\xintthefloatexpr`.

```
\xintDigits:=36;
```

```
\xintthefloatexpr (1/13+1/121)*(1/179-1/173)/(1/19-1/18)\relax
```

```
0.00564487459334466559166166079096852897
```

⁷² Since 1.2f the `^` handles half-integer exponents, contrarily to `\xintFloatPower`.

```
\xintthefloatexpr\xintexpr (1/13+1/121)*(1/179-1/173)/(1/19-1/18)\relax\relax
0.00564487459334466559166166079096852912
```

The latter is the rounding of the exact result. The former one has its last three digits wrong due to the cumulative effect of rounding errors in the intermediate computations, as compared to exact evaluations.

I recall here from [subsection 3.2](#) that with release [1.2f](#) the float macros for addition, subtraction, multiplication and division round their arguments first to P significant places with P the asked-for precision of the output; and similarly the power macros and the square root macro. This does not modify anything for computations with arguments having at most P significant places already.

10.10 `\xinteval`, `\xintieval`, `\xintiieval`, `\xintfloateval`

New with
1.3d



Prior to [1.3d](#), these macros existed but with a different meaning: they arose in the once-expanded `\xintexpr`, etc..., i.e. one had:

```
\def\xintexpr{\romannumeral0\xinteval}
```

The old macros were renamed into `\xintexpr`, etc..., in order to free their names for new meanings, more alike what one finds in other math packages.

x ★ `\xinteval` is an *f-expandable* macro which is basically defined like this:

```
\def\xinteval#1{\romannumeral-`0\xinttheexpr#1\relax}
```

thus expands in two steps (its exact definition differs from the one given above in order to achieve a slight optimization).

```
\xinteval{add(x^2, x = 100..110), add(x^3, x = 100..110)}
```

121385, 12768525

x ★ `\xintieval` is similarly related to `\xinttheiexpr`. Its optional argument must be located inside the braces:

```
\xintieval{[7] 355/113}
```

3.1415929

x ★ `\xintiieval` is similarly related to `\xinttheiexpr`.

```
\xintiieval{add(x^2, x = 100..110), add(x^3, x = 100..110)}
```

121385, 12768525

x ★ `\xintfloateval` is similarly related to `\xintthefloatexpr`. Its optional argument must be located inside the braces:

```
\xintfloateval{[7] 355/113}
```

3.141593

New with
1.3e

When negative it tells how many digits to remove from the prevailing precision (`\xinttheDigits`):

```
\xintfloateval{[-2] 355/113} has \xinttheDigits\ minus 2 digits.
```

3.1415929203540 has 16 minus 2 digits.

These macros are useful when one uses some extra wrapper doing some parsing of its input, like the `\num` macro of [siunitx](#), which would choke on some of the syntax elements allowed inside `\xintexpr..` `\relax` (for example brackets). As shown in the above examples, these macros, like the underlying parsers accept arbitrarily many comma separated expressions.

10.11 Using an expression parser within another one

This was already illustrated before. In the following:

```
\xintthefloatexpr \xintexpr add(1/i, i=1234..1243)\relax ^100\relax
```

5.136088460396579e-210, the inner sum is computed exactly. Then it will be rounded to `\xinttheDigits` significant digits, and then its power will be evaluated as a float operation. One should avoid the "`\xintthe`" parsers in inner positions as this induces digit by digit parsing of the inner computation result by the outer parser. Here is the same computation done with floats all the way:

```
\xintthefloatexpr add(1/i, i=1234..1243)^100\relax
```

5.136088460396643e-210

Not surprisingly this differs from the previous one which was exact until raising to the 100th power.

The fact that the inner expression occurs inside a bigger one has nil influence on its behaviour. There is the limitation though that the outputs from `\xintexpr` and `\xintfloatexpr` can not be used directly in `\xinttheiexpr` integer-only parser. But one can do:

```
\xinttheiexpr round(\xintfloatexpr 3.14^10\relax)\relax % or trunc
```

93174

10.12 The `\xintthecoords` macro

It converts a comma separated list into the format for list of coordinates as expected by the `TikZ coordinates` syntax.⁷³

```
\begin{figure}[htbp]
\centering\begin{tikzpicture}[scale=10]\xintDigits:=8;
\clip (-1.1,-.25) rectangle (.3,.25);
\draw [blue] (-1.1,0)--(1,0);
\draw [blue] (0,-1)--(0,+1);
\draw [red] plot[smooth] coordinates {%
\xintthecoords % (converts what is next into (x1, y1) (x2, y2)... format)
\xintfloatexpr seq((x^2-1,mul(x-t,t=-1+[0..4]/2)),x=-1.2..[0.1]..+1.2) \relax };
\end{tikzpicture}
\caption{Coordinates with \cs{xintthecoords}.}
\end{figure}
```

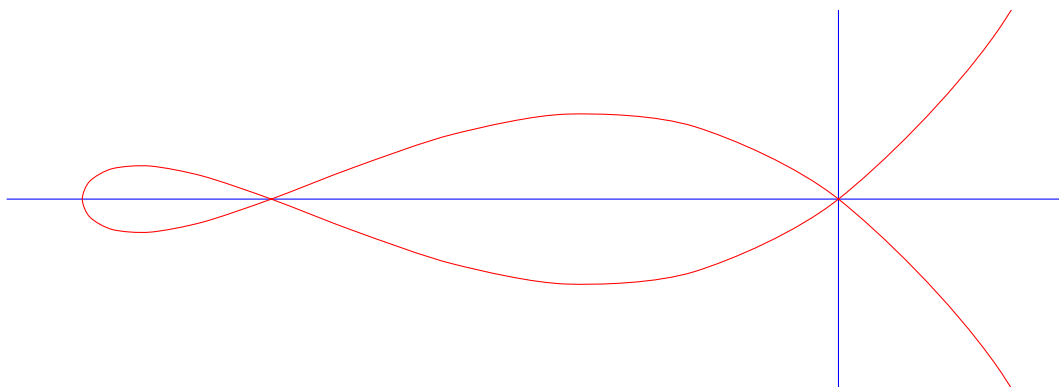


Figure 1: Coordinates with `\xintthecoords`.

As exemplified above, `\xintthecoords` is to be used followed immediately by either `\xintfloatexpr` or `\xintiexpr` or `\xintiiexpr`. See <https://tex.stackexchange.com/a/447290> for another example.

As `TikZ` will not understand the `A/B[N]` format which is used on output by `\xintexpr`, `\xintthecoords\xintexpr` has no use inside a `TikZ` picture but may have other usages; the reason for the spaces in output is to allow if necessary to print on the page for examination and give \TeX a change to establish line-breaks.

```
\edef\x{\xintthecoords \xintexpr rrseq(1/2,1/3; @1+@2, x=1..20)\relax }
\meaning\x +++
```

macro:->(1/2, 1/3) (5/6, 7/6) (12/6, 19/6) (31/6, 50/6) (81/6, 131/6) (212/6, 343/6) (555/6, 898/6) (1453/6, 2351/6) (3804/6, 6155/6) (9959/6, 16114/6) (26073/6, 42187/6)+++

⁷³ The implementation had to work around the problem that `TikZ` seemingly allows only a maximal number of about one hundred expansion steps for the list to be entirely produced.

10.13 `\xintifboolexpr`, `\xintifboolfloatexpr`, `\xintifboolliexpr`

xnn ★ `\xintifboolexpr{<expr>}{<YES>}{<NO>}` does `\xinttheexpr<expr>\relax` and then executes the `<YES>` or the `<NO>` branch depending on whether the outcome was non-zero or zero. Thus one can read *if bool expr* as meaning *if not zero*:

if `<expr>`-ession does not vanish do `<YES>` else do `<NO>`

The expression is not limited to using only comparison operators and Boolean logic (`<`, `>`, `==`, `!=`, `&&`, `||`, `all()`, `any()`, `xor()`, `bool()`, `togl()`, ...), it can be the most general computation.

xnn ★ `\xintifboolfloatexpr{<expr>}{<YES>}{<NO>}` does `\xintthefloatexpr<expr>\relax` and then executes the `<YES>` or the `<NO>` branch depending on whether the outcome was non zero or zero.

xnn ★ `\xintifboolliexpr{<expr>}{<YES>}{<NO>}` does `\xinttheiexpr<expr>\relax` and then executes the `<YES>` or the `<NO>` branch depending on whether the outcome was non zero or zero.

The expression argument must be a single one, comma separated sub-expressions will cause low-level errors.

10.14 `\xintifsgnexpr`, `\xintifsgnfloatexpr`, `\xintifsgniexpr`

xnnn ★ `\xintifsgnexpr{<expr>}{<<0>}{<=0>}{<>0>}` evaluates the `\xintexpression` and chooses the branch corresponding to its sign.

xnnn ★ `\xintifsgnfloatexpr{<expr>}{<<0>}{<=0>}{<>0>}` evaluates the `\xintfloatexpression` and chooses the branch corresponding to its sign.

xnnn ★ `\xintifsgniexpr{<expr>}{<<0>}{<=0>}{<>0>}` evaluates the `\xintiexpression` and chooses the branch corresponding to its sign.

New with 1.3d The expression argument must be a single one, comma separated sub-expressions will cause low-level errors.

10.15 The `\xintNewExpr` macro

The macro is used as:

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

- `<stuff>` will be inserted inside `\xinttheexpr . . . \relax`,
- `n` is an integer between zero and nine, inclusive, which is the number of parameters of `\myformula`,
- the placeholders `#1`, `#2`, ..., `#n` are used inside `<stuff>` in their usual rôle,^{74 75}
- the `[n]` is mandatory, even for `n=0`.⁷⁶
- the macro `\myformula` is defined without checking if it already exists, \TeX users might prefer to do first `\newcommand*{<myformula> {}}` to get a reasonable error message in case `\myformula` already exists,
- the protection against active characters is done automatically (as long as the whole thing has not already been fetched as a macro argument and the catcodes correspondingly already frozen).

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

```
\xintNewFloatExpr \FA [2]{(#1+#2)^10}
\xintNewFloatExpr \FB [2]{sqrt(#1*#2)}
\begin{enumerate}[nosep]
\item \FA {5}{5}
```

⁷⁴ if `\xintNewExpr` is used inside a macro, the `#`'s must be doubled as usual. ⁷⁵ the `#`'s will in practice have their usual catcode, but category code other `#`'s are accepted too. ⁷⁶ there is some use for `\xintNewExpr[0]` compared to an `\edef` as `\xintNewExpr` has some built-in catcode protection.

```
\item \FB {30}{10}
\item \FA {\FB {30}{10}}{\FB {40}{20}}
\end{enumerate}
```

1. 1.0000000000000000e10
2. 17.32050807568877
3. 3.891379490446502e16

The use of `\xintNewExpr` circumvents the impact of the `\xintexpr` parsers on TeX's memory: it is useful if one has a formula which has to be re-evaluated thousands of times with distinct inputs each with dozens, or hundreds of digits.

A ```formula''` created by `\xintNewExpr` is thus a macro whose parameters are given to a possibly very complicated combination of the various macros of `xint` and `xintfrac`. Consequently, one can not use at all any infix notation in the inputs, but only the formats which are recognized by the `xintfrac` macros.

This is thus quite different from a macro with parameters which one would have defined via a simple `\def` or `\newcommand` as for example:

```
\newcommand\myformula [1]{\xinttheexpr (#1)^3\relax}
```

Such a macro `\myformula`, if it was used tens of thousands of times with various big inputs would end up populating large parts of TeX's memory. It would thus be better for such use cases to go for:

```
\xintNewExpr\myformula [1]{#1^3\relax}
```

Here naturally the situation is over-simplified and it would be even simpler to go directly for the use of the macro `\xintPow` or `\xintPower`.

`\xintNewExpr` tries to do as many evaluations as are possible at the time the macro parameters are still parameters. Let's see a few examples. For this I will use `\meaning` which reveals the contents of a macro.

1. the examples use a mysterious `\fixmeaning` macro, which is there to get in the display `\romannumeral`^^@` rather than the frankly cabalistic `\romannumeral`` which made the admiration of the readers of the documentation dated 2015/10/19 (the second ``` stood for an ascii code zero token as per T1 encoded `newtxtt` font). Thus the true meaning is ```fixed''` to display something different which is how the macro could be defined in a standard `tex` source file (modulo, as one can see in example, the use of characters such as `:` as letters in control sequence names). Prior to 1.2a, the meaning would have started with a more mundane `\romannumeral-`0`, but I decided at the time of releasing 1.2a to imitate the serious guys and switch for the more hacky yet `\romannumeral`^^@` everywhere in the source code (not only in the macros produced by `\xintNewExpr`), or to be more precise for an equivalent as the caret has catcode letter in `xint`'s source code, and I had to use another character.
2. the meaning reveals the use of some private macros from the `xint` bundle, which should not be directly used. If the things look a bit complicated, it is because they have to cater for many possibilities.
3. the point of showing the meaning is also to see what has already been evaluated in the construction of the macros.

```
\xintNewIExpr\FA [1]{13*25*78*#1+2826*292}\fixmeaning\FA
```

```
macro:#1->\romannumeral`^^@\xintCSV::csv {\xintiiAdd {\xintiiMul {25350}{#1}}{825192}}
```

```
\xintNewIExpr\FA [2]{(3/5*9/7*13/11*#1-#2)*3^7}
\printnumber{\fixmeaning\FA}
```

```
macro:#1#2->\romannumeral`^^@\xintSPRaw::csv {\xintRound::csv {0}{\xintMul {\xintSub {\xintMu}
1 {351/385[0]}{#1}}{#2}}{2187/1[0]}}
```

```
% an example with optional parameter
```

```
\xintNewIExpr\FA [3]{[24] (#1+#2)/(#1-#2)^#3}
\printnumber{\fixmeaning\FA}
```

```
macro:#1#2#3->\romannumeral`^^@\xintSPRaw::csv {\xintRound::csv {24}{\xintDiv {\xintAdd {#1}{#2}}{\xintPow {\xintSub {#1}{#2}}{#3}}}
```

```
\xintNewFloatExpr\FA [2]{[12] 3.1415^3*#1-#2^5}
\printnumber{\fixmeaning\FA}
```

```
macro:#1#2->\romannumeral`^^@\xintPFloat::csv {12}{\XINTinFloatSub {\XINTinFloatMul {31003533}
39837500[-14]}{#1}}{\XINTinFloatPowerH {#2}{5}}
```

```
\xintNewExpr\DET[9]{ #1*#5*#9+#2*#6*#7+#3*#4*#8-#1*#6*#8-#2*#4*#9-#3*#5*#7 }
\printnumber{\fixmeaning\DET}
```

```
macro:#1#2#3#4#5#6#7#8#9->\romannumeral`^^@\xintSPRaw::csv {\xintSub {\xintSub {\xintSub {\xintAdd {\xintAdd {\xintMul {\xintMul {#1}{#5}}{#9}}{\xintMul {\xintMul {#2}{#6}}{#7}}}{\xintMul {\xintMul {#3}{#4}}{#8}}}{\xintMul {\xintMul {#1}{#6}}{#8}}}{\xintMul {\xintMul {#2}{#4}}{#9}}}{\xintMul {\xintMul {#3}{#5}}{#7}}
```

Notice that since 1.2c it is perhaps more natural to do:

```
% attention that «ad» would try to use non-existent variable "ad"
\xintdeffunc det2(a, b, c, d) := a*d - b*c ;
% This is impossible because we must use single letters :
% \xintdeffunc det3(x_11, x_12, x_13, x_21, x_22, x_23, x_31, x_32, x_33) :=
% x_11 * det2 (x_22, x_23, x_32, x_33) + x_21 * det2 (x_32, x_33, x_12, x_13)
% + x_31 * det2 (x_12, x_13, x_22, x_23);
\xintdeffunc det3 (a, b, c, u, v, w, x, y, z) := a*v*z + b*w*x + c*u*y - b*u*z - c*v*x - a*w*y ;
\xinttheexpr det3 (1,1,1,1,2,4,1,3,9), det3 (1,10,100,1,100,10000,1,1000,1000000),
90*900*990, reduce(det3 (1,1/2,1/3,1/2,1/3,1/4,1/3,1/4,1/5))\relax\newline
\xintdeffunc det3bis (a, b, c, u, v, w, x, y, z) :=
a*det2(v,w,y,z)-b*det2(u,w,x,z)+c*det2(u,v,x,y);
\pdfsetrandomseed 123456789 % xint.pdf should be predictable from xint.dtx !
\xinttheexpr subs(subs(subs(subs(subs(subs(subs(subs(
% we use one extra pair of parentheses to hide the commas from the subs
(a, b, c, u, v, w, x, y, z, det3 (a, b, c, u, v, w, x, y, z),
det3bis (a, b, c, u, v, w, x, y, z)),
z=\pdfuniformdeviate 1000), y=\pdfuniformdeviate 1000), x=\pdfuniformdeviate 1000),
w=\pdfuniformdeviate 1000), v=\pdfuniformdeviate 1000), u=\pdfuniformdeviate 1000),
c=\pdfuniformdeviate 1000), b=\pdfuniformdeviate 1000), a=\pdfuniformdeviate 1000)\relax
```

2, 80190000, 80190000, 1/2160

339, 694, 33, 664, 31, 921, 891, 763, 353, 188129929, 188129929

The last computation with its nine nested `subs` can be coded more economically (and efficiently), exploiting the fact that a single dummy variable can expand to a whole list:

```
\pdfsetrandomseed 123456789 % xint.pdf should be predictable from xint.dtx !
\xinttheexpr subs((L, det3(L), det3bis(L)), % parentheses used to hide the inner commas
L=\pdfuniformdeviate 1000, \pdfuniformdeviate 1000, \pdfuniformdeviate 1000,
\pdfuniformdeviate 1000, \pdfuniformdeviate 1000, \pdfuniformdeviate 1000,
\pdfuniformdeviate 1000, \pdfuniformdeviate 1000, \pdfuniformdeviate 1000)\relax
```

339, 694, 33, 664, 31, 921, 891, 763, 353, 188129929, 188129929

With `\xintverbosetrue` we will find in the log:

```
Function det3 for \xintexpr parser associated to \XINT_expr_userfunc_det3 with meaning macro:#1#2#3#4#5#6#7#8#9->\xintSub {\xintSub {\xintSub {\xintAdd {\xintAdd {\xintMul {\xintMul {#1}{#5}}{#9}}{\xintMul {\xintMul {#2}{#6}}{#7}}}{\xintMul {\xintMul {#3}{#4}}{#8}}}{\xintMul {\xintMul {#2}{#4}}{#9}}}{\xintMul {\xintMul {#3}{#5}}{#7}}}{\xintMul {\xintMul {#1}{#6}}{#8}}
```

```
Function det3bis for \xintexpr parser associated to \XINT_expr_userfunc_det3bis with meaning macro:#1#2#3#4#5#6#7#8#9->\xintAdd {\xintSub {\xintMul {#1}{\xintExpandArgs {XINT_expr_userfunc_det2}{#5}{#6}{#8}{#9}}}{\xintMul {#2}{\xintExpandArgs {XINT_expr_userfunc_det2}{#4}{#6}{#7}{#9}}}}{\xintMul {#3}{\xintExpandArgs {XINT_expr_userfunc_det2}{#4}{#5}{#7}{#8}}}
```


Lists, including Python-like selectors, are compatible with `\xintNewExpr`:⁷⁷

```
\xintNewExpr\Foo[5]{\empty[#1..[#2]..#3][#4:#5]}
\begin{itemize}[nosep]
\item |\Foo{1}{3}{90}{20}{30}|->|\Foo{1}{3}{90}{20}{30}
\item |\Foo{1}{3}{90}{-40}{-15}|->|\Foo{1}{3}{90}{-40}{-15}
\item |\Foo{1.234}{-0.123}{-10}{3}{7}|->|\Foo{1.234}{-0.123}{-10}{3}{7}
\end{itemize}
\fddef\test {\Foo {0}{10}{100}{3}{6}}\meaning\test +++
• \Foo{1}{3}{90}{20}{30}->61, 64, 67, 70, 73, 76, 79, 82, 85, 88
• \Foo{1}{3}{90}{-40}{-15}->1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49, 52,
  55, 58, 61, 64, 67, 70, 73
• \Foo{1.234}{-0.123}{-10}{3}{7}->865[-3], 742[-3], 619[-3], 496[-3]
macro:->30, 40, 50+++
```

In this last example the macro `\Foo` will not be able to handle an empty `#4` or `#5`: this is only possible in an expression, because the parser identifies `][:` or `:]` and handles them appropriately. During the construction of `\Foo` the parser will find `][#4:` and not `][:.`

The `\xintdeffunc`, `\xintdefiifunc`, `\xintdeffloatfunc` declarators added to xintexpr since release 1.2c are based on the same underlying mechanism as `\xintNewExpr`, `\xintNewIExpr`, ... The discussion that follows applies to them too.

