The xint bundle

Jean-François Burnol
jfbu (at) free (dot) fr
Package version: 1.09h (2013/11/28)

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

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

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

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

Further modules:

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

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

xintgcd implements the Euclidean algorithm and its typesetting.

xintcfrac deals with the computation of continued fractions.

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

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

Contents

Documentation generated from the source file with timestamp “05-12-2013 at 17:08:06 CET”.

1 Quick introduction

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

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

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

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

The package macros expand their arguments[1]; as they are themselves completely expandable, this means that one may nest them arbitrarily deep to construct complicated (and still completely expandable) formulas. But one will presumably prefer to use the (expandable!) `\xintexpr ... \relax` parser as it allows infix notations, function names (corresponding to some of the package macros), comparison operators, boolean operators, 2way and 3way conditionals.

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

```
\def\allowsplits #1%
    {\ifx #1\relax \else #1\hskip 0pt plus 1pt\relax\expandafter\allowsplits\fi}%
\def\printnumber #1{\expandafter\expandafter\expandafter\allowsplits #1\relax }%
```

[1] see in section 7 the related explanations.

```
%% (all macros from the xint bundle expand in two steps to their final output).
```

An alternative (footnote 10) is to suitably configure the thousand separator with the numprint package (does not work in math mode; I also tried siunitx but even in text mode could not get it to break numbers accross lines). Recently I became aware of the seqsplit package[2] which can be used to achieve this splitting accross lines, and does work in inline math mode.

The utilities provided by **xinttools** (section 25), some completely expandable, others not, are of independent interest. Their use is illustrated through various examples: among those, it is shown in subsection 25.26 how to implement in a completely expandable way the quick sort algorithm and also how to illustrate it graphically. Other examples include some dynamically constructed alignments with automatically computed prime number cells (subsection 25.11, subsection 25.21).

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

2 Recent changes

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

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

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

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

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

- new `\xintZapFirstSpaces`, `\xintZapLastSpaces`, `\xintZapSpaces`, `\xintZapSpacesB`, for expandably stripping away leading and/or ending spaces.
- `\xintCSVtoList` by default uses `\xintZapSpacesB` to strip away spaces around commas (or at the start and end of the comma separated list).
- also the `\xintFor` loop will strip out all spaces around commas and at the start and the end of its list argument; and similarly for `\xintForpair`, `\xintForthree`, `\xintForfour`.
- `\xintFor` *et al.* accept all macro parameters from `#1` to `#9`.
- for reasons of inner coherence some macros previously with one extra '`i`' in their names (e.g. `\xintiMON`) now have a doubled '`ii`' (`\xintiiMON`) to indicate that they skip the overhead of parsing their inputs via `\xintNum`. Macros with a *single* '`i`' such as `\xintiAdd` are those which maintain the non-**xintfrac** output format for big integers, but do parse their inputs via `\xintNum` (since release `1.09a`).

[2] http://ctan.org/pkg/seqsplit

They too may have doubled-i variants for matters of programming optimization when working only with (big) integers and not fractions or decimal numbers, interested advanced users should check the code source.

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

- new \xintintegers, \xintdimensions, \xintrationals for infinite \xintFor loops, interrupted with \xintBreakFor and \xintBreakForAndDo.
- new \xintifForFirst, \xintifForLast for the \xintFor and \xintFor* loops,
- the \xintFor and \xintFor* loops are now \long, the replacement text and the items may contain explicit \par's.
- bug fix, the \xintFor loop (not \xintFor*) did not correctly detect an empty list.
- new conditionals \xintifCmp, \xintifInt, \xintifOdd.
- bug fix, \xintiSqrt {0} crashed. :-((
- the documentation has been enriched with various additional examples, such as the the quick sort algorithm illustrated or the computation of prime numbers (subsection 25.11, subsection 25.14, subsection 25.21).
- the documentation explains with more details various expansion related issues, particularly in relation to conditionals.