10.15.1 Conditional operators and `\NewExpr`

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

```
\xintNewExpr\Formula [3]{ if((#1>#2) && (#2>#3), sqrt(#1-#2)*sqrt(#2-#3), #1^2+#3/#2) }%
\printnumber{\fixmeaning\Formula }
macro:#1#2#3->\romannumeral`^^@\xintSPRaw::csv {\xintiifNotZero {\xintAND {\xintGt {#1}{#2}}
{\xintGt {#2}{#3}}}{\xintMul {\XINTinFloatSqrtdigits {\xintSub {#1}{#2}}}{\XINTinFloatSqrtdig
its {\xintSub {#2}{#3}}}}{\xintAdd {\xintPow {#1}{2}}{\xintDiv {#3}{#2}}}}
```

This formula (with its `\xintiifNotZero`) will gobble the false branch without evaluating it when used with given arguments.

Remark: the meaning above reveals some of the private macros used by the package. They are not for direct use.

Another example

```
\xintNewExpr\myformula[3]{ ifsgn(#1,#2/#3,#2-#3,#2*#3) }%
\fixmeaning\myformula
macro:#1#2#3->\romannumeral`^^@\xintSPRaw::csv {\xintiifSgn {#1}{\xintDiv {#2}{#3}}{\xintSub
{#2}{#3}}{\xintMul {#2}{#3}}}
```


Again, this macro gobbles the false branches, as would have the operator `??` inside an `\xintexpr` session.

10.15.2 External macros and `\xintNewExpr`; the `protect` function

For macros within such a created `xint`-formula macro, there are two cases:

- the macro does not involve the numbered parameters in its arguments: it may then be left as is, and will be evaluated once during the construction of the formula,
- it does involve at least one of the macro parameters as argument. Then:

⁷⁷ The `\empty` token is optional here, but it would be needed in case of `\xintNewFloatExpr` or `\xintNewIExpr`.

 the whole thing (macro + argument) should be `protect`-ed, not in the \TeX sense (!), but in the following way: `protect(\macro {#1})`.

Here is a silly example illustrating the general principle: the macros here have equivalent functional forms which are more convenient; but some of the more obscure package macros of `xint` dealing with integers do not have functions pre-defined to be in correspondance with them, use this mechanism could be applied to them.

```
\xintNewExpr\formulaA[2]{protect(\xintRound{#1}{#2}) - protect(\xintTrunc{#1}{#2})}%
\printnumber{\fixmeaning\formulaA}

\xintNewIIExpr\formulaB [3]{rem(#1,quo(protect(\the\numexpr #2\relax),#3))}%
\noindent\printnumber{\fixmeaning\formulaB }
macro:#1#2->\romannumeral`^^@\xintSPRaw::csv {\xintSub {\xintRound {#1}{#2}}{\xintTrunc {#1}{#2}}}
macro:#1#2#3->\romannumeral`^^@\xintCSV::csv {\xintiiRem {#1}{\xintiiQuo {\the \numexpr #2\relax }{#3}}}
```

Only macros involving the `#1`, `#2`, etc. . . should be protected in this way; the `+`, `*`, etc. . . symbols, the functions from the `\xintexpr` syntax, none should ever be included in a protected string.

10.15.3 Limitations of `\NewExpr` and `\deffunc`

`\xintNewExpr` will pre-evaluate everything as long as it does not contain the macro parameters `#1`, `#2`, . . . and the special measures to take when these are inside branches to `?` and `??` (replace these operators by `if` and `ifsgn`) or as arguments to macros external to `xintexpr` (use `protect`) were discussed in [subsubsection 10.15.1](#) and [subsubsection 10.15.2](#).

The main remaining limitation is that expressions with dummy variables are compatible with `\xintNewExpr` only to the extent that the iterated-over list of values does not depend on the macro parameters `#1`, `#2`, . . . For example, this works:

```
\xintNewExpr \FA [2] {reduce(add((t+#1)/(t+#2), t=0..5))}
\FA {1}{1}, \FA {1}{2}, \FA {2}{3}
6, 617/140, 1339/280 but the 5 can not be abstracted into a third argument #3.
```

There are no restriction on using macro parameters `#1`, `#2`, . . . with list constructs. For example, this works:

```
\xintNewIExpr \FB [3] {[4] `+`([1/3..[#1/3]..#2]*#3)}
\begin{itemize}[nosep]
\item \FB {1}{10/3}{100} % (1/3+2/3+...+10/3)*100
\item \FB {5}{5}{20} % (1/3+6/3+11/3)*20
\item \FB {3}{4}{1} % (1/3+4/3+7/3+10/3)*1
\end{itemize}
• 1833.3333
• 120.0000
• 7.3333
```

Some simple expressions with `add` or `mul` can be also expressed with ``+`` and ``*`` and list operations. But there is no hope for `seq`, `iter`, etc. . . if the `#1`, `#2`, . . . are used inside the list argument: `seq(x(x+#1)(x+#2),x=1..#3)` is currently not compatible with `\xintNewExpr`. But `seq(x(x+#1)(x+#2), x=1..10)` has no problem.

All the preceding applies identically for `\xintdeffunc`, `\xintdefiifunc`, `\xintdeffloatfunc` which share the same routines as `\xintNewExpr`, `\xintNewIIExpr`, . . ., replacing the `#1`, `#2`, . . . in the discussion by the letters used as function arguments.

Here is a final syntax restriction: it is possible to use sub-expressions only if they use `\xintexpr`, those with `\xinttheexpr`, or `\xinteval` are illegal.

```
\xintNewExpr \FC [4] {#1+\xintexpr #2*#3\relax + #4}
\printnumber{\fixmeaning\FC}
```


10.17 `\xintNewIExpr`

Like `\xintNewExpr` but using `\xinttheiexpr`. As `\xintiexpr` admits an optional rounding argument `[N]` the same caveat when square brackets come first in the expression as in the discussion of `\xintNewFloatExpr` applies.

10.18 `\xintNewIIExpr`

Like `\xintNewExpr` but using `\xinttheiiexpr`.

10.19 `\xintNewBoolExpr`

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

10.20 The `\xintdefvar`, `\xintdefiivar`, `\xintdeffloatvar` macros

See [subsection 2.6](#) for their documentation.

10.21 The `\xintdeffunc`, `\xintdefiifunc`, `\xintdeffloatfunc` macros

See [subsubsection 2.7.1](#) for their documentation.

10.22 The `\xintNewFunction` macro

See [subsubsection 2.7.14](#) for its documentation.

10.23 Technicalities

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

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

The `\escapechar` setting may be arbitrary when using `\xintexpr`.

The format of the output of `\xintexpr<stuff>\relax` is a ! (with catcode 11) followed by various things:

```
\edef\l { \xintexpr 1.23^10\relax } \meaning\l
macro:->!\XINT_expr_usethe \XINT_protectii \XINT_expr_print \.=792594609605189126649/1[-20]
```

Note that `\xintexpr` expands in an `\edef`, contrarily to `\numexpr` which is non-expandable, if not prefixed by `\the`, `\number`, or `\romannumeral` or in some other context where \TeX is building a number. See [subsection 2.10](#) for some illustration.


I decided to put all intermediate results (from each evaluation of an infix operators, or of a parenthesized subpart of the expression, or from application of the minus as prefix, or of the exclamation sign as postfix, or any encountered braced material) inside `\csname...\endcsname`, as this can be done expandably and encapsulates an arbitrarily long fraction in a single token (left with undefined meaning), thus providing tremendous relief to the programmer in his/her expansion control.

As the `\xintexpr` computations corresponding to functions and infix or postfix operators are done inside `\csname...\endcsname`, the *f-expandability* could possibly be dropped and one could imagine implementing the basic operations with expandable but not *f-expandable* macros (as `\xintXTrunc`.) I have not investigated that possibility.

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

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

However, this mechanism is completely inoperant for parentheses involved in the syntax of the `seq`, `add`, `mul`, `subs`, `rseq` and `rrseq` functions, and missing parentheses may cause the parser to fetch tokens beyond the ending `\relax` necessarily ending up in cryptic low-level TeX-errors.

 Note that the `,<letter>=` part must be visible, it can not arise from expansion (the equal sign does not have to be an equal sign, it can be any token and will be gobbled). However for `iter`, `iter2`, `r`, `rseq`, `rrseq`, the initial values delimited by a `;` are parsed in the normal way, and in particular may be braced or arise from expansion. This is useful as the `;` may be hidden from `\xintdeffunc` as `{;}` for example. Again, this remark does *not* apply to the comma `,` which precedes the `<letter>=` part. The comma will be fetched by delimited macros and must be there. Nesting is handled by checking (again using suitable delimited macros) that parentheses are suitably balanced.

Note that `\relax` is *mandatory* (contrarily to the situation for `\numexpr`).

10.24 Acknowledgements (2013/05/25)

I was greatly helped in my preparatory thinking, prior to producing such an expandable parser, by the commented source of the `l3fp` package, specifically the `l3fp-parse.dtx` file (in the version of April-May 2013; I think there was in particular a text called 'roadmap' which was helpful). Also the source of the `calc` package was instructive, despite the fact that here for `\xintexpr` the principles are necessarily different due to the aim of achieving expandability.

11 Macros of the xintrig package

<p>.1 <code>\xintreloadxintrig</code>..... 140</p> <p>.2 Constants..... 140</p>		<p>.3 Functions..... 141</p> <p>.4 Important implementation notes..... 143</p>
---	--	--

This package provides trigonometric functions for use with *xintexpr*. The sole macro is `\xintreloadxintrig`.

This package was first included in release 1.3e (2019/04/01) of *xintexpr*. It is automatically loaded by *xintexpr*.

Acknowledgements I finally decided to release some such functions under friendly pressure of Jürgen GILG and Thomas SÖLL, let them both be thanked here.

11.1 `\xintreloadxintrig`



After modifying `\xintDigits`, one must issue `\xintreloadxintrig` to let the package re-configure itself.

The library is pre-configured to be able to handle a precision of up to about 60 digits (make this 59 at most for the inverse functions). But absence of guard digits (whether in the used hard-coded constants or in passing over values from one auxiliary function to the next) due to high level (user) interface used for the programming means that the produced values are definitely expected to be wrong in the last digit or last two digits. I should actually give some estimate of the actual maximal error in *ulps* unit, but I have not done the complete analysis for lack of time.

Final computation results should thus probably be printed via `\xintfloateval{[-2]}...` in order to strip off (with rounding) the last two digits, if one does not like seeing those non-meaningful figures in the last one or two positions (I don't say those last two figures are *systematically* off). For example, to achieve 16 digits of precision one should work with a precision of 18 digits (being careful to have issued `\xintreloadxintrig`) and round results using `\xintfloateval{[-2].2}...`.

Another approach is to use `\xintieval{[D]}...` for conversion to a fixed point format.

In future, lower level coding will probably replace the high-level interface, or at least the macros produced by the high-level interface will be hacked into to tell the float macros to work at a somewhat elevated precision.

11.2 Constants

They are the correct rounding to `\xintDigits` precision of the mathematically exact ones. Their values get incorporated into the trigonometrical functions at the time of their definitions during loading or reloading of the package. They are left free to use, or modified, or `\xintunassignvar`'d, as this will have no impact whatsoever on the functions.

twoPi what could that be?

threePiover2

Pi

Piover2

oneRadian this is one radian in degrees: $180/\pi$

oneDegree this is one degree in radian: $\pi/180$

invfact2 this is 1/2!

invfact3 this is 1/3!

...

invfact44 this is 1/44!

For a (very) slight optimization of usage, it is recommended to convert them to macro form, for example:

```
\edef\oneDegree{\xintfloatexpr oneDegree\relax}
\xintfloateval{\sin(37\oneDegree)}\newline
\xintfloateval{\sind(37)}\newline
```

0.6018150231520484

0.6018150231520484

By the way, the above value differs by [1ulp](#) from correct rounding of exact one (which looks [...52048279917...](#)), see [subsection 11.4](#).

11.3 Functions

11.3.1 Direct trigonometry

With the variable in radians:

sin(x) sine

cos(x) cosine

tan(x) tangent

cot(x) cotangent

sec(x) secant

csc(x) cosecant

With the variable in degrees:

sind(x) sine

cosd(x) cosine

tand(x) tangent

cotd(x) cotangent

secd(x) secant

cscd(x) cosecant

Only available with the variable in radians:

tg(x) tangent

cotg(x) cotangent

sinc(x) cardinal sine $\text{sinc}(x) = \sin(x)/x$

11.3.2 Inverse trigonometry

With the value in radians:

asin(x) arcsine

acos(x) arccosine

atan(x) arctangent

Arg(x, y) the main branch of the argument of the complex number $x+iy$, from $-\pi$ (excluded) to π (included). Inherent rounding of output makes $-\text{Pi}$ a possible return value.

pArg(x, y) the branch of the argument of the complex number $x+iy$ with values going from 0 (included) to 2π (excluded). Inherent rounding makes twoPi a possible return value.

atan2(y, x) it is **Arg(x, y)**. Note the reversal of the arguments, this seems to be the most frequently encountered convention across languages.

With the value in degrees:

asind(x) arcsine

acosd(x) arccosine

atand(x) arctangent

Argd(x, y) the main branch of the argument of the complex number $x+iy$, from -180 (excluded) to 180 (included). Inherent rounding of output can cause -180 to be returned.

pArgd(x, y) the branch of the argument of the complex number $x+iy$ with values going from 0 (included) to 360 (excluded). Inherent rounding of output can cause 360 to be returned.

atan2d(y, x) it is **Arg(x, y)**. Note the reversal of the arguments, this seems to be the most frequently encountered convention across languages.

11.3.3 Conversion functions (optional definitions left to user decision)

Python provides functions **degrees()** and **radians()**. But as most of the **xinttrig** functions are already defined for the two units, I felt this was not really needed. It is a oneliner to add them:

```
\xintdeffloatfunc radians(x) := x * oneDegree;
\xintdeffloatfunc degrees(x) := x * oneRadian;
\xintdefefunc radians(x) := x * oneDegree;
\xintdefefunc degrees(x) := x * oneRadian;
```

The variants for **\xintexpr** above do an exact multiplication, I did not add a **float()** wrapper to force rounding as anyhow the trigonometrical functions will do this initial rounding of their arguments. But if you define a variable for multiple later use using such a **degrees()** function, it would be better to add a **float()** wrapper in the variable definition so the rounding is already done: rounding an already rounded value is unavoidable overhead but proceeds faster as it is quickly realized the input actually needs no rounding.

Notice however that the conversion factors above are without guard digits. One can do this:

```
\xintdeffloatfunc radians(x) := float(\xintexpr x * oneDegreewithmoredigits\relax);
\xintdeffloatfunc degrees(x) := float(\xintexpr x * oneRadianwithmoredigits\relax);
```

But recall that **x** will normally already be a rounded value, so this is perhaps a bit complex for not much ado. Probably better to work overall with an elevated precision and print final results at a lower precision.

11.4 Important implementation notes

- The package is almost entirely implemented using the high level user interface of [xintexpr](#), see [section 2](#) for [\xintdeffloatefunc](#) and [\xintdeffloatvar](#), the main two exceptions are:
 1. the range reduction for the [sind\(\)](#) and [cosd\(\)](#) functions which required for optimized efficiency the coding at some more core level.
 2. a change at core level was done to [\xintdeffunc](#) in order to facilitate the transfer of the defined functions from the float parser to the exact parser. See [subsubsection 10.15.3](#), the source code comments in [sourcexint.pdf](#) and the discussion of [\xintdeffunc](#) for details. The [\xintdefefunc](#) added at 1.3e was also motivated by this context.

To avoid problems if the package is reloaded at a time the user has used some letter variables as assigned variables, I added [\xintensuredummy](#) and [\xintrestorelettervar](#).

- It is not possible from this interface to (easily) let the computation proceed with a temporarily elevated precision ('`guard digits`'). Expect thus some errors in the last places; basically one should use the optional rounding argument of either [\xintfloateval](#) or [\xintieval](#) to reduce the number of digits of printed values by about two digits, if one hopes to get correct rounding (most of the time).
- Currently, [xint](#) is lacking some dedicated internal representation of floats which means that most operations re-parse the digit tokens of their arguments to count them. . . this does not contribute to efficiency (you can load the module under [\xintverbosetrue](#) regime and see how the nested macros look like and get an idea of how many times some rather silly re-counting of mantissa lengths will get done!)
- One should not overwrite some function names which are employed as auxiliaries: [sin_aux](#), [cos_0](#), [aux](#), [sin_](#), [cos_](#), [sind_](#), [cosd_](#), [asin_1](#). . . others. . . [asin_a](#), [asind_a](#), [atan_a](#), [atand_a](#), [atan_b](#), [atand_b](#). If you redefine any one of them, you break the whole thing.
- Floats with large exponents are integers and are multiple of 1000; hence modulo 360 all such ``angles`` are multiple of 40 degrees. Needless to say that considering usage of the sine and cosine functions with such large float numbers is meaningless.
- Regarding such a big float angle in radians, [xinttrig](#) converts it to degrees by multiplication by (pre-rounded) $180/\pi$, then it does range reduction modulo 360 and goes back to radians in the appropriate octants and use the series (roughly said). Thus [cos\(\)](#) and [sin\(\)](#) will be evaluated as if at some of the finitely many rounded multiples of (exact) $2\pi/9$. When the unit in the last place of the original input was for example $1e9$ it is clear that the final result means nothing at all; this intrinsic problem is not one of conversion from radians to degrees: the unit in the last place interval extends above possibly astronomical numbers of intervals of length 2π . This is an in-built inadequacy of (large) floating point numbers for trigonometrical evaluations; the argument should be treated then as a uniformly distributed random variable modulo 2π , and the sine and cosine values should be random variables realizing the value distribution of these mathematical functions. Clearly this adds some programming complication of deciding how to make the transition. Too lazy for that.
- Did I say the implementation was done at very high level (for the most part), hence has ample room for optimization? This is particularly the case for the handling of small inputs by functions such as sine or arcsine.

12 Macros of the `xintlog` package

.1	<code>\poormanloghack</code>	144		.2	Functions	144
----	------------------------------------	-----	--	----	-----------------	-----

This package provides logarithms, exponentials and fractional powers for use with `xintexpr`.

This package was first included in release 1.3e (2019/04/01) of `xintexpr`. It is automatically loaded by `xintexpr`.

Currently it is a wrapper to import package `poormanlog` which computes with 8 or 9 digits of precision, adding the `log()`, `exp()`, and `pow()` function to the `log10()` and `pow10()` provided by the package.

12.1 `\poormanloghack`

`\poormanloghack{**}` use it to let the `**` operator be remapped to the `pow()` function.

`\poormanloghack{^}` use it to let the `^` operator be remapped to the `pow()` function.

If used, they obey \TeX scoping as usual.

```
\begingroup
\poormanloghack{**}\xintfloateval{[8]1.234**5.678}\newline
\poormanloghack{^}\xintfloateval{[8]1.234^5.678}\par
\endgroup
% ** and ^ now do not accept fractional exponents: only half-integer ones and
% only in \xintfloateval, not \xinteval.
```

3.2997989

3.2997989

Notice that in `\xintfloateval` those (equivalent) operators already natively handle half-integer exponents. Once remapped to the `pow()` function they will become less precise than the original ones for half-integer and integer exponents.

12.2 Functions

All those functions achieve only about 8 or 9 digits of precision. Notice in particular that the digits beyond the ninth printed by `log()` have no significance (here we suppose $1 < x < 10$), but I did not add the rounding overhead as it is expected anyhow that the final result will be appropriately rounded. Notice however that `log10()` should be seen as going from floating point to fixed point (in the sense of the number of fractional digits) and `pow10()` from fixed point to floating point.

log10(x) logarithm in base 10

pow10(x) fractional powers of 10

log(x) natural logarithm via `log10(x)*2.3025850923` formula; only the first 8 or 9 digits of the output are significant...

exp(x) exponential function via `pow10(x*0.434294481903)` formula

pow(x, y) computes x^y via the formula `pow10(y*log10(x))`

```
\xintfloateval{[9] log(2), exp(1), pow(2,0.5)}
0.693147179, 2.71828183, 1.41421356 Notice that the last digit of log(2) is not the correctly
rounded one... I did say 9 or 8 digits or precision... The documentation of poormanlog mentions an
error of up to 2 units in the ninth digit when computing log10(x) for 1 < x < 10 and 10^x for 0 < x < 1.
```

13 Macros of the xintbinhex package

<p>.1 \xintDecToHex 146 .2 \xintDecToBin 146 .3 \xintHexToDec 146 .4 \xintBinToDec 146</p>	<p>.5 \xintBinToHex 146 .6 \xintHexToBin 147 .7 \xintCHexToBin 147</p>
---	--

This package provides expandable conversions of (big) integers to and from binary and hexadecimal.

First version of this package was in the 1.08 (2013/06/07) release of *xint*. Its routines remained un-modified until their complete rewrite at release 1.2m (2017/07/31). The new macros are faster, using techniques from the 1.2 (2015/10/10) release of *xintcore*. But the inputs are now limited to a few thousand digits, whereas the 1.08 could handle (slowly...) tens of thousands of digits.

Table 3 recapitulates the maximal allowed sizes (they got increased at 1.2n): for macro `\xintFooToBar` in the first column, the value in the second column is the maximal N such that `\edef\X{\xintFooToBar{<N digits>}}` does not raise an error with standard \TeX memory parameters (input stack size=5000, expansion depth=10000, parameter stack size=10000). The tests were done with TL2017 and *etex*. Nested calls will allow slightly lesser values only. The third column gives the corresponding maximal size of output. The fourth column gives the \TeX parameter cited in the error message when trying with $N+1$ digits.

	Max length of input	-> length of output	Limiting factor
<code>\xintDecToHex</code>	6014	4995	input stack size=5000
<code>\xintDecToBin</code>	6014	19979	input stack size=5000
<code>\xintHexToDec</code>	8298	9992	input stack size=5000
<code>\xintBinToDec</code>	19988	6017	input stack size=5000
<code>\xintBinToHex</code>	19988	4997	input stack size=5000
<code>\xintHexToBin</code>	4996	19984	input stack size=5000
<code>\xintCHexToBin</code>	4997	19988	input stack size=5000

Table 3: Maximal sizes of inputs (at 1.2n) for xintbinhex macros

Roughly, base 10 numbers are limited to 6000 digits, hexadecimal numbers to (almost) 5000 digits, and binary numbers to (almost) 20000 digits. With the surprising exception of `\xintHexToDec` which allows almost 8300 hexadecimal digits on input.

The argument is first *f-expanded*. It may optionally have a unique leading minus sign (a plus sign is not allowed), and leading zeroes.

An input (possibly signed) with no leading zeroes is guaranteed to give an output without leading zero, with the sole, deliberate, exception of `\xintCHexToBin`: from N hexadecimal digits it produces $4N$ binary digits, hence possibly with up to three leading zeroes (if the input had none.)

Inputs with leading zeroes usually produce outputs with an unspecified, case-dependent, number of leading zeroes (`\xintBinToHex` always uses the minimal number of hexadecimal digits needed to represent the binary digits, inclusive of leading zeroes if present.)

The macros converting from binary or decimal are robust against non terminated inputs like `\the\numexpr 2+3` or `\the\mathcode`\-.`. The macro `\xintHexToDec` also but not `\xintHexToBin` and `\xintCHexToBin` (anyway there are no primitive in (e)- \TeX to my knowledge which will generate hexadecimal digits and may force expansion of next token).

Hexadecimal digits *A..F* must be in uppercase. Category code for them on input may be *letter* or *other*. On output they are of category code *letter*, and in uppercase.

Low-level unrecoverable errors will happen if for example a supposedly binary input contains other digits than 0 and 1. Inputs can not start with a `0b`, `0x`, `#x`, " or similar prefix: only digits/letters according to the binary, decimal, or hexadecimal notation.

With this package loaded additionally to `xintexpr`, hexadecimal input is possible in expressions: simply by using the prefix `"`. Such hexadecimal numbers may have a fractional part. Lowercase hexadecimal letters are currently *not* recognized as such in expressions. Currently the `p` postfix notation from standard programming languages standing for an extra power of two multiplicand is not implemented.

13.1 `\xintDecToHex`

f ★ Converts from decimal to hexadecimal.

```
\xintDecToHex{2718281828459045235360287471352662497757247093699959574966967627724076630353}
547594571382178525166427427466391932003}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C6032936BF37DAC918}
814C63
```

13.2 `\xintDecToBin`

f ★ Converts from decimal to binary.

```
\xintDecToBin{2718281828459045235360287471352662497757247093699959574966967627724076630353}
547594571382178525166427427466391932003}
->1000110101001001110010111100011001101001010010011010100101110000010100011111011110100001}
01010000001011110010001010011100011111000001011000101111100010000011011000100011100010010001}
0111010111011100101011010101110110000010111011001110001101001001110010111101000110110111001}
110010001101100011000000011001010010011011010111110011011110110101100100100011000100000010}
100110001100011
```

13.3 `\xintHexToDec`

f ★ Converts from hexadecimal to decimal.

```
\xintHexToDec{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C603}
2936BF37DAC918814C63}
->271828182845904523536028747135266249775724709369995957496696762772407663035354759457138217}
8525166427427466391932003
```

13.4 `\xintBinToDec`

f ★ Converts from binary to decimal.

```
\xintBinToDec{1000110101001001110010111100011001101001010010011010100101110000010100011111}
10111110100001010100000010111100100010100111000111110000010110001011111000100000110110001000}
11100010010001011101011101111001010110101011101100000101110110011100011010010011100101111010}
001101101110011100100011011000110000000110010100100110110101111100110111101101011001001000}
11000100000010100110001100011}
->271828182845904523536028747135266249775724709369995957496696762772407663035354759457138217}
8525166427427466391932003
```

13.5 `\xintBinToHex`

f ★ Converts from binary to hexadecimal. The input is first zero-filled to `4N` binary digits, hence the output will have `N` hexadecimal digits (thus, if the input did not have a leading zero, the output will not either).

```
\xintBinToHex{1000110101001001110010111100011001101001010010011010100101110000010100011111}
10111110100001010100000010111100100010100111000111110000010110001011111000100000110110001000}
11100010010001011101011101111001010110101011101100000101110110011100011010010011100101111010}
11100010010001011101011101111001010110101011101100000101110110011100011010010011100101111010}
```

```
00110110111001110010001101100011000000011001010010011011010111111001101111101101011001001000
11000100000010100110001100011}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C6032936BF37DAC918
814C63
```

13.6 \xintHexToBin

f ★ Converts from hexadecimal to binary. Up to three leading zeroes of the output are trimmed.

```
\xintHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C603
2936BF37DAC918814C63}
->10001101010010011100101111000110011010010100100110101001011100000101000111110111110100001
01010000001011110010001010011100011111000001011000101111100010000011011000100011100010010001
0111010111011100101011010101110110000010111011001110001101001001110010111101000110110111001
1100100011011000110000000110010100100110110101111100110111110110101100100100011000100000010
100110001100011
```

13.7 \xintCHexToBin

f ★ Converts from hexadecimal to binary. Same as `\xintHexToBin`, but an input with N hexadecimal digits will give an output with exactly $4N$ binary digits, leading zeroes are not trimmed.

```
\xintCHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C60
32936BF37DAC918814C63}
->000100001101010010011100101111000110011010010100100110101001011100000101000111110111110100
00101010000001011110010001010011100011111000001011000101111100010000011011000100011100010010
001011101011101110010101101010111011000001011101100111000110100100111001011110100011011011
0011100100011011000110000000110010100100110110101111100110111110110101100100100011000100000
010100110001100011
```

This can be combined with `\xintBinToHex` for round-trips preserving leading zeroes for $4N$ binary digits numbers, whereas using `\xintHexToBin` gives reproducing round-trips only for $4N$ binary numbers not starting with `0000`.

This zero-fills to $4N$ digits the input, hence gives here a leading zero in output:

```
\xintBinToHex{0001111}\newline
Chaining, we end up with  $4N-3$  digits, as three binary zeroes are trimmed:
\xintHexToBin{\xintBinToHex{0001111}}\newline
But this will always reproduce the initial input zero-filled to length  $4N$ :
\xintCHexToBin{\xintBinToHex{0001111}}\par
Another example (visible space characters manually inserted):\newline
$00000000111101001010001\xrightarrow{\text{\string\xintBinToHex}}
\xintBinToHex{00000000111101001010001}\xrightarrow{\text{\string\xintHexToBin\hphantom{X}}}
\text{\textvisiblespace\textvisiblespace\textvisiblespace}
\xintHexToBin{\xintBinToHex{00000000111101001010001}}$\newline
$00000000111101001010001\xrightarrow{\text{\string\xintBinToHex}}
\xintBinToHex{00000000111101001010001}\xrightarrow{\text{\string\xintCHexToBin}}
\xintCHexToBin{\xintBinToHex{00000000111101001010001}}$
\par
```

This zero-fills to $4N$ digits the input, hence gives here a leading zero in output: 0F

Chaining, we end up with $4N-3$ digits, as three binary zeroes are trimmed: 01111

But this will always reproduce the initial input zero-filled to length $4N$: 00001111

Another example (visible space characters manually inserted):

```
00000000111101001010001  $\xrightarrow{\text{\string\xintBinToHex}}$  00FA51  $\xrightarrow{\text{\string\xintHexToBin}}$  00000000111101001010001
00000000111101001010001  $\xrightarrow{\text{\string\xintBinToHex}}$  00FA51  $\xrightarrow{\text{\string\xintCHexToBin}}$  00000000111101001010001
```

14 Macros of the xintgcd package

<p>.1 <code>\xintiiGCD</code>..... 148</p> <p>.2 <code>\xintGCD</code>..... 148</p> <p>.3 <code>\xintGCDof</code>..... 148</p> <p>.4 <code>\xintiiLCM</code>..... 148</p> <p>.5 <code>\xintLCM</code>..... 148</p> <p>.6 <code>\xintLCMof</code>..... 149</p>	<p>.7 <code>\xintBezout</code>..... 149</p> <p>.8 <code>\xintEuclideanAlgorithm</code>..... 149</p> <p>.9 <code>\xintBezoutAlgorithm</code>..... 149</p> <p>.10 <code>\xintTypesetEuclideanAlgorithm</code>..... 149</p> <p>.11 <code>\xintTypesetBezoutAlgorithm</code>..... 150</p>
---	---

This package was included in the original release 1.0 (2013/03/28) of the *xint* bundle.

Since release 1.09a the macros filter their inputs through the `\xintNum` macro, so one can use count registers, or fractions as long as they reduce to integers.

Since release 1.1, the two ```typeset``` macros require the explicit loading by the user of package *xinttools*.

At 1.3d macros `\xintiiGCD` and `\xintiiLCM` are copied over to *xint*, hence `gcd()` and `lcm()` functions in `\xintiexpr` are available simply from loading *xintexpr*, xintgcd is not needed.

New with
1.3d

14.1 `\xintiiGCD`

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

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

New with
1.3d

With release 1.3d, this macro is also available from loading *xint*, hence also with *xintexpr*, as it used by the `gcd()` function in `\xintiexpr`, hence removes a dependency of *xintexpr* on xintgcd.

14.2 `\xintGCD`

^{Num Num}
f f ★ `\xintGCD` uses `\xintNum` overhead to make its arguments into strict integers first. With *xintfrac* loaded this conversion means truncation to integers.

14.3 `\xintGCDof`

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

14.4 `\xintiiLCM`

ff ★ `\xintiiLCM{N}{M}` computes the least common multiple of integers. It is positive, except if one *N* or *M* vanishes, in which case the macro returns zero.

```
\xintiiLCM{10000}{1113}=11130000
\xintiiLCM{123456789012345}{9876543210321}=406442103762636081733470915
```

New with
1.3d

With release 1.3d, this macro is also available from loading *xint*, hence also with *xintexpr*, as it used by the `lcm()` function in `\xintiexpr`, hence removes a dependency of *xintexpr* on xintgcd.

14.5 `\xintLCM`

^{Num Num}
f f ★ `\xintLCM` uses `\xintNum` overhead to make its arguments into strict integers first. With *xintfrac* loaded this conversion means truncation to integers.

14.6 \xintLCMof

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

14.7 \xintBezout

$f f$ ^{Num Num} \star `\xintBezout{N}{M}` returns three numbers U, V, D within braces where D is the (non-negative) GCD, and $UN + VM = D$.

```
\oodef\X{\xintBezout {10000}{1113}}\meaning\X\par
\xintAssign {\xintBezout {10000}{1113}}\to\U\V\D
U: \meaning\U, V: \meaning\V, D: \meaning\D\par
AU+BV: \xinttheiexpr 10000*\U+1113*\V\relax\par
\noindent\oodef\X{\xintBezout {123456789012345}{9876543210321}}\meaning\X\par
\xintAssign \X\to\U\V\D
U: \meaning\U, V: \meaning\V, D: \meaning\D\par
AU+BV: \xinttheiexpr 123456789012345*\U+9876543210321*\V\relax
macro:->{-131}{1177}{1}
U: macro:->-131, V: macro:->1177, D: macro:->1
AU+BV: 1
macro:->{256654313730}{-3208178892607}{3}
U: macro:->256654313730, V: macro:->-3208178892607, D: macro:->3
AU+BV: 3
```

14.8 \xintEuclideanAlgorithm

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

```
\edef\X{\xintEuclideanAlgorithm {10000}{1113}}\meaning\X
macro:->{5}{10000}{1}{1113}{8}{1096}{1}{17}{64}{8}{2}{1}{8}{0}
The first item 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.
```

14.9 \xintBezoutAlgorithm

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

```
\edef\X{\xintBezoutAlgorithm {10000}{1113}}\printnumber{\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}{11}
77}{131}{8}{0}{10000}{1113}
The first item 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.
```

14.10 \xintTypesetEuclideanAlgorithm

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



```
Usage of this macro requires the user to load xinttools.
\xintTypesetEuclideanAlgorithm {123456789012345}{9876543210321}
123456789012345 = 12 × 9876543210321 + 4938270488493
9876543210321 = 2 × 4938270488493 + 2233335
```

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

14.11 `\xintTypesetBezoutAlgorithm`

Num Num
f f

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



Usage of this macro requires the user to load `xinttools`.

```

\xintTypesetBezoutAlgorithm {10000}{1113}
10000 = 8 × 1113 + 1096
  8 = 8 × 1 + 0
  1 = 8 × 0 + 1
1113 = 1 × 1096 + 17
  9 = 1 × 8 + 1
  1 = 1 × 1 + 0
1096 = 64 × 17 + 8
 584 = 64 × 9 + 8
  65 = 64 × 1 + 1
  17 = 2 × 8 + 1
1177 = 2 × 584 + 9
 131 = 2 × 65 + 1
   8 = 8 × 1 + 0
10000 = 8 × 1177 + 584
1113 = 8 × 131 + 65
131 × 10000 - 1177 × 1113 = -1
    
```

15 Macros of the xintseries package

<p>.1 \xintSeries..... 151</p> <p>.2 \xintiSeries 152</p> <p>.3 \xintRationalSeries..... 153</p> <p>.4 \xintRationalSeriesX 156</p> <p>.5 \xintPowerSeries 157</p> <p>.6 \xintPowerSeriesX..... 159</p>	<p>.7 \xintFxpPtPowerSeries 159</p> <p>.8 \xintFxpPtPowerSeriesX 160</p> <p>.9 \xintFloatPowerSeries 161</p> <p>.10 \xintFloatPowerSeriesX 162</p> <p>.11 Computing log 2 and π..... 162</p>
---	---

This package was first released with version 1.03 (2013/04/14) of the *xint* bundle.

The $\frac{\text{Frac}}{f}$ expansion type of various macro arguments is only a $\frac{\text{Num}}{f}$ if only *xint* but not *xintfrac* is loaded. The macro `\xintiSeries` is special and expects summing big integers obeying the strict format, even if *xintfrac* is loaded.