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

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

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

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

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

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

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

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

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

- \xintnumexpr .. \relax is \xintexpr round(..) \relax.
- \xintNewExpr now works with the standard macro parameter character #.
- both regular \xintexpr-essions and commands defined by \xintNewExpr will work with comma separated lists of expressions,
- new commands \xintFloor, \xintCeil, \xintMaxof, \xintMinof (package **xintfrac**), \xintGCDof, \xintLCM, \xintLCMof (package **xintgcd**), \xintifLt, \xintifGt, \xintifSgn, \xintANDof, ...
- The arithmetic macros from package **xint** now filter their operands via \xintNum which means that they may use directly count registers and \numexpr-essions without having to prefix them by \the. This is thus similar to the situation holding previously but with **xintfrac** loaded.
- a bug introduced in 1.08b made \xintCmp crash when one of its arguments was zero. :-((

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

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

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

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

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

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

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

- The **xintfrac** macros accept numbers written in scientific notation, the \xintFloat command serves to output its argument with a given number D of significant figures. The value of D is either given as optional argument to \xintFloat or set with \xintDigits := D;. The default value is 16.
- The **xintexpr** package is a new core constituent (which loads automatically **xintfrac** and **xint**) and implements the expandable expanding parsers
  \xintexpr . . . \relax, and its variant \xintfloatexpr . . . \relax
  allowing on input formulas using the standard form with infix operators +, -, *, /, and ^, and arbitrary levels of parenthesizing. Within a float expression the operations are executed according to the current value of \xintDigits. Within an \xintexpr-ession the binary operators are computed exactly.
- The floating point precision D is set (this is a local assignment to a \mathchar variable) with \xintDigits := D; and queried with \xinttheDigits. It may be set to anything up to 32767.[3] The macro incarnations of the binary operations admit an optional argument which will replace pointwise D; this argument may exceed the 32767 bound.

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

- To write the `\xintexpr` parser I benefited from the commented source of the LaTeX3 parser; the `\xintexpr` parser has its own features and peculiarities. See its documentation.

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

3 Overview

The main characteristics are:

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

'Arbitrarily big': this means with less than `2^31-1=2147483647` digits, as most of the macros will have to compute the length of the inputs and these lengths must be treatable as TeX integers, which are at most `2147483647` in absolute value. This is a distant theoretical upper bound, the true limitation is from the *time* taken by the expansion-compatible algorithms, this will be commented upon soon.

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

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

The bigintcalc package by Heiko Oberdiek provided expandable operations (using some of `\numexpr` possibilities, when available) on arbitrarily big integers, beyond the TeX bound. The present package does this again, using more of `\numexpr` (**`xint`** requires the ε-TeX extensions) for higher speed, and also on fractions, not only integers. Arbitrary precision floating points operations are a derivative, and not the initial design goal.[4, 5]

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

The **`xint`** package can be used for 24, 40, etc... significant figures but one rather quickly (not much beyond 100 figures perhaps) hits against a 'wall' created by the constraint of expandability: currently, multiplying out two one-hundred digits numbers takes circa 80 or 90 times longer than for two ten-digits numbers, which is reasonable, but multiplying out two one-thousand digits numbers takes more than 500 times longer than for two one hundred-digits numbers. This shows that the algorithm is drifting from quadratic to cubic

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

[5] 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.)

in that range. On my laptop multiplication of two 1000-digits numbers takes some seconds, so it can not be done routinely in a document.[6]

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

The second conclusion, somewhat depressing after all the hard work, is that if one really wants to do computations with *hundreds* of digits, one should drop the expandability requirement. And indeed, as clearly demonstrated long ago by the pi computing file by D. Roegel one can program 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.[7] [8]

4 Missing things

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

5 Some examples

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

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

123456^99:

```
\xintiPow{123456}{99}: 114738181166266556633273330008454586747025480423426102975889545437359089469703202762264705426632058346902708682211681334152500324038762776168953222117634295872033762216088606915850757168019716710712087697033536507377487778737784987816067499997983665812517232752154970541659566738491153332674854107560766971890623518995832377826369998110953239399323518999222056458781270149587767914316773543725385844594871559412151974163986661258969837372587167573949494355201709502618658016651990307184144322311696783769 6
```

1234/56789 with 1500 digits after the decimal point:

```
\xintTrunc{1500}{1234/56789}\dots: 0.021729560302171195125816619415731
```

[6] without entering into too much technical details, the