The arguments serving as indices are of the $\frac{\text{num}}{X}$ expansion type.

In some cases one or two of the macro arguments are only expanded at a later stage not immediately.

Since 1.3, `\xintAdd` and `\xintSub` use systematically the least common multiple of the denominators. Some of the comments in this chapter refer to the earlier situation where often the denominators were simply multiplied together. *They have yet to be updated to reflect the new situation brought by the 1.3 release.* Some of these comments may now be off-synced from the actual computation results and thus may be wrong.

15.1 \xintSeries

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

```
\def\coeff #1{\xintiiMON{#1}/#1.5} % (-1)^n/(n+1/2)
\fddef\w {\xintSeries {0}{50}{\coeff}} % we want to re-use it
\fddef\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.
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n + \frac{1}{2}} = \xintFrac\z \]
```

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

The definition of `\coeff` as `\xintiiMON{#1}/#1.5` is quite suboptimal. It allows *#1* to be a big integer, but anyhow only small integers are accepted as initial and final indices (they are of the $\frac{\text{num}}{X}$ type). Second, when the *xintfrac* parser sees the *#1.5* it will remove the dot hence create a denominator with one digit more. For example $1/3.5$ turns internally into $10/35$ whereas it would be more efficient to have $2/7$. For info here is the non-reduced `\w`:

$$\frac{86954669143685470216056396050813301139357}{550137335796950015126399586130198645252875} 10^1$$

It would have been bigger still in releases earlier than 1.1: now, the *xintfrac* `\xintAdd` routine does not multiply blindly denominators anymore, it checks if one is a multiple of the other. However it does not practice systematic reduction to lowest terms.

A more efficient way to code `\coeff` is illustrated next.

```
\def\coeff #1{\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}%
% The [0] in \coeff is a tiny optimization: in its presence the \xintfracname parser
% sees something which is already in internal format.
\def\w {\xintSeries {0}{50}{\coeff}}
\[\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}}=\xintfrac\w\]
```

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

The reduced form `\z` as displayed above only differs from this one by a factor of 1.

```
\def\coeffleibnitz #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}
\cnta 1
\loop
% in this loop we recompute from scratch each partial sum!
% we can afford that, as \xintSeries is fast enough.
\noindent\hbox to 2em{\hfil\texttt{\the\cnta.} }%
    \xintTrunc {12}{\xintSeries {1}{\cnta}{\coeffleibnitz}}\dots
\endgraf
\ifnum\cnta < 30 \advance\cnta 1 \repeat
```

1. 1.000000000000...	11. 0.736544011544...	21. 0.716390450794...
2. 0.500000000000...	12. 0.653210678210...	22. 0.670935905339...
3. 0.833333333333...	13. 0.730133755133...	23. 0.714414166209...
4. 0.583333333333...	14. 0.658705183705...	24. 0.672747499542...
5. 0.783333333333...	15. 0.725371850371...	25. 0.712747499542...
6. 0.616666666666...	16. 0.662871850371...	26. 0.674285961081...
7. 0.759523809523...	17. 0.721695379783...	27. 0.711322998118...
8. 0.634523809523...	18. 0.666139824228...	28. 0.675608712404...
9. 0.745634920634...	19. 0.718771403175...	29. 0.710091471024...
10. 0.645634920634...	20. 0.668771403175...	30. 0.676758137691...

15.2 \xintiSeries

$\sum_{x=A}^{x=B} f \star$

`\xintiSeries{A}{B}{\coeff}` computes $\sum_{n=A}^{n=B} \text{\coeff}\{n\}$ where `\coeff{n}` must *f-expand* to a (possibly long) integer in the strict format.

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

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

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

```
\def\coeff #1{\xintiRound {40} % rounding at 40
  {\the\numexpr\ifodd #1 -2\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+\frac{1}{2}} \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]}%
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}}
  = \xintTrunc {50}{\xintiSeries {0}{50}{\exactcoeff}}\dots\]
```

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

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

This shows indeed that our sum of truncated terms estimated wrongly the 39th and 40th digits of the exact result⁷⁸ and that the sum of rounded terms fared a bit better.

15.3 `\xintRationalSeries`

`\xintRationalSeries{A}{B}{f}{\ratio}` evaluates $\sum_{n=A}^{n=B} F(n)$, where $F(n)$ is specified indirectly via the data of $f=F(A)$ and the one-parameter macro `\ratio` which must be such that `\macro{n}` expands to $F(n)/F(n-1)$. The name indicates that `\xintRationalSeries` was designed to be useful in the cases where $F(n)/F(n-1)$ is a rational function of n but it may be anything expanding to a fraction. The macro `\ratio` must be an expandable-only compatible macro and expand to its value after iterated full expansion of its first item. A and B are fed to a `\numexpr` hence may be count registers or arithmetic expressions built with such; they must obey the \TeX bound. The initial term f may be a macro `\f`, it will be expanded to its value representing $F(A)$.

```
\def\ratio #1{2/#1[0]}% 2/n, to compute exp(2)
\cnta 0 % previously declared count
\begin{quote}
\loop \fdef\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}\par
\ifnum\cnta<20 \advance\cnta 1 \repeat
\end{quote}
```

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

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

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

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

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

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

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

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

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

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

⁷⁸ 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.

$$\begin{aligned} \sum_{n=0}^{10} \frac{2^n}{n!} &= 7.388994708994\dots = \frac{26813184}{3628800} = \frac{34913}{4725} \\ \sum_{n=0}^{11} \frac{2^n}{n!} &= 7.389046015712\dots = \frac{294947072}{39916800} = \frac{164591}{22275} \\ \sum_{n=0}^{12} \frac{2^n}{n!} &= 7.389054566832\dots = \frac{3539368960}{479001600} = \frac{691283}{93555} \\ \sum_{n=0}^{13} \frac{2^n}{n!} &= 7.389055882389\dots = \frac{46011804672}{6227020800} = \frac{14977801}{2027025} \\ \sum_{n=0}^{14} \frac{2^n}{n!} &= 7.389056070325\dots = \frac{644165281792}{87178291200} = \frac{314533829}{42567525} \\ \sum_{n=0}^{15} \frac{2^n}{n!} &= 7.389056095384\dots = \frac{9662479259648}{1307674368000} = \frac{4718007451}{638512875} \\ \sum_{n=0}^{16} \frac{2^n}{n!} &= 7.389056098516\dots = \frac{154599668219904}{20922789888000} = \frac{1572669151}{212837625} \\ \sum_{n=0}^{17} \frac{2^n}{n!} &= 7.389056098884\dots = \frac{2628194359869440}{355687428096000} = \frac{16041225341}{2170943775} \\ \sum_{n=0}^{18} \frac{2^n}{n!} &= 7.389056098925\dots = \frac{47307498477912064}{6402373705728000} = \frac{103122162907}{13956067125} \\ \sum_{n=0}^{19} \frac{2^n}{n!} &= 7.389056098930\dots = \frac{898842471080853504}{121645100408832000} = \frac{4571749222213}{618718975875} \\ \sum_{n=0}^{20} \frac{2^n}{n!} &= 7.389056098930\dots = \frac{17976849421618118656}{2432902008176640000} = \frac{68576238333199}{9280784638125} \end{aligned}$$

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

$$\begin{aligned} \sum_{n=0}^0 \frac{(-1)^n}{n!} &= 1.00000000000000000000\dots = 1 = 1 \\ \sum_{n=0}^1 \frac{(-1)^n}{n!} &= 0\dots = 0 = 0 \\ \sum_{n=0}^2 \frac{(-1)^n}{n!} &= 0.50000000000000000000\dots = \frac{1}{2} = \frac{1}{2} \\ \sum_{n=0}^3 \frac{(-1)^n}{n!} &= 0.33333333333333333333\dots = \frac{2}{6} = \frac{1}{3} \\ \sum_{n=0}^4 \frac{(-1)^n}{n!} &= 0.37500000000000000000\dots = \frac{9}{24} = \frac{3}{8} \\ \sum_{n=0}^5 \frac{(-1)^n}{n!} &= 0.36666666666666666666\dots = \frac{44}{120} = \frac{11}{30} \\ \sum_{n=0}^6 \frac{(-1)^n}{n!} &= 0.36805555555555555555\dots = \frac{265}{720} = \frac{53}{144} \\ \sum_{n=0}^7 \frac{(-1)^n}{n!} &= 0.36785714285714285714\dots = \frac{1854}{5040} = \frac{103}{280} \\ \sum_{n=0}^8 \frac{(-1)^n}{n!} &= 0.36788194444444444444\dots = \frac{14833}{40320} = \frac{2119}{5760} \\ \sum_{n=0}^9 \frac{(-1)^n}{n!} &= 0.36787918871252204585\dots = \frac{133496}{362880} = \frac{16687}{45360} \\ \sum_{n=0}^{10} \frac{(-1)^n}{n!} &= 0.36787946428571428571\dots = \frac{1334961}{3628800} = \frac{16481}{44800} \\ \sum_{n=0}^{11} \frac{(-1)^n}{n!} &= 0.36787943923360590027\dots = \frac{14684570}{39916800} = \frac{1468457}{3991680} \\ \sum_{n=0}^{12} \frac{(-1)^n}{n!} &= 0.36787944132128159905\dots = \frac{176214841}{479001600} = \frac{16019531}{43545600} \\ \sum_{n=0}^{13} \frac{(-1)^n}{n!} &= 0.36787944116069116069\dots = \frac{2290792932}{6227020800} = \frac{63633137}{172972800} \\ \sum_{n=0}^{14} \frac{(-1)^n}{n!} &= 0.36787944117216190628\dots = \frac{32071101049}{87178291200} = \frac{2467007773}{6706022400} \end{aligned}$$

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

15.4 \xintRationalSeriesX

$\sum_{n=0}^{\infty} \frac{f^n}{f^n} f \star$

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

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

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

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

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

```
\def\firstterm #1{1[0]}% first term of the exponential series
% although it is the constant 1, here it must be defined as a
% one-parameter macro. Next comes the ratio function for exp:
\def\ratioexp #1#2{\xintDiv {#1}{#2}}% x/n
% These are the (-1)^(n-1)/n of the log(1+h) series:
\def\coefflog #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}%
% Let L(h) be the first 10 terms of the log(1+h) series and
% let E(t) be the first 10 terms of the exp(t) series.
% The following computes E(L(a/10)) for a=1,...,12.
\begin{multicols}{3}\raggedcolumns
\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
\end{multicols}
```

1.00000000000000000000...	1.499954310225476533...	1.907197560339468199...
1.0999999999999083906...	1.599659266069210466...	1.845117565491393752...
1.199999998111624029...	1.698137473697423757...	1.593831932293536053...
1.299999835744121464...	1.791898112718884531...	
1.399996091955359088...	1.870485649686617459...	

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]))=1635914436931178893034310888060876341482507357910234976572613140141591073957392
0639913787199465741057336677116573252341295218688/148719494266594663864467456000000000[-90]
(length of numerator: 127)
E(L(12[-2]))=166565833577572344676438956190268741913273209931571832475681257750593560183622
231934396045400537542264448715028348166448083362882112998458872460667950411608822312198051662
927273729660728412213074817261522841754729971712/148719494266594663864467456000000000[-180]
(length of numerator: 217)
E(L(123[-3]))=1670119920600550269986632390690022782662159669686695081451916268879387348622
732699865460786588039790140031169033780259351489004488146989366276335580667381519585306031672
406127856731756929927428636793983034074132050846923834747227198046227719821611171970458736202
25769049115687215712723182386527055033735053312/148719494266594663864467456000000000[-270] (length
of numerator: 307)
```

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))=482282038627508858480480322976551631937190834987521266229448636834789214635376522
0421966954177876452794933/4219967838162271878244372527760312278633063806332105808131741656092
50056936721328812056161288192000000000[0] (length of numerator: 105; length of denominator:
105)
E(L(1/71))=61900396707853503464065509951594765402729481828843984628828889229972383231974982
859719400152182490594357208368328392373910672874993163246058732446704305028542916962821162872
58603878135499973539887212860467/610406689757999825826438639900357618952467711000335704202712
671092203332984981842891074510835779826956944462566758343900417497150172256263898307611707752
791999877852355941834008347347315123552256000000000[0] (length of numerator: 203; length of
denominator: 203)
E(L(1/712))=3003564353778406020559670408411885925389093114199308387996560136260710297841742
496819290884958041362038132421744055614153154268292413172870530372734533290558141538915173252
756941123200263645694953665349180314390511046104875297961920582057259996416578066159049290482
98946463533146662233869249/299935178105220909766968489591763105361775507557039697364359215352
224604108923285325397380419112021214124247158817340492547166400824709873409851519325042814942
240645967888744414705331478482078635497788470006171032646666387826770190191301139308374215312
81047806202596610291401752576000000000[0] (length of numerator: 288; length of denominator:
288)
```

Thus decimal numbers such as `0.123` (equivalently `123[-3]`) give less computing intensive tasks than fractions such as `1/712`: in the case of decimal numbers the (raw) denominators originate in the coefficients of the series themselves, powers of ten of the input within brackets being treated separately. And even then the numerators will grow with the size of the input in a sort of linear way, the coefficient being given by the order of series: here 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!

Hence, truncating the output (or better, rounding) is the only way to go if one needs a general calculus of special functions. This is why the package xintseries provides, besides `\xintSeries`, `\xintRationalSeries`, or `\xintPowerSeries` which compute exact sums, `\xintFxFtPowerSeries` for fixed-point computations and a (tentative naive) `\xintFloatPowerSeries`.

15.5 `\xintPowerSeries`

`\xintPowerSeries{A}{B}{\coeff}{f}` evaluates the sum $\sum_{n=A}^{n=B} \text{\coeff}[n] \cdot f^n$. The initial and final indices are given to a `\numexpr` expression. The `\coeff` macro (which, as argument to `\xintPowerSeries`

`\num` `Frac` `Frac`
`x` `f` `f` ★

is expanded only at the time `\coeff{n}` is needed) should be defined as a one-parameter expandable macro, its input will be an explicit number.

The `f` can be either a fraction directly input or a macro `\f` expanding to such a fraction. It is actually more efficient to encapsulate an explicit fraction `f` in such a macro, if it has big numerators and denominators ('big' means hundreds of digits) as it will then take less space in the processing until being (repeatedly) used.

This macro computes the exact result (one can use it also for polynomial evaluation), using a Horner scheme which helps avoiding a denominator build-up (this problem however, even if using a naive additive approach, is much less acute since release 1.1 and its new policy regarding `\xint-Add`).

```
\def\geom #1{1[0]} % the geometric series
\def\f {5/17[0]}
\[ \sum_{n=0}^{20} \Bigl(\frac{5}{17}\Bigr)^n
=\xintFrac{\xintIrr{\xintPowerSeries {0}{20}{\geom}{\f}}}
=\xintFrac{\xinttheexpr (17^21-5^21)/12/17^20\relax\}
```

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

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

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

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

```
\setlength{\columnsep}{0pt}
\begin{multicols}{3}
\cnta 1 % previously declared count
\loop % in this loop we recompute from scratch each partial sum!
% we can afford that, as \xintPowerSeries is fast enough.
\noindent\hbox to 2em{\hfil\texttt{\the\cnta.} }%
\xintTrunc {12}
{\xintPowerSeries {1}{\cnta}{\coefflog}{\f}}\dots
\endgraf
\ifnum \cnta < 30 \advance\cnta 1 \repeat
\end{multicols}
```

1. 0.500000000000...	11. 0.693109245355...	21. 0.693147159757...
2. 0.625000000000...	12. 0.693129590407...	22. 0.693147170594...
3. 0.666666666666...	13. 0.693138980431...	23. 0.693147175777...
4. 0.682291666666...	14. 0.693143340085...	24. 0.693147178261...
5. 0.688541666666...	15. 0.693145374590...	25. 0.693147179453...
6. 0.691145833333...	16. 0.693146328265...	26. 0.693147180026...
7. 0.692261904761...	17. 0.693146777052...	27. 0.693147180302...
8. 0.692750186011...	18. 0.693146988980...	28. 0.693147180435...
9. 0.692967199900...	19. 0.693147089367...	29. 0.693147180499...
10. 0.693064856150...	20. 0.693147137051...	30. 0.693147180530...

```
\def\coeffarctg #1{/\the\numexpr\ifodd #1 -2*#1-1\else2*#1+1\fi\relax }%
```

```
% the above gives (-1)^(2n+1)/(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)
% Notice in passing this aspect of \numexpr:
%      **** \numexpr -(1)\relax is ilegal !!! ****
\def\frac{1/25[0]}{5} 1/5^2
\[\mathrm{Arctg}(\frac{1}{5})\approx \frac{1}{5}\sum_{n=0}^{15} \frac{(-1)^n}{(2n+1)25^n}
= \xintFrac{\xintIrr {\xintDiv {\xintPowerSeries {0}{15}{\coeffarctg}{f}{5}}\}}
```

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

15.6 \xintPowerSeriesX

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

```
\def\ratioexp #1#2{\xintDiv {#1}{#2}}% x/n
% These are the (-1)^(n-1)/n of the log(1+h) series:
\def\coefflog #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}%
% Let L(h) be the first 10 terms of the log(1+h) series and
% let E(t) be the first 10 terms of the exp(t) series.
% The following computes L(E(a/10)-1) for a=1,..., 12.
\begin{multicols}{3}\raggedcolumns
\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
\end{multicols}
```

0.09999999998556159...	0.499511320760604148...	-1.597091692317639401...
0.199999995263443554...	0.593980619762352217...	-12.648937932093322763...
0.299999338075041781...	0.645144282733914916...	-66.259639046914679687...
0.399974460740121112...	0.398118280111436442...	-304.768437445462801227...

15.7 \xintFxPtPowerSeries

`\xintFxPtPowerSeries{A}{B}{\coeff}{f}{D}` computes $\sum_{n=A}^{n=B} \text{coeff}\{n\} \cdot f^n$ with each term of the series truncated to `D` digits after the decimal point. As usual, `A` and `B` are completely expanded through their inclusion in a `\numexpr` expression. Regarding `D` it will be similarly be expanded each time it is used inside an `\xintTrunc`. The one-parameter macro `\coeff` is similarly expanded at the time it is used inside the computations. Idem for `f`. If `f` itself is some complicated macro it is thus better to use the variant `\xintFxPtPowerSeriesX` which expands it first and then uses the result of that expansion.

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

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

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

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

```
\def\coeffexp #1{\xintiiFac {#1}[0]}% 1/n!
\def\ f {-1/2[0]}% [0] for faster input parsing
\cnta 0 % previously declared \count register
\noindent\loop
$\xintFxFtPowerSeries {0}{\cnta}{\coeffexp}{\f}{20}$\
\ifnum\cnta<19 \advance\cnta 1 \repeat\par
\xintFxFtPowerSeries {0}{19}{\coeffexp}{\f}{25}= 0.6065306597126334236037992
```

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

$$\text{\xintPowerSeries {0}{19}{\coeffexp}{\f}} = \frac{38682746160036397317757}{63777066403145711616000}$$

$$= 0.606530659712633423603799152126\dots$$

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.

15.8 \xintFxFtPowerSeriesX

`\xintFxFtPowerSeriesX{A}{B}{\coeff}{\f}{D}` computes, exactly as `\xintFxFtPowerSeries`, the sum of $\text{\coeff{n}} \cdot f^n$ from $n=A$ to $n=B$ with each term of the series being *truncated* to D digits after the decimal point. The sole difference is that \f is first expanded and it is the result of this which is used in the computations.

Let us illustrate this on the numerical exploration of the identity $\log(1+x) = -\log(1/(1+x))$

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

```
\cnta 0
\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}% (-1)^{n-1}/n
\def\coeffalt #1{\the\numexpr\ifodd#1 -1\else1\fi\relax [0]}% (-1)^n
\begin{multicols}2
\loop
\noindent \hbox to 2.5cm {\hss\texttt{D(\the\cnta/100): }}%
\xintAdd {\xintFxFtPowerSeriesX {1}{10}{\coefflog}{\the\cnta [-2]}{5}}
{\xintFxFtPowerSeriesX {1}{10}{\coefflog}
```

```

        {\xintFxFtPowerSeriesX {1}{10}{\coeffalt}{\the\cnta [-2]}{5}}
        {5}}\endgraf
\ifnum\cnta < 49 \advance\cnta 7 \repeat
\end{multicols}

```

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.

```

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

```

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

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

15.9 \xintFloatPowerSeries

`\xintFloatPowerSeries[P]{A}{B}{\coeff}{f}` computes $\sum_{n=A}^{n=B} \text{coeff}\{n\} \cdot f^n$ with a floating point precision given by the optional parameter `P` or by the current setting of `\xintDigits`.

In the current, preliminary, version, no attempt has been made to try to guarantee to the final result the precision `P`. Rather, `P` is used for all intermediate floating point evaluations. So rounding errors will make some of the last printed digits invalid. The operations done are first the evaluation of f^A using `\xintFloatPow`, then each successive power is obtained from this first one by multiplication by `f` using `\xintFloatMul`, then again with `\xintFloatMul` this is multiplied with `\coeff{n}`, and the sum is done adding one term at a time with `\xintFloatAdd`. To sum up, this is just the naive transformation of `\xintFxFtPowerSeries` from fixed point to floating point.

```

\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}%
\xintFloatPowerSeries [8]{1}{30}{\coefflog}{-1/2[0]}
-6.9314718e-1

```

$\left[\frac{x}{f} \right]_f^x$ ★


```

}%
}%
\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 ≈ 0.693147180559945309417232121458176568075500134360255254120484...
≈ 0.69314718055994530941723212145817656807550013436025525412068000711...
≈ 0.6931471805599453094172321214581765680755001343602552541206800094933723...

```

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

```

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

```

\def\coeffarctg #1{\the\numexpr\ifodd#1 -1\else1\fi\relax/%
  \the\numexpr 2*#1+1\relax [0]}%
%\def\coeffarctg #1{\romannumeral0\xintmon{#1}/\the\numexpr 2*#1+1\relax }%
\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% #1 may be a count register, \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%
{\xinttrunc {#4}
 {\xintSub
  {\xintMul {16/5}{\xintFxFtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}

```

```

    {\xintMul{4/239}{\xintFxFtPowerSeries {0}{#2}{\coeffarctg}{\xb}{#3}}}%
  }}%
\begin{framed}
  \[ \pi = \Machin {60}\dots \]
\end{framed}

```

$$\pi = 3.141592653589793238462643383279502884197169399375105820974944\dots$$

Here is a variant `\MachinBis`, which evaluates the partial sums exactly using `\xintPowerSeries`, before their final truncation. No need for a ```+3'` then.

```

\def\MachinBis #1% #1 may be a count register,
% the final result will be truncated to #1 digits post decimal point
  \romannumeral0\expandafter\MachinBisA \expandafter
  % number of terms for arctg(1/5):
  {\the\numexpr #1*5/7\expandafter}\expandafter
  % number of terms for arctg(1/239):
  {\the\numexpr #1*10/45\expandafter}\expandafter
  % allow #1 to be a count register:
  {\the\numexpr #1\relax }}%
\def\MachinBisA #1#2#3%
{\xinttrunc {#3} %
{\xintSub
  {\xintMul {16/5}{\xintPowerSeries {0}{#1}{\coeffarctg}{\xa}}}
  {\xintMul{4/239}{\xintPowerSeries {0}{#2}{\coeffarctg}{\xb}}}%
}}%

```

Let us use this variant for a loop showing the build-up of digits:

```

\begin{multicols}{2}
  \cnta 0 % previously declared \count register
  \loop \noindent
    \centeredline{\dtt{\MachinBis{\cnta}}}%
  \ifnum\cnta < 30
  \advance\cnta 1 \repeat
\end{multicols}

```

	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? compile this copy of the `\Machin` with `etex` (or `pdftex`):

```

% Compile with e-TeX extensions enabled (etex, pdftex, ...)
\input xintfrac.sty
\input xintseries.sty
% pi = 16 Arctg(1/5) - 4 Arctg(1/239) (John Machin's formula)

```

```

\def\coeffarctg #1{\the\numexpr\ifodd#1 -1\else1\fi\relax/%
                    \the\numexpr 2*#1+1\relax [0]}%

\def\xa {1/25[0]}%
\def\xb {1/57121[0]}%
\def\Machin #1{%
  \romannumeral0\expandafter\MachinA \expandafter
  {\the\numexpr (#1+3)*5/7\expandafter}\expandafter
  {\the\numexpr (#1+3)*10/45\expandafter}\expandafter
  {\the\numexpr #1+3\expandafter}\expandafter
  {\the\numexpr #1\relax }}%
\def\MachinA #1#2#3#4%
{\xinttrunc {#4}
 {\xintSub
  {\xintMul {16/5}{\xintFxFtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}}
  {\xintMul {4/239}{\xintFxFtPowerSeries {0}{#2}{\coeffarctg}{\xb}{#3}}}%
 }}%
\pdfresettimer
\fdef\Z {\Machin {1000}}
\odef\W {\the\pdfelapsedtime}
\message{Z}
\message{computed in \xintRound {2}{\W/65536} seconds.}
\bye

```

This will log the first 1000 digits of π after the decimal point. On my laptop (a 2012 model) this took about 5.05 seconds last time I tried.^{79 80}

As mentioned in the introduction, the file `pi.tex` by D. ROEGEL shows that orders of magnitude faster computations are possible within \TeX , but recall our constraints of complete expandability and be merciful, please.

Why truncating rather than rounding? One of our main competitors on the market of scientific computing, a canadian product (not encumbered with expandability constraints, and having barely ever heard of \TeX ;-), prints numbers rounded in the last digit. Why didn't we follow suit in the macros `\xintFxFtPowerSeries` and `\xintFxFtPowerSeriesX`? To round at D digits, and excluding a rewrite or cloning of the division 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 negligible effect per summand compared to the additional cost for this term of having been truncated at $D+1$ then rounded). Rounding is the way to go when setting up algorithms to evaluate functions destined to be composed one after the other: exact algebraic operations with many summands and an f variable which is a fraction are costly and create an even bigger fraction; replacing f with a reasonable rounding, and rounding the result, is necessary to allow arbitrary chaining.

But, for the computation of a single constant, we are really interested in the exact decimal expansion, so we truncate and compute more terms until the earlier result gets validated. Finally if we do want the rounding we can always do it on a value computed with $D+1$ truncation.

⁷⁹ With 1.09i and earlier *xint*, this used to be 42 seconds; starting with 1.09j, and prior to 1.2, it was 16 seconds (this was probably due to a more efficient division with denominators at most 9999). The 1.2 *xintcore* achieves a further gain at 5.6 seconds. ⁸⁰ With `\xintDigits :=1001`, the non-optimized implementation with the *iter* of *xintexpr* fame using the Brent-Salamin algorithm, took, last time I tried (1.2i), about 7 seconds on my laptop (the last two digits were wrong, which is ok as they serve as guard digits), and for obtaining about 500 digits, it was about 1.7s. This is not bad, taking into account that the syntax is almost free rolling speech, contrarily to the code above for the Machin formula computation; we would like to use the quadratically convergent Brent-Salamin algorithm for more digits, but with such computations with numbers of one thousand digits we are beyond the border of the reasonable range for *xint*. Innocent people not knowing what it means to compute with \TeX , and with the extra constraint of expandability will wonder why this is at least thousands of times slower than with any other language (with a little Python program using the *Decimal* library, I timed the Brent-Salamin algorithm to 4.4ms for about 1000 digits and 1.14ms for 500 digits.) I will just say that for example digits are represented and manipulated via their ascii-code ! all computations must convert from ascii-code to cpu words; furthermore nothing can be stored away. And there is no memory storage with $O(1)$ time access... if expandability is to be verified.

16 Macros of the xintfrac package

<ul style="list-style-type: none"> .1 Package overview 166 .2 <code>\xintCFrac</code>..... 171 .3 <code>\xintGCFrac</code>..... 171 .4 <code>\xintGGCFrac</code> 172 .5 <code>\xintGctoGCx</code> 172 .6 <code>\xintFtoC</code>..... 172 .7 <code>\xintFtoCs</code>..... 172 .8 <code>\xintFtoCx</code>..... 172 .9 <code>\xintFtoGC</code>..... 173 .10 <code>\xintFGtoC</code>..... 173 .11 <code>\xintFtoCC</code>..... 173 .12 <code>\xintCstoF</code>..... 174 .13 <code>\xintCtoF</code>..... 174 .14 <code>\xintGctoF</code>..... 175 .15 <code>\xintCstoCv</code>..... 175 	<ul style="list-style-type: none"> .16 <code>\xintCtoCv</code>..... 176 .17 <code>\xintGCtoCv</code>..... 176 .18 <code>\xintFtoCv</code>..... 176 .19 <code>\xintFtoCCv</code>..... 176 .20 <code>\xintCntoF</code>..... 176 .21 <code>\xintGCntoF</code>..... 177 .22 <code>\xintCntoCs</code>..... 177 .23 <code>\xintCntoGC</code>..... 177 .24 <code>\xintGCntoGC</code> 178 .25 <code>\xintCstoGC</code>..... 178 .26 <code>\xintiCstoF</code>, <code>\xintiGctoF</code>, <code>\xintiCstoCv</code>, <code>\xintiGctoCv</code> 178 .27 <code>\xintGctoGC</code>..... 178 .28 Euler's number e..... 178
---	--

First version of this package was included in release 1.04 (2013/04/25) of the *xint* bundle. It was kept almost unchanged until 1.09m of 2014/02/26 which brought some new macros: `\xintFtoC`, `\xintCtoF`, `\xintCtoCv`, dealing with sequences of braced partial quotients rather than comma separated ones, `\xintFGtoC` which is to produce ``guaranteed'' coefficients of some real number known approximately, and `\xintGGCFrac` for displaying arbitrary material as a continued fraction; also, some changes to existing macros: `\xintFtoCs` and `\xintCntoCs` insert spaces after the commas, `\xintCstoF` and `\xintCstoCv` authorize spaces in the input also before the commas.

Note: `\xintCstoF` and `\xintCstoCv` create a partial dependency on *xinttools* (its `\xintCSVtoList`.) This section contains:

1. an [overview](#) of the package functionalities,
2. a description of each one of the package macros,
3. further illustration of their use via the study of the [convergents of e](#).

16.1 Package overview

The package computes partial quotients and convergents of a fraction, or conversely start from coefficients and obtain the corresponding fraction; three macros `\xintCFrac`, `\xintGCFrac` and `\xintGGCFrac` are for typesetting (the first two assume that the coefficients are numeric quantities acceptable by the *xintfrac* `\xintFrac` macro, the last one will display arbitrary material), the others can be nested (if applicable) or see their outputs further processed by other macros from the *xint* bundle, particularly the macros of *xinttools* dealing with sequences of braced items or comma separated lists.

A *simple* continued fraction has coefficients $[c_0, c_1, \dots, c_N]$ (usually called partial quotients, but I dislike this entrenched terminology), where c_0 is a positive or negative integer and the others are positive integers.

Typesetting is usually done via the *amsmath* macro `\cfrac`:

```
\ [ c_0 + \cfrac{1}{c_1+\cfrac{1}{c_2+\cfrac{1}{c_3+\cfrac{1}{\ddots}}}}\]
```

$$c_0 + \frac{1}{c_1 + \frac{1}{c_2 + \frac{1}{c_3 + \dots}}}$$

Here is a concrete example:

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

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

But it is the macro `\xintCFrac` which did all the work of *computing* the continued fraction and using `\cfrac` from `amsmath` to typeset it.

A *generalized* continued fraction has the same structure but the numerators are not restricted to be 1, and numbers used in the continued fraction may be arbitrary, also fractions, irrationals, complex, indeterminates.⁸¹ The *centered* continued fraction is an example:

```
\[ \xintFrac {915286/188421}=\xintGCFrac {5+-1/7+1/39+-1/53+-1/13}
=\xintCFrac {915286/188421}\]
```

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

The macro `\xintGCFrac`, contrarily to `\xintCFrac`, does not compute anything, it just typesets starting from a generalized continued fraction in inline format, which in this example was input literally. We also used `\xintCFrac` for comparison of the two types of continued fractions.

To let \TeX compute the centered continued fraction of f there is `\xintFtoCC`:

```
\[ \xintFrac {915286/188421}\to\xintFtoCC {915286/188421}\]
```

$$\frac{915286}{188421} \rightarrow 5 + -1/7 + 1/39 + -1/53 + -1/13$$

The package macros are expandable and may be nested (naturally `\xintCFrac` and `\xintGCFrac` must be at the top level, as they deal with typesetting).

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

$$5 - \frac{1}{7 + \frac{1}{39 - \frac{1}{53 - \frac{1}{13}}}}$$

⁸¹ `xintcfrac` may be used with indeterminates, for basic conversions from one inline format to another, but not for actual computations. See `\xintGGCFrac`.

The `'inline'` format expected on input by `\xintGCFrac` is

$$a_0 + b_0/a_1 + b_1/a_2 + b_2/a_3 + \dots + b_{n-2}/a_{n-1} + b_{n-1}/a_n$$

Fractions among the coefficients are allowed but they must be enclosed within braces. Signed integers may be left without braces (but the `+` signs are mandatory). No spaces are allowed around the plus and fraction symbols. The coefficients may themselves be macros, as long as these macros are *f-expandable*.

```
\[ \xintFrac{\xintGCtoF {1+-1/57+\xintPow {-3}{7}/\xintiiQuo {132}{25}}}
= \xintGCFrac {1+-1/57+\xintPow {-3}{7}/\xintiiQuo {132}{25}}\]
```

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

To compute the actual fraction one has `\xintGCtoF`:

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

$$\frac{1907}{1902}$$

For non-numeric input there is `\xintGGCFrac`.

```
\[\xintGGCFrac {a_0+b_0/a_1+b_1/a_2+b_2/\ddots+\ddots/a_{n-1}+b_{n-1}/a_n}\]
```

$$a_0 + \frac{b_0}{a_1 + \frac{b_1}{a_2 + \frac{b_2}{\dots + \frac{b_{n-1}}{a_{n-1} + \frac{b_{n-1}}{a_n}}}}}$$

For regular continued fractions, there is a simpler comma separated format:

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

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

The macro `\xintFtoCs` produces from a fraction *f* the comma separated list of its coefficients.

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

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

If one prefers other separators, one can use the two arguments macros `\xintFtoCx` whose first argument is the separator (which may consist of more than one token) which is to be used.

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

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

This allows under Plain TeX with `amstex` to obtain the same effect as with \LaTeX +`\amsmath`+`\xintCFrac`:

```
$$\xintFwOver{2721/1001}=\xintFtoCx {+\cfrac1{\}}{2721/1001}\endcfrac$$
```

As a shortcut to `\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` To obtain the coefficients as a sequence of braced numbers, there is `\xintFtoC` (this is a shortcut for `\xintFtoCx {}`). This list (sequence) may then be manipulated using the various macros of `xinttools` such as the non-expandable macro `\xintAssignArray` or the expandable `\xintApply` and `\xintListWithSep`.

Conversely to go from such a sequence of braced coefficients to the corresponding fraction there is `\xintCtoF`.

The `\printnumber` (subsection 1.3) macro which we use in this document to print long numbers can also be useful on long continued fractions.

```
\printnumber{\xintFtoCC {35037018906350720204351049/244241737886197404558180}}
143+1/2+1/5+-1/4+-1/4+-1/4+-1/3+1/2+1/2+1/6+-1/22+1/2+1/10+-1/5+-1/11+-1/3+1/4+-1/2+1/2+1/4+
-1/2+1/23+1/3+1/8+-1/6+-1/9
```

If we apply `\xintGtoF` to this generalized continued fraction, we discover that the original fraction was reducible:

```
\xintGtoF {143+1/2+...+-1/9}=2897319801297630107/20197107104701740
```

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

```
\xintGtoF {143+1/2+...+-1/6}=328124887710626729/2287346221788023
```

and indeed:

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

The various fractions obtained from the truncation of a continued fraction to its initial terms are called the convergents. The macros of xintfrac such as `\xintFtoCv`, `\xintFtoCCv`, and others which compute such convergents, return them as a list of braced items, with no separator (as does `\xintFtoC` for the partial quotients). Here is an example:

```
\[\xintFrac{915286/188421}\to
\xintListWithSep{,}{\xintApply\xintFrac{\xintFtoCv{915286/188421}}\}
\]

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

```

```
\[\xintFrac{915286/188421}\to
\xintListWithSep{,}{\xintApply\xintFrac{\xintFtoCCv{915286/188421}}\}
\]

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

```

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

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

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

Next, we use the following code:

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

It produces:

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

The macro `\xintCtoF` allows to specify the coefficients as a function given by a one-parameter macro. The produced values do not have to be integers.

```
\def\cn #1{\xintiiPow {2}{#1}}% 2^n
\[\xintFrac{\xintCtoF {6}{\cn}}=\xintCFrac [1]{\xintCtoF {6}{\cn}}\]
```


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

Notice the use of the optional argument [1] to `\xintCFrac`. Other possibilities are [r] and (default) [c].

```
\def\cn #1{\xintPow {2}{-#1}}%
\[\xintFrac{\xintCntoF {6}{\cn}}=\xintGCFrac [r]{\xintCntoGC {6}{\cn}}=
\xintFtoCs {\xintCntoF {6}{\cn}}\]
\frac{3159019}{2465449} = 1 + \frac{1}{\frac{1}{2} + \frac{1}{\frac{1}{4} + \frac{1}{\frac{1}{8} + \frac{1}{\frac{1}{16} + \frac{1}{\frac{1}{32} + \frac{1}{64}}}}}} = [1, 3, 1, 1, 4, 14, 1, 1, 1, 1, 79, 2, 1, 1, 2]
```

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

There are also `\xintGCntoF` and `\xintGCntoGC` which allow the same for generalized fractions. An initial portion of a generalized continued fraction for π is obtained like this

```
\def\an #1{\the\numexpr 2*#1+1\relax }%
\def\bn #1{\the\numexpr (#1+1)*(#1+1)\relax }%
\[\xintFrac{\xintDiv {4}{\xintGCntoF {5}{\an}{\bn}}} =
\cfrac{4}{\xintGCFrac{\xintCntoGC {5}{\an}{\bn}}} =
\xintTrunc {10}{\xintDiv {4}{\xintGCntoF {5}{\an}{\bn}}\dots\]
\frac{92736}{29520} = \frac{4}{1 + \frac{1}{3 + \frac{4}{5 + \frac{9}{7 + \frac{16}{9 + \frac{25}{11}}}}}} = 3.1414634146\dots
```

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

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

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

When studying the continued fraction of some real number, there is always some doubt about how many terms are valid, when computed starting from some approximation. If $f \leq x \leq g$ and f, g both have the same first K partial quotients, then x also has the same first K quotients and convergents. The macro `\xintFGtoC` outputs as a sequence of braced items the common partial quotients of its two arguments. We can thus use it to produce a sure list of valid convergents of π for example, starting from some proven lower and upper bound:

```

 $\pi$ \to [\xintListWithSep{,}
  {\xintFGtoC {3.14159265358979323}{3.14159265358979324}}, \dots]
\noindent $\pi$ \to\xintListWithSep{,\allowbreak\;}
  {\xintApply{\xintFrac}
  {\xintCtoCv{\xintFGtoC {3.14159265358979323}{3.14159265358979324}}}}, \dots
  \pi \to [3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, \dots]
 $\pi \to 3, \frac{22}{7}, \frac{333}{106}, \frac{355}{113}, \frac{103993}{33102}, \frac{104348}{33215}, \frac{208341}{66317}, \frac{312689}{99532}, \frac{833719}{265381}, \frac{1146408}{364913}, \frac{4272943}{1360120}, \frac{5419351}{1725033}, \frac{80143857}{25510582},$ 
 $\frac{165707065}{52746197}, \frac{245850922}{78256779}, \frac{411557987}{131002976}, \dots$ 

```

16.2 \xintCFrac

Frac
f

`\xintCFrac{f}` is a math-mode only, \TeX with `amsmath` only, macro which first computes then displays with the help of `\cfrac` the simple continued fraction corresponding to the given fraction. It admits an optional argument which may be `[l]`, `[r]` or (the default) `[c]` to specify the location of the one's in the numerators of the sub-fractions. Each coefficient is typeset using the `\xintFrac` macro from the `xintfrac` package. This macro is *f-expandable* in the sense that it prepares expandably the whole expression with the multiple `\cfrac`'s, but it is not completely expandable naturally as `\cfrac` isn't.

16.3 \xintGCFrac

f

`\xintGCFrac{a+b/c+d/e+f/g+h/...+x/y}` uses similarly `\cfrac` to prepare the typesetting with the `amsmath` `\cfrac` (\TeX) of a generalized continued fraction given in inline format (or as macro which will *f-expand* to it). It admits the same optional argument as `\xintCFrac`. Plain \TeX with `amstex` users, see `\xintGctoGCx`.

```

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

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


```

This is mostly a typesetting macro, although it does provoke the expansion of the coefficients. See `\xintGctoF` if you are impatient to see this specific fraction computed.

It admits an optional argument within square brackets which may be either `[l]`, `[c]` or `[r]`. Default is `[c]` (numerators are centered).

Numerators and denominators are made arguments to the `\xintFrac` macro. This allows them to be themselves fractions or anything *f-expandable* giving numbers or fractions, but also means however that they can not be arbitrary material, they can not contain color changing macros for example. One of the reasons is that `\xintGCFrac` tries to determine the signs of the numerators and chooses accordingly to use `+` or `-`.

16.4 `\xintGGCFrac`

f `\xintGGCFrac{a+b/c+d/e+f/g+h/...+x/y}` is a clone of `\xintGCFrac`, hence again \TeX specific with package `amsmath`. It does not assume the coefficients to be numbers as understood by `xintfrac`. The macro can be used for displaying arbitrary content as a continued fraction with `\cfrac`, using only plus signs though. Note though that it will first *f-expand* its argument, which may be thus be one of the xintfrac macros producing a (general) continued fraction in inline format, see `\xintFtoCx` for an example. If this expansion is not wished, it is enough to start the argument with a space.

```
\[\xintGGCFrac {1+q/1+q^2/1+q^3/1+q^4/1+q^5/\ddots}\]
```

$$1 + \frac{q}{1 + \frac{q^2}{1 + \frac{q^3}{1 + \frac{q^4}{1 + \frac{q^5}{\ddots}}}}}$$

16.5 `\xintGtoGCx`

nnf ★ `\xintGtoGCx{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

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

The following can be used by Plain \TeX +`amstex` users to obtain an output similar as the ones produced by `\xintGCFrac` and `\xintGGCFrac`:

```
$$\xintGtoGCx {+\cfrac{\{\}\{a+b/\dots\}\endcfrac}{}
$\xintGtoGCx {+\cfrac\xintFwOver{\{\}\xintFwOver}{a+b/\dots}\endcfrac}{}$
```

16.6 `\xintFtoC`

Frac f ★ `\xintFtoC{f}` computes the coefficients of the simple continued fraction of *f* and returns them as a list (sequence) of braced items.

```
\fdef\test{\xintFtoC{-5262046/89233}}\texttt{\meaning\test}
```

macro: ->{-59}{33}{27}{100}

16.7 `\xintFtoCs`

Frac f ★ `\xintFtoCs{f}` returns the comma separated list of the coefficients of the simple continued fraction of *f*. Notice that starting with 1.09m a space follows each comma (mainly for usage in text mode, as in math mode spaces are produced in the typeset output by \TeX itself).

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

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

16.8 `\xintFtoCx`

Frac n f ★ `\xintFtoCx{sep}{f}` returns the list of the coefficients of the simple continued fraction of *f* separated with the help of *sep*, which may be anything (and is kept unexpanded). For example, with Plain \TeX and `amstex`,

```
$$\xintFtoCx {+\cfrac1\{\}\{-5262046/89233\}\endcfrac}{}$
```

will display the continued fraction using `\cfrac`. Each coefficient is inside a brace pair `{ }`,

allowing a macro to end the separator and fetch it as argument, for example, again with Plain TeX and *amstex*:

```
\def\highlight #1{\ifnum #1>200 \textcolor{red}{#1}\else #1\fi}
\$\xintFtoCx {+\cfrac1{\ \highlight}{104348/33215}\endcfrac$}
```

Due to the different and extremely cumbersome syntax of `\cfrac` under TeX it proves a bit tortuous to obtain there the same effect. Actually, it is partly for this purpose that 1.09m added `\xintGGCFrac`. We thus use `\xintFtoCx` with a suitable separator, and then the whole thing as argument to `\xintGGCFrac`:

```
\def\highlight #1{\ifnum #1>200 \fcolorbox{blue}{white}{\boldmath\color{red}$#1$}%
\else #1\fi}
\[\xintGGCFrac {\xintFtoCx {+1/\highlight}{208341/66317}}\]
```

$$3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1 + \frac{1}{\boxed{292} + \frac{1}{2}}}}}$$

16.9 `\xintFtoGC`

$\frac{f}{f}$ ★ `\xintFtoGC{f}` does the same as `\xintFtoCx{+1/}{f}`. Its output may thus be used in the package macros expecting such an 'inline format'.

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

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

16.10 `\xintFGtoC`

$\frac{f}{f}$ $\frac{f}{f}$ ★ `\xintFGtoC{f}{g}` computes the common initial coefficients to two given fractions *f* and *g*. Notice that any real number $f < x < g$ or $f > x > g$ will then necessarily share with *f* and *g* these common initial coefficients for its regular continued fraction. The coefficients are output as a sequence of braced numbers. This list can then be manipulated via macros from *xinttools*, or other macros of xintcfrac.

```
\fdef\test{\xintFGtoC{-5262046/89233}{-5314647/90125}}\texttt{\meaning\test}
```

```
macro:->{-59}{33}{27}
```

```
\fdef\test{\xintFGtoC{3.141592653}{3.141592654}}\texttt{\meaning\test}
```

```
macro:->{3}{7}{15}{1}
```

```
\fdef\test{\xintFGtoC{3.1415926535897932384}{3.1415926535897932385}}\meaning\test
```

```
macro:->{3}{7}{15}{1}{292}{1}{1}{1}{2}{1}{3}{1}{14}{2}{1}{1}{2}{2}{2}
```

```
\xintRound {30}{\xintCstoF{\xintListWithSep{,}{\test}}}
```

```
3.141592653589793238386377506390
```

```
\xintRound {30}{\xintCtoF{\test}}
```

```
3.141592653589793238386377506390
```

```
\fdef\test{\xintFGtoC{1.41421356237309}{1.4142135623731}}\meaning\test
```

```
macro:->{1}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}
```

16.11 `\xintFtoCC`

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

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

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

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

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

16.12 `\xintCstoF`

f ★ `\xintCstoF{a,b,c,d,...,z}` computes the fraction corresponding to the coefficients, which may be fractions or even macros expanding to such fractions. The final fraction may then be highly reducible.



Usage of this macro requires the user to load `xinttools`.

Starting with release 1.09m spaces before commas are allowed and trimmed automatically (spaces after commas were already silently handled in earlier releases).

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

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

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

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

A generalized continued fraction may produce a reducible fraction (`\xintCstoF` tries its best not to accumulate in a silly way superfluous factors but will not do simplifications which would be obvious to a human, like simplification by 3 in the result above).

16.13 `\xintCtoF`

f ★ `\xintCtoF{{a}{b}{c}...{z}}` computes the fraction corresponding to the coefficients, which may be fractions or even macros.

```
\xintCtoF {\xintApply {\xintiiPow 3}{\xintSeq {1}{5}}}
14946960/4805083
\[\ \xintFrac{14946960/4805083}=\xintCFrac {14946960/4805083}\]
```

$$\frac{14946960}{4805083} = 3 + \frac{1}{9 + \frac{1}{27 + \frac{1}{81 + \frac{1}{243}}}}$$

In the example above the power of 3 was already pre-computed via the expansion done by `\xintApply`, but if we try with `\xintApply { \xintiiPow 3}` where the space will stop this expansion, we can check that `\xintCtoF` will itself provoke the needed coefficient expansion.

16.14 `\xintGctoF`

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

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

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

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

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

The macro tries its best not to accumulate superfluous factor in the denominators, but doesn't reduce the fraction to irreducible form before returning it and does not do simplifications which would be obvious to a human.

16.15 `\xintCstoCv`

f ★ `\xintCstoCv{a,b,c,d,...,z}` returns the sequence of the corresponding convergents, each one within braces.



Usage of this macro requires the user to load `xinttools`.

It is allowed to use fractions as coefficients (the computed convergents have then no reason to be the real convergents of the final fraction). When the coefficients are integers, the convergents are irreducible fractions, but otherwise it is not necessarily the case.

```
\xintListWithSep:{\xintCstoCv{1,2,3,4,5,6}}
1/1:3/2:10/7:43/30:225/157:1393/972
\xintListWithSep:{\xintCstoCv{1,1/2,1/3,1/4,1/5,1/6}}
1/1:3/1:9/7:45/19:225/159:1575/729
```

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

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

16.16 `\xintCtoCv`

f ★ `\xintCtoCv{a}{b}{c}...{z}` returns the sequence of the corresponding convergents, each one within braces.

```
\fdef\test{\xintCtoCv {11111111111}\texttt{\meaning\test}}
```

```
macro:->{1/1}{2/1}{3/2}{5/3}{8/5}{13/8}{21/13}{34/21}{55/34}{89/55}{144/89}
```

16.17 `\xintGCtoCv`

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

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

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

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

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

16.18 `\xintFtoCv`

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

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

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

16.19 `\xintFtoCCv`

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

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

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

16.20 `\xintCntoF`

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

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

```
72625/49902[0]
```

This example shows that the fraction is output with a trailing number in square brackets (representing a power of ten), this is for consistency with what do most macros of `xintfrac`, and does not have to be always this annoying [0] as the coefficients may for example be numbers in scientific notation. To avoid these trailing square brackets, for example if the coefficients are known to be integers, there is always the possibility to filter the output via `\xintPRaw`, or `\xintIrr` (the latter is overkill in the case of integer coefficients, as the fraction is guaranteed to be irreducible then).

16.21 \xintGCntoF

$\overset{\text{num}}{x} f f \star$ `\xintGCntoF{N}{\macroA}{\macroB}` returns the fraction *f* corresponding to the inline generalized continued fraction $a_0 + b_0/a_1 + b_1/a_2 + \dots + b_{(N-1)}/a_N$, with $a(j) = \macroA{j}$ and $b(j) = \macroB{j}$. The *N* parameter is given to a `\numexpr`.

```
\def\coeffA #1{\the\numexpr #1+4-3*((#1+2)/3)\relax }%
\def\coeffB #1{\the\numexpr \ifodd #1 -\fi 1\relax }% (-1)^n
\[\xintGCFrac{\xintGCntoGC {6}{\coeffA}{\coeffB}} =
\xintFrac{\xintGCntoF {6}{\coeffA}{\coeffB}}\]
```

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

There is also `\xintGCntoGC` to get the 'inline format' continued fraction.

16.22 \xintCntoCs

$\overset{\text{num}}{x} f \star$ `\xintCntoCs{N}{\macro}` produces the comma separated list of the corresponding coefficients, from $n=0$ to $n=N$. The *N* is given to a `\numexpr`.

```
\xintCntoCs {5}{\macro}
1, 2, 5, 10, 17, 26
\[\xintFrac{\xintCntoF {5}{\macro}} = \xintCFrac{\xintCntoF {5}{\macro}}\]
```

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

16.23 \xintCntoGC

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

```
\def\macro #1{\the\numexpr \ifodd #1 -1-#1 \else 1+#1 \fi \relax / \the\numexpr 1+#1*#1 \relax }
\edef\x{\xintCntoGC {5}{\macro}} \meaning\x
\[\xintGCFrac{\xintCntoGC {5}{\macro}}\]
macro:->\{1/\the \numexpr 1+0*0\relax }+1/{-2/\the \numexpr 1+1*1\relax }+1/{3/\the \numexpr
1+2*2\relax }+1/{-4/\the \numexpr 1+3*3\relax }+1/{5/\the \numexpr 1+4*4\relax }+1/{-6/\the
\numexpr 1+5*5\relax }
```

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

16.24 \xintGCntoGC

$\overset{\text{num}}{X}$ *ff* ★ `\xintGCntoGC{N}{\macroA}{\macroB}` evaluates the coefficients and then returns the corresponding $\{a_0\}+\{b_0\}/\{a_1\}+\{b_1\}/\{a_2\}+\dots+\{b_{(N-1)}\}/\{a_N\}$ inline generalized fraction. *N* is given to a `\numex` *xpr*. The coefficients are enclosed into pairs of braces, and may thus be fractions, the fraction slash will not be confused in further processing by the continued fraction slashes.

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

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

16.25 \xintCstoGC

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

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

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

16.26 \xintiCstoF, \xintiGctoF, \xintiCstoCv, \xintiGctoCv

f ★ Essentially the same as the corresponding macros without the 'i', but for integer-only input. Infinitesimally faster, mainly for internal use by the package.

16.27 \xintGctoGC

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

```
\fdef\x { \xintGctoGC {1+\xintPow{1.5}{3}/{1/7}+{-3/5}}%
\xintiiFac {6}+\xintCstoF {2,-7,-5}/16}} \meaning\x
```

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

To be honest I have forgotten for which purpose I wrote this macro in the first place.

16.28 Euler's number e

Let us explore the convergents of Euler's number *e*. The volume of computation is kept minimal by the following steps:

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

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.

```

\def\z {\xintCtoF {199}{\cn}}%
\begingroup\parindent 0pt \leftskip 2.5cm
\indent\llap {Numerator = }\printnumber{\xintNumerator\z}\par
\indent\llap {Denominator = }\printnumber{\xintDenominator\z}\par
\indent\llap {Expansion = }\printnumber{\xintTrunc{268}\z}\dots\par\endgroup
Numerator = 568964038871896267597523892315807875293889017667917446057232024547192296961118
23017524386017499531081773136701241708609749634329382906
Denominator = 331123817669737619306256360816356753365468823729314438156205615463246659728581
86546133769206314891601955061457059255337661142645217223
Expansion = 1.7182818284590452353602874713526624977572470936999595749669676277240766303535
475945713821785251664274274663919320030599218174135966290435729003342952605956
307381323286279434907632338298807531952510190115738341879307021540891499348841
675092447614606680822648001684774118...

```

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.

17 Macros of the [xinttools](#) package

.1	<code>\xintRevWithBraces</code>	181			<code>\xintloopindex</code> , <code>\xintbreakiloop</code> , <code>\xint-</code>
.2	<code>\xintZapFirstSpaces</code> , <code>\xintZapLas-</code>				<code>breakiloopanddo</code> , <code>\xintloopskiptonext</code> ,
	<code>tSpaces</code> , <code>\xintZapSpaces</code> , <code>\xintZapSpacesB</code>	181			<code>\xintloopskipandredo</code>
.3	<code>\xintCSVtoList</code>	182	.15	<code>\xintApplyInline</code>	192
.4	<code>\xintNthElt</code>	183	.16	<code>\xintFor</code> , <code>\xintFor*</code>	193
.5	<code>\xintKeep</code>	184	.17	<code>\xintifForFirst</code> , <code>\xintifForLast</code>	196
.6	<code>\xintKeepUnbraced</code>	184	.18	<code>\xintBreakFor</code> , <code>\xintBreakForAndDo</code>	196
.7	<code>\xintTrim</code>	185	.19	<code>\xintintegers</code> , <code>\xintdimensions</code> , <code>\xin-</code>	
.8	<code>\xintTrimUnbraced</code>	185		<code>rationals</code>	196
.9	<code>\xintListWithSep</code>	186	.20	<code>\xintForpair</code> , <code>\xintForthree</code> , <code>\xintFor-</code>	
.10	<code>\xintApply</code>	186		<code>four</code>	198
.11	<code>\xintApplyUnbraced</code>	186	.21	<code>\xintAssign</code>	198
.12	<code>\xintSeq</code>	187	.22	<code>\xintAssignArray</code>	199
.13	<code>\xintloop</code> , <code>\xintbreakloop</code> , <code>\xintbreak-</code>		.23	<code>\xintDigitsOf</code>	199
	<code>loopanddo</code> , <code>\xintloopskiptonext</code>	187	.24	<code>\xintRelaxArray</code>	199
.14	<code>\xintiloop</code> , <code>\xintiloopindex</code> , <code>\xintouter-</code>				

These utilities used to be provided within the [xint](#) package; since 1.09g (2013/11/22) they have been moved to an independently usable package [xinttools](#), which has none of the [xint](#) facilities regarding big numbers. Whenever relevant release 1.09h has made the macros `\long` so they accept `\par` tokens on input.

The completely expandable utilities (up to `\xintloop`) are documented first, then the non-expandable utilities.

A brief overview is in [section 4](#) and [section 5](#) has more examples of use of macros of this package.

17.1 `\xintRevWithBraces`

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

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

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

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

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

17.2 `\xintZapFirstSpaces`, `\xintZapLastSpaces`, `\xintZapSpaces`, `\xintZapSpacesB`

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

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

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

It expands in two steps, and if the goal is to apply it to the expansion text of `\x` to define `\y`, then one can do: `\odef\y{\romannumeral0\expandafter\xintzapfirstspaces\expandafter{\x}}` (one can also define a wrapper macro to `\xintZapFirstSpaces` in order to expand once the argument first, but [xinttools](#) not being a programming layer, it provides no «Generate Variants» facilities).

Other use case: inside a macro which received a parameter `#1`, one can do `\oodef\x{\xintZapFirstSpaces {#1}}`, or, if `#1`, after leading spaces have been stripped can accept `\edef` expansion, one can do `\edef\x{\xintZapFirstSpaces{#1}}`.

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

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

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

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

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

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

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

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

17.3 `\xintCSVtoList`

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

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

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

Contiguous spaces and tab characters, are collapsed by TeX into single spaces. All such spaces

around commas⁸² are removed, as well as the spaces at the start and the spaces at the end of the list.⁸³ The items may contain explicit `\par`'s or empty lines (converted by the \TeX input parsing into `\par` tokens).

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

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

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

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

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

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

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

Else, we may have direct use:

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

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

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

17.4 `\xintNthElt`

$\overset{\text{num}}{x}$ `\xintNthElt{x}{<list>}` gets (expandably) the x th item of the `<list>`. A braced item will lose one level of brace pairs. The token list is first *f-expanded*.

⁸² and multiple space tokens are not a problem; but those at the top level (not hidden inside braces) *must* be of character code 32. ⁸³ let us recall that this is all done completely expandably... There is absolutely no alteration of any sort of the item apart from the stripping of initial and final space tokens (of character code 32) and brace removal if and only if the item apart from initial and final spaces (or more generally multiple `char 32` space tokens) is braced.

Items are counted starting at one.

```
\xintNthElt {3}{\agh}\u{zzz}\v{Z}} is zzz
\xintNthElt {3}{\agh}\u{zzz}\v{Z}} is {zzz}
\xintNthElt {2}{\agh}\u{zzz}\v{Z}} is \u
\xintNthElt {37}{\xintiiFac {100}}=9 is the thirty-seventh digit of 100!.
\xintNthElt {10}{\xintFtoCv {566827/208524}}=1457/536
is the tenth convergent of 566827/208524 (uses xintcfrac package).
\xintNthElt {7}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=7
\xintNthElt {0}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=9
\xintNthElt {-3}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=7
```

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

If $x<0$, the macro returns the $|x|$ th element from the end of the list. Thus for example $x=-1$ will fetch the last item of the list.

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

^{num}
 x n ★

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

If x is strictly larger (in absolute value) than the length of the list then `\xintNthElt` produces empty contents.

17.5 `\xintKeep`

^{num}
 x f ★

`\xintKeep{x}{\langle list \rangle}` expands the token list argument L and produces a new list, depending on the value of x :

- if $x>0$, the new list contains the first x items from L (counting starts at one.) *Each such item will be output within a brace pair.* Use `\xintKeepUnbraced` if this is not desired. This means that if the list item was braced to start with, there is no modification, but if it was a token without braces, then it acquires them.
- if $x>=\text{length}(L)$, the new list is the old one with all its items now braced.
- if $x=0$ the empty list is returned.
- if $x<0$ the last $|x|$ elements compose the output in the same order as in the initial list; as the macro proceeds by removing head items the kept items end up in output as they were in input: no added braces.
- if $x=-\text{length}(L)$ the output is identical with the input.

`\xintKeepNoExpand` does the same without first *f-expanding* its list argument.

```
\fdef\test {\xintKeep {17}{\xintKeep {-69}{\xintSeq {1}{100}}}\meaning\test\par
\noindent\fdef\test {\xintKeep {7}{\{1\}\{2\}\{3\}\{4\}\{5\}\{6\}\{7\}\{8\}\{9\}}\meaning\test\par
\noindent\fdef\test {\xintKeep {-7}{\{1\}\{2\}\{3\}\{4\}\{5\}\{6\}\{7\}\{8\}\{9\}}\meaning\test\par
\noindent\fdef\test {\xintKeep {7}{123456789}\meaning\test\par
\noindent\fdef\test {\xintKeep {-7}{123456789}\meaning\test\par
```

macro: ->{32}{33}{34}{35}{36}{37}{38}{39}{40}{41}{42}{43}{44}{45}{46}{47}{48}

macro: ->{1}{2}{3}{4}{5}{6}{7}

macro: ->{3}{4}{5}{6}{7}{8}{9}

macro: ->{1}{2}{3}{4}{5}{6}{7}

macro: ->3456789

17.6 `\xintKeepUnbraced`

Same as `\xintKeep` but no brace pairs are added around the kept items from the head of the list in the case $x>0$: each such item will lose one level of braces. Thus, to remove braces from all items of the list, one can use `\xintKeepUnbraced` with its first argument larger than the length of the list;

the same is obtained from `\xintListWithSep`{<list>}. But the new list will then have generally many more items than the original ones, corresponding to the unbraced original items.

For $x < 0$ the macro is no different from `\xintKeep`. Hence the name is a bit misleading because brace removal will happen only if $x > 0$.

`\xintKeepUnbracedNoExpand` does the same without first *f-expanding* its list argument.

```
\fdef\test {\xintKeepUnbraced {10}{\xintSeq {1}{100}}}\meaning\test\par
\noindent\fdef\test {\xintKeepUnbraced {7}{\{1\}{2\}{3\}{4\}{5\}{6\}{7\}{8\}{9\}}}\meaning\test\par
\noindent\fdef\test {\xintKeepUnbraced {-7}{\{1\}{2\}{3\}{4\}{5\}{6\}{7\}{8\}{9\}}}\meaning\test\par
\noindent\fdef\test {\xintKeepUnbraced {7}{123456789}}\meaning\test\par
\noindent\fdef\test {\xintKeepUnbraced {-7}{123456789}}\meaning\test\par
```

```
macro:->12345678910
macro:->1234567
macro:->{3}{4}{5}{6}{7}{8}{9}
macro:->1234567
macro:->3456789
```

17.7 `\xintTrim`

^{num}_x f ★ `\xintTrim`{x}{<list>} expands the list argument and gobbles its first x elements.

- if $x > 0$, the first x items from L are gobbled. The remaining items are not modified.
- if $x = \text{length}(L)$, the returned list is empty.
- if $x = 0$ the original list is returned (with no added braces.)
- if $x < 0$ the last $|x|$ items of the list are removed. *The head items end up braced in the output.* Use `\xintTrimUnbraced` if this is not desired.
- if $x \leq -\text{length}(L)$ the output is empty.

`\xintTrimNoExpand` does the same without first *f-expanding* its list argument.

```
\fdef\test {\xintTrim {17}{\xintTrim {-69}{\xintSeq {1}{100}}}\meaning\test\par
\noindent\fdef\test {\xintTrim {7}{\{1\}{2\}{3\}{4\}{5\}{6\}{7\}{8\}{9\}}}\meaning\test\par
\noindent\fdef\test {\xintTrim {-7}{\{1\}{2\}{3\}{4\}{5\}{6\}{7\}{8\}{9\}}}\meaning\test\par
\noindent\fdef\test {\xintTrim {7}{123456789}}\meaning\test\par
\noindent\fdef\test {\xintTrim {-7}{123456789}}\meaning\test\par
```

```
macro:->{18}{19}{20}{21}{22}{23}{24}{25}{26}{27}{28}{29}{30}{31}
macro:->{8}{9}
macro:->{1}{2}
macro:->89
macro:->{1}{2}
```

17.8 `\xintTrimUnbraced`

Same as `\xintTrim` but in case of a negative x (cutting items from the tail), the kept items from the head are not enclosed in brace pairs. They will lose one level of braces. The name is a bit misleading because when $x > 0$ there is no brace-stripping done on the kept items, because the macro works simply by gobbling the head ones.

`\xintTrimUnbracedNoExpand` does the same without first *f-expanding* its list argument.

```
\fdef\test {\xintTrimUnbraced {-90}{\xintSeq {1}{100}}}\meaning\test\par
\noindent\fdef\test {\xintTrimUnbraced {7}{\{1\}{2\}{3\}{4\}{5\}{6\}{7\}{8\}{9\}}}\meaning\test\par
\noindent\fdef\test {\xintTrimUnbraced {-7}{\{1\}{2\}{3\}{4\}{5\}{6\}{7\}{8\}{9\}}}\meaning\test\par
\noindent\fdef\test {\xintTrimUnbraced {7}{123456789}}\meaning\test\par
\noindent\fdef\test {\xintTrimUnbraced {-7}{123456789}}\meaning\test\par
```

```
macro:->12345678910
macro:->{8}{9}
macro:->12
macro:->89
macro:->12
```

17.9 `\xintListWithSep`

nf ★ `\xintListWithSep{⟨sep⟩}{⟨list⟩}` inserts the separator `⟨sep⟩` in-between all items of the given list of braced items (or individual tokens). The items are fetched as does \TeX with undelimited macro arguments, thus they end up unbraced in output. If the `⟨list⟩` is only one (or multiple) space tokens, the output is empty.

The list argument `⟨list⟩` gets *f-expanded* first (thus if it is a macro whose contents are braced items, the first opening brace stops the expansion, and it is as if the macro had been expanded once.) The separator `⟨sep⟩` is not pre-expanded, it ends up as is in the output (if the `⟨list⟩` contained at least two items.)

nn ★ The variant `\xintListWithSepNoExpand` does the same job without the initial expansion of the `⟨list⟩` argument.

```
\edef\foo{\xintListWithSep{,}{123456789{10}{11}{12}}}\meaning\foo\newline
\edef\foo{\xintListWithSep{:}{\xintiiFac{20}}}\meaning\foo\newline
\oodef\FOO{\xintListWithSepNoExpand{\FOO}{\bat\baz\biz\buz}}\meaning\FOO\newline
% a braced item or a space stops the f-expansion:
\oodef\foo{\xintListWithSep{\FOO}{\bat\baz\biz\buz}}\meaning\foo\newline
\oodef\foo{\xintListWithSep{\FOO}{\bat\baz\biz\buz}}\meaning\foo\par
```

```
macro:->1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
macro:->2:4:3:2:9:0:2:0:0:8:1:7:6:6:4:0:0:0:0
macro:->\bat \FOO \baz \FOO \biz \FOO \buz
macro:->\bat \FOO \baz \FOO \biz \FOO \buz
macro:->\bat \FOO \baz \FOO \biz \FOO \buz
```

17.10 `\xintApply`

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

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

The `⟨list⟩` may itself be some macro expanding (in the previously described way) to the list of tokens to which the macro `⟨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{\xintiiFac {20}}=7567097991823359999
```

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

17.11 `\xintApplyUnbraced`

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

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

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

```
\def\macro #1{\expandafter\def\csname myself#1\endcsname {#1}}
\xintApplyUnbraced\macro{{elta}{eltb}{eltc}}
\begin{enumerate}[nosep,label=(\arabic{*})]
\item \meaning\myselfelta
\item \meaning\myselfeltb
\item \meaning\myselfeltc
\end{enumerate}
```

- (1) macro:->elta
- (2) macro:->eltb
- (3) macro:->eltc


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

17.12 `\xintSeq`

$\left[\frac{\text{num}}{x}\right] \frac{\text{num}}{x} \frac{\text{num}}{x}$ ★ `\xintSeq[d]{x}{y}` generates expandably `{x}{x+d}...` up to and possibly including `{y}` if `d>0` or down to and including `{y}` if `d<0`. Naturally `{y}` is omitted if `y-x` is not a multiple of `d`. If `d=0` the macro returns `{x}`. If `y-x` and `d` have opposite signs, the macro returns nothing. If the optional argument `d` is omitted it is taken to be the sign of `y-x`. Hence `\xintSeq {1}{0}` is not empty but `{1}{0}`. But `\xintSeq [1]{1}{0}` is empty.

The arguments `x` and `y` are expanded inside a `\numexpr` so they may be count registers or a `\TeX` `\value{countername}`, or arithmetic with such things.

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

 When the macro is used without the optional argument `d`, it can only generate up to about 5000 numbers, the precise value depends upon some `TeX` memory parameter (input save stack).

With the optional argument `d` the macro proceeds differently (but less efficiently) and does not stress the input save stack.

17.13 `\xintloop`, `\xintbreakloop`, `\xintbreakloopaddo`, `\xintloopskiptonext`

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

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

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

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

One will usually employ some count registers to manage the exit test from the loop; this breaks expandability, see `\xintilloop` for an expandable integer indexed loop. Use in alignments will be complicated by the fact that cells create groups, and also from the fact that any encountered un-expandable material will cause the `TeX` input scanner to insert `\endtemplate` on each encountered `&`

or `\cr`; thus `\xintbreakloop` may not work as expected, but the situation can be resolved via `\xintt_firstofone{&}` or use of `\TAB` with `\def\tab{&}`. It is thus simpler for alignments to use rather than `\xintloop` either the expandable `\xintApplyUnbraced` or the non-expandable but alignment compatible `\xintApplyInline`, `\xintFor` or `\xintFor*`.

As an example, let us suppose we have two macros `\A{i}{j}` and `\B{i}{j}` behaving like (small) integer valued matrix entries, and we want to define a macro `\C{i}{j}` giving the matrix product (*i* and *j* may be count registers). We will assume that `\A[I]` expands to the number of rows, `\A[J]` to the number of columns and want the produced `\C` to act in the same manner. The code is very dispendious in use of `\count` registers, not optimized in any way, not made very robust (the defined macro can not have the same name as the first two matrices for example), we just wanted to quickly illustrate use of the nesting capabilities of `\xintloop`.⁸⁴

```
\newcount\rowmax \newcount\colmax \newcount\summax
\newcount\rowindex \newcount\colindex \newcount\sumindex
\newcount\tmpcount
\makeatletter
\def\MatrixMultiplication #1#2#3{%
  \rowmax #1[I]\relax
  \colmax #2[J]\relax
  \summax #1[J]\relax
  \rowindex 1
  \xintloop % loop over row index i
  {\colindex 1
  \xintloop % loop over col index k
  {\tmpcount 0
  \sumindex 1
  \xintloop % loop over intermediate index j
  \advance\tmpcount \numexpr #1\rowindex\sumindex*#2\sumindex\colindex\relax
  \ifnum\sumindex<\summax
  \advance\sumindex 1
  \repeat }%
  \expandafter\edef\csname\string#3{\the\rowindex.\the\colindex}\endcsname
  {\the\tmpcount}%
  \ifnum\colindex<\colmax
  \advance\colindex 1
  \repeat }%
  \ifnum\rowindex<\rowmax
  \advance\rowindex 1
  \repeat
  \expandafter\edef\csname\string#3{I}\endcsname{\the\rowmax}%
  \expandafter\edef\csname\string#3{J}\endcsname{\the\colmax}%
  \def #3##1{\ifx[##1\expandafter\Matrix@helper@size
  \else\expandafter\Matrix@helper@entry\fi #3{##1}}%
}
\def\Matrix@helper@size #1#2#3{\csname\string#1{#3}\endcsname }%
\def\Matrix@helper@entry #1#2#3%
  {\csname\string#1{\the\numexpr#2.\the\numexpr#3}\endcsname }%
\def\A #1{\ifx[#1\expandafter\A@size
  \else\expandafter\A@entry\fi {#1}}%
\def\A@size #1#2{\ifx I#23\else4\fi}% 3rows, 4columns
\def\A@entry #1#2{\the\numexpr #1+#2-1\relax}% not pre-computed...
\def\B #1{\ifx[#1\expandafter\B@size
  \else\expandafter\B@entry\fi {#1}}%
\def\B@size #1#2{\ifx I#24\else3\fi}% 4rows, 3columns
\def\B@entry #1#2{\the\numexpr #1-#2\relax}% not pre-computed...
```

⁸⁴ for a more sophisticated implementation of matrix multiplication, inclusive of determinants, inverses, and display utilities, with entries big integers or decimal numbers or even fractions see <http://tex.stackexchange.com/a/143035/4686> from November 11, 2013.

```

\makeatother
\MatrixMultiplicationA\B\C \MatrixMultiplicationC\C\D
\MatrixMultiplicationC\D\E \MatrixMultiplicationC\E\F
\begin{multicols}2
  \[\begin{pmatrix}
    \A11&\A12&\A13&\A14\\
    \A21&\A22&\A23&\A24\\
    \A31&\A32&\A33&\A34
  \end{pmatrix}
  \times
  \begin{pmatrix}
    \B11&\B12&\B13\\
    \B21&\B22&\B23\\
    \B31&\B32&\B33\\
    \B41&\B42&\B43
  \end{pmatrix}
  =
  \begin{pmatrix}
    \C11&\C12&\C13\\
    \C21&\C22&\C23\\
    \C31&\C32&\C33
  \end{pmatrix}\]
  \[\begin{pmatrix}
    \C11&\C12&\C13\\
    \C21&\C22&\C23\\
    \C31&\C32&\C33
  \end{pmatrix}^2 = \begin{pmatrix}
    \D11&\D12&\D13\\
    \D21&\D22&\D23\\
    \D31&\D32&\D33
  \end{pmatrix}\]
  \[\begin{pmatrix}
    \C11&\C12&\C13\\
    \C21&\C22&\C23\\
    \C31&\C32&\C33
  \end{pmatrix}^3 = \begin{pmatrix}
    \E11&\E12&\E13\\
    \E21&\E22&\E23\\
    \E31&\E32&\E33
  \end{pmatrix}\]
  \[\begin{pmatrix}
    \C11&\C12&\C13\\
    \C21&\C22&\C23\\
    \C31&\C32&\C33
  \end{pmatrix}^4 = \begin{pmatrix}
    \F11&\F12&\F13\\
    \F21&\F22&\F23\\
    \F31&\F32&\F33
  \end{pmatrix}\]
\end{multicols}

```

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

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

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

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

17.14 `\xintilooop`, `\xintilooopindex`, `\xintouterilooopindex`, `\xintbreakilooop`, `\xintbreakilooopanddo`, `\xintilooopskiptonext`, `\xintilooopskipandredo`

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

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

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

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

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

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

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

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

```
\expandafter\xintbreakilooopanddo\expandafter\macro\xintilooopindex.%
etc.. etc.. \repeat
```

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

```
\xint_afterfi{\expandafter\xintbreakilooopanddo\expandafter\macro\xintilooopindex.}%
\fi etc..etc.. \repeat
```

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

Inside alignments, if the looped-over text contains a `&` or a `\cr`, any un-expandable material before a `\xintilooopindex` will make it fail because of `\endtemplate`; in such cases one can always

either replace & by a macro expanding to it or replace it by a suitable `\firstofone{&}`, and similarly for `\cr`.

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

```
\begin{group}
\edef\z
{\xintilooop [10001+2]
  {\xintilooop [3+2]
    \ifnum\xintouterilooopindex<\numexpr\xintilooopindex*\xintilooopindex\relax
      \xintouterilooopindex,
      \expandafter\xintbreakilooop
    \fi
    \ifnum\xintouterilooopindex=\numexpr
      (\xintouterilooopindex/\xintilooopindex)*\xintilooopindex\relax
    \else
      \repeat
    }% no space here
  \ifnum \xintilooopindex < 10999 \repeat }%
\meaning\z\endgroup
```

macro:->10007, 10009, 10037, 10039, 10061, 10067, 10069, 10079, 10091, 10093, 10099, 10103, 10111, 10133, 10139, 10141, 10151, 10159, 10163, 10169, 10177, 10181, 10193, 10211, 10223, 10243, 10247, 10253, 10259, 10267, 10271, 10273, 10289, 10301, 10303, 10313, 10321, 10331, 10333, 10337, 10343, 10357, 10369, 10391, 10399, 10427, 10429, 10433, 10453, 10457, 10459, 10463, 10477, 10487, 10499, 10501, 10513, 10529, 10531, 10559, 10567, 10589, 10597, 10601, 10607, 10613, 10627, 10631, 10639, 10651, 10657, 10663, 10667, 10687, 10691, 10709, 10711, 10723, 10729, 10733, 10739, 10753, 10771, 10781, 10789, 10799, 10831, 10837, 10847, 10853, 10859, 10861, 10867, 10883, 10889, 10891, 10903, 10909, 10937, 10939, 10949, 10957, 10973, 10979, 10987, 10993, and we should have taken some steps to not have a trailing comma, but the point was to show that one can do that in an `\edef`! See also [subsection 5.2](#) which extracts from this code its way of testing primality.

Let us create an alignment where each row will contain all divisors of its first entry. Here is the output, thus obtained without any count register:

```
\begin{multicols}2
\tabskip1ex \normalcolor
\halign{&\hfil#\hfil\cr
  \xintilooop [1+1]
  {\expandafter\bfseries\xintilooopindex &
  \xintilooop [1+1]
  \ifnum\xintouterilooopindex=\numexpr
    (\xintouterilooopindex/\xintilooopindex)*\xintilooopindex\relax
  \xintilooopindex&\fi
  \ifnum\xintilooopindex<\xintouterilooopindex\space % CRUCIAL \space HERE
  \repeat \cr }%
  \ifnum\xintilooopindex<30
  \repeat
}
\end{multicols}
```

1 1	10 1 2 5 10
2 1 2	11 1 11
3 1 3	12 1 2 3 4 6 12
4 1 2 4	13 1 13
5 1 5	14 1 2 7 14
6 1 2 3 6	15 1 3 5 15
7 1 7	16 1 2 4 8 16
8 1 2 4 8	17 1 17
9 1 3 9	18 1 2 3 6 9 18

```

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

```

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

```

\tabskiplex
\def\firstofone #1{#1}%
\halign{&\hfil#\hfil\cr
\xintloop [1+1]
{\bfseries\xintloopindex\firstofone{&}}%
\xintloop [1+1] \ifnum\xintouterloopindex=\numexpr
(\xintouterloopindex/\xintloopindex)*\xintloopindex\relax
\xintloopindex\firstofone{&}\fi
\ifnum\xintloopindex<\xintouterloopindex\space % \space is CRUCIAL
\repeat \firstofone{\cr}}%
\ifnum\xintloopindex<30 \repeat }

```

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

17.15 `\xintApplyInline`

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

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

```

\def\Macro #1{\advance\cnta #1 , \the\cnta}
\cnta 0
0\xintApplyInline\Macro {3141592653}.

```

0, 3, 4, 8, 9, 14, 23, 25, 31, 36, 39. The first argument `\macro` does not have to be an expandable macro.

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

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

```

\centerline{\normalcolor\begin{tabular}{ccc}
  $$ & $$^2 & $$^3 \\ \hline
\def\Row #1{ #1 & \xintiiSqr {#1} & \xintiiPow {#1}{3} \\ \hline }%
\xintApplyInline \Row {\xintCSVtoList{17,28,39,50,61}}
\end{tabular}}\medskip

```

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

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

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

One may nest various `\xintApplyInline`'s. For example (see the [table](#) on the current page):

```
\begin{figure*}[ht!]
  \centering\phantomsection\label{float}
  \def\Row #1{#1:\xintApplyInline {\Item {#1}}{0123456789}\ }%
  \def\Item #1#2{&\xintiiPow {#1}{#2}}%
  \centeredline {\begin{tabular}{ccccccccc} &0&1&2&3&4&5&6&7&8&9\ \hline
    \xintApplyInline \Row {0123456789}
  \end{tabular}}
\end{figure*}
```

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

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

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

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

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

But see [\xintFor](#).

17.16 `\xintFor`, `\xintFor*`

`\xintFor` is a new kind of for loop.⁸⁵ Rather than using macros for encapsulating list items, its

behaviour is like a macro with parameters: #1, #2, . . . , #9 are used to represent the items for up to nine levels of nested loops. Here is an example:

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

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

Technical notes:

- The #1 is replaced in the iterated-over text exactly as in general TeX macros or LaTeX commands. This spares the user quite a few \expandafter's or other tricks needed with loops which have the values encapsulated in macros, like LaTeX's \@for and \@tfor.
- \xintFor (and \xintFor*) isn't purely expandable: one can not use it inside an \edef. But it may be used, as will be shown in examples, in some contexts such as LaTeX's tabular which are usually hostile to non-expandable loops.
- \xintFor (and \xintFor*) does some assignments prior to executing each iteration of the replacement text, but it acts purely expandably after the last iteration, hence if for example the replacement text ends with a \\, the loop can be used insided a tabular and be followed by a \hline without creating the dreaded ``Misplaced \noalign'' error.
- It does not create groups.
- It makes no global assignments.
- The iterated replacement text may close a group which was opened even before the start of the loop (typical example being with & in alignments).

```
\begin{tabular}{rcccc}
\hline
\xintFor #1 in {A, B, C} \do {%
  #1:\xintFor #2 in {a, b, c, d, e} \do {&($ #2 \to #1 $)}\\ }%
\hline
\end{tabular}
```

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

- There is no facility provided which would give access to a count of the number of iterations as it is technically not easy to do so it in a way working with nested loops while maintaining the ``expandable after done'' property; something in the spirit of \xint-iloopindex is possible but this approach would bring its own limitations and complications. Hence the user is invited to update her own count or LaTeX counter or macro at each iteration, if needed.
- A \macro whose definition uses internally an \xintFor loop may be used inside another \xintFor loop even if the two loops both use the same macro parameter. The loop definition inside \macro must use ## as is the general rule for definitions done inside macros.
- \xintFor is for comma separated values and \xintFor* for lists of braced items; their respective expansion policies differ. They are described later.

⁸⁵ first introduced with [xint 1.09c](#) of 2013/10/09.

Regarding `\xintFor`:

- the spaces between the various declarative elements are all optional,
- in the list of comma separated values, spaces around the commas or at the start and end are ignored,
- if an item must contain itself its own commas, then it should be braced, and the braces will be removed before feeding the iterated-over text,
- the list may be a macro, it is expanded only once,
- items are not pre-expanded. The first item should be braced or start with a space if the list is explicit and the item should not be pre-expanded,
- empty items give empty `#1`'s in the replacement text, they are not skipped,
- an empty list executes once the replacement text with an empty parameter value,
- the list, if not a macro, must be braced.

**fn* Regarding `\xintFor*`:

- it handles lists of braced items (or naked tokens),
- it *f-expands* the list,
- and more generally it *f-expands* each naked token encountered before assigning the `#1` values (gobbling spaces in the process); this makes it easy to simulate concatenation of multiple lists `\x`, `\y`: if `\x` expands to `{1}{2}{3}` and `\y` expands to `{4}{5}{6}` then `{\x\y}` as argument to `\xintFor*` has the same effect as `{1}{2}{3}{4}{5}{6}`.

For a further illustration see the use of `\xintFor*` at the end of [subsection 2.10](#).

- spaces at the start, end, or in-between items are gobbled (but naturally not the spaces inside braced items),
- except if the list argument is a macro (with no parameters), it must be braced.,
- an empty list leads to an empty result.

The macro `\xintSeq` which generates arithmetic sequences is to be used with `\xintFor*` as its output consists of successive braced numbers (given as digit tokens).

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

```
stuff with -7
stuff with -5
stuff with -3
stuff with -1
stuff with 1
```

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

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

This is a general remark applying for any nesting of loops, one should avoid recreating the inner lists of arguments at each iteration of the outer loop.

When the loop is defined inside a macro for later execution the `#` characters must be doubled.⁸⁶

For example:

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

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

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

⁸⁶ sometimes what seems to be a macro argument isn't really; in `\raisebox{1cm}{\xintFor #1 in {a,b,c}\do {#1}}` no doubling should be done.

The iterated macros as well as the list items are allowed to contain explicit `\par` tokens.

17.17 `\xintifForFirst`, `\xintifForLast`

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

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

Pay attention to these implementation features:

- if an inner `\xintFor` loop is positioned before the `\xintifForFirst` or `\xintifForLast` of the outer loop it will contaminate their settings. This applies also naturally if the inner loop arises from the expansion of some macro located before the outer conditionals.

One fix is to make sure that the outer conditionals are expanded before the inner loop is executed, e.g. this will be the case if the inner loop is located inside one of the branches of the conditional.

Another approach is to enclose, if feasible, the inner loop in a group of its own.

- if the replacement text closes a group (e.g. from a `&` inside an alignment), the conditionals will lose their ascribed meanings and end up possibly undefined, depending whether there is some outer loop whose execution started before the opening of the group.

The fix is to arrange things so that the conditionals are expanded before \TeX encounters the closing-group token.

17.18 `\xintBreakFor`, `\xintBreakForAndDo`

One may immediately terminate an `\xintFor` or `\xintFor*` loop with `\xintBreakFor`.

As it acts by clearing up all the rest of the replacement text when encountered, it will not work from inside some `\if...\fi` without suitable `\expandafter` or swapping technique.

Also it can't be used from inside braces as from there it can't see the end of the replacement text.

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

17.19 `\xintintegers`, `\xintdimensions`, `\xintrationals`

If the list argument to `\xintFor` (or `\xintFor*`, both are equivalent in this context) is `\xintintegers` (equivalently `\xintegers`) or more generally `\xintintegers[start+delta]` (*the whole within braces!*)⁸⁷, then `\xintFor` does an infinite iteration where `#1` (or `#2`, ..., `#9`) will run through the arithmetic sequence of (short) integers with initial value `start` and increment `delta` (default values: `start=1`, `delta=1`; if the optional argument is present it must contain both of them, and they may be explicit integers, or macros or count registers). The `#1` (or `#2`, ..., `#9`) will stand for `\numexpr <opt sign><digits>\relax`, and the literal representation as a string of digits can thus be obtained as `\the#1` or `\number#1`. Such a `#1` can be used in an `\ifnum` test with no need to be postfixed with a space or a `\relax` and one should *not* add them.

If the list argument is `\xintdimensions` or more generally `\xintdimensions[start+delta]` (*within braces!*), then `\xintFor` does an infinite iteration where `#1` (or `#2`, ..., `#9`) will run through the

⁸⁷ the `start+delta` optional specification may have extra spaces around the plus sign of near the square brackets, such spaces are removed. The same applies with `\xintdimensions` and `\xintrationals`.

arithmetic sequence of dimensions with initial value `start` and increment `delta`. Default values: `start=0pt`, `delta=1pt`; if the optional argument is present it must contain both of them, and they may be explicit specifications, or macros, or dimen registers, or length macros in \TeX (the stretch and shrink components will be discarded). The `#1` will be `\dimexpr <opt sign><digits>sp\relax`, from which one can get the literal (approximate) representation in points via `\the#1`. So `#1` can be used anywhere \TeX expects a dimension (and there is no need in conditionals to insert a `\relax`), and one should *not* do it), and to print its value one uses `\the#1`. The chosen representation guarantees exact incrementation with no rounding errors accumulating from converting into points at each step.

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

```
\begingroup\small
\noindent\parbox{\dimexpr\linewidth-3em}{\color[named]{OrangeRed}%
\xintFor #1 in {\xintrationals [10/21+1/21]} \do
{#1=\xintifInt {#1}
{\textcolor{blue}{\xintTrunc{10}{#1}}}
{\xintTrunc{10}{#1}}% display in blue if an integer
\xintifGt {#1}{1.123}{\xintBreakFor}{, }%
}}
\endgroup\smallskip
10/21=0.4761904761,      11/21=0.5238095238,      12/21=0.5714285714,      13/21=0.6190476190,
14/21=0.6666666666,      15/21=0.7142857142,      16/21=0.7619047619,      17/21=0.8095238095,
18/21=0.8571428571,      19/21=0.9047619047,      20/21=0.9523809523,      21/21=1.0000000000,
22/21=1.0476190476, 23/21=1.0952380952, 24/21=1.1428571428
```

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

```
\noindent\parbox{\dimexpr.7\linewidth}{\raggedright
\xintFor #1 in {\xintrationals [0.000+0.125]} \do
{\edef\tmp{\xintTrunc{3}{#1}}%
\xintifInt {#1}
{\textcolor{blue}{\tmp}}
{\tmp}%
\xintifGt {#1}{2}{\xintBreakFor}{, }%
}}\smallskip
0, 0.125, 0.250, 0.375, 0.500, 0.625, 0.750, 0.875, 1.000, 1.125,
1.250, 1.375, 1.500, 1.625, 1.750, 1.875, 2.000, 2.125
```

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

17.20 `\xintForpair`, `\xintForthree`, `\xintForfour`

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

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

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

`\xintForpair` must be followed by either `#1#2`, `#2#3`, `#3#4`, ..., or `#8#9` with `#1` usable as an alias for `#1#2`, `#2` as alias for `#2#3`, etc ... and similarly for `\xintForthree` (using `#1#2#3` or simply `#1`, `#2#3#4` or simply `#2`, ...) and `\xintForfour` (with `#1#2#3#4` etc...).

Nesting works as long as the macro parameters are distinct among `#1`, `#2`, ..., `#9`. A macro which expands to an `\xintFor` or a `\xintFor(pair, three, four)` can be used in another one with no constraint about using distinct macro parameters.

`\par` tokens are accepted in both the comma separated list and the replacement text.

17.21 `\xintAssign`

`\xintAssign<braced things>\to<as many cs as they are things>` defines (without checking if something gets overwritten) the control sequences on the right of `\to` to expand to the successive tokens or braced items located to the left of `\to`. `\xintAssign` is not an expandable macro.

f-expansion is first applied to the material in front of `\xintAssign` which is fetched as one argument if it is braced. Then the expansion of this argument is examined and successive items are assigned to the macros following `\to`. There must be exactly as many macros as items. No check is done. The macro assignments are done with removal of one level of brace pairs from each item.

After the initial *f-expansion*, each assigned (brace-stripped) item will be expanded according to the setting of the optional parameter.

For example `\xintAssign [e]...` means that all assignments are done using `\edef`. With `[f]` the assignments will be made using `\fdef`. The default is simply to make the definitions with `\def`, corresponding to an empty optional parameter `[]`. Possibilities for the optional parameter are: `[]`, `[g]`, `[e]`, `[x]`, `[o]`, `[go]`, `[oo]`, `[goo]`, `[f]`, `[gf]`. For example `[oo]` means a double expansion.

```
\xintAssign \xintiiDivision{1000000000000}{133333333}\to\Q\R
\meaning\Q\newline
\meaning\R\newline
\xintAssign {\xintiiDivision{1000000000000}{133333333}}\to\X
\meaning\X\newline
\xintAssign [oo]{\xintiiDivision{1000000000000}{133333333}}\to\X
\meaning\X\newline
\xintAssign \xintiiPow{7}{13}\to\SevenToThePowerThirteen
\meaning\SevenToThePowerThirteen\par
```

macro:->7500

macro:->2500

macro:->\xintiiDivision {1000000000000}{133333333}

macro:->{7500}{2500}

macro:->96889010407

Two special cases:

- if after this initial expansion no brace is found immediately after `\xintAssign`, it is assumed that there is only one control sequence following `\to`, and this control sequence is then defined via `\def` (or what is set-up by the optional parameter) to expand to the material between `\xintAssign` and `\to`.
- if the material between `\xintAssign` and `\to` is enclosed in two brace pairs, the first brace pair is removed, then the *f-expansion* is immediately stopped by the inner brace pair, hence `\xintAssign` now finds a unique item and thus defines only a single macro to be this item, which is now stripped of the second pair of braces.

Note: prior to release 1.09j, `\xintAssign` did an `\edef` by default for each item assignment but it now does `\def` corresponding to no or empty optional parameter.

It is allowed for the successive braced items to be separated by spaces. They are removed during the assignments. But if a single macro is defined (which happens if the argument after *f-expansion* does not start with a brace), naturally the scooped up material has all intervening spaces, as it is considered a single item. But an upfront initial space will have been absorbed by *f-expansion*.

```
\def\X{ {a} {b} {c} {d} }\def\Y{ u {a} {b} {c} {d} }
\xintAssign\X\to\A\B\C\D
\xintAssign\Y\to\Z
\meaning\A, \meaning\B, \meaning\C, \meaning\D+++
\meaning\Z+++
\par
```

macro:->a, macro:->b, macro:->c, macro:->d+++

macro:->u {a} {b} {c} {d} +++

As usual successive space characters in input make for a single TeX space token.

17.22 `\xintAssignArray`

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

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

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

`\xintAssignArray` admits an optional parameter, for example `\xintAssignArray [e]` means that the definitions of the macros will be made with `\edef`. The empty optional parameter (default) means that definitions are done with `\def`. Other possibilities: `[]`, `[o]`, `[oo]`, `[f]`. Contrarily to `\xintAssign` one can not use the `g` here to make the definitions global. For this, one should rather do `\xintAssignArray` within a group starting with `\globaldefs 1`.

17.23 `\xintDigitsOf`

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

```
\xintDigitsOf\xintiiPow {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`.

17.24 `\xintRelaxArray`

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

This documentation has been compiled without the source code, which is available in the separate file:
`sourcexint.pdf`,
which will open in a PDF viewer via `texdoc sourcexint.pdf`.
To produce a single file including both the user documentation and the source code:

- run `etex` on `xint.dtx` to generate `xint.tex` among other files,
- edit `xint.tex` to set the `\NoSourceCode` toggle within it to `0`,
- run `make clean` and then `make xint.pdf`.

This will need `latexmk`; if not available you will need in replacement of the last step to execute manually `latex` on `xint.tex` (thrice) then `dvipdfmx`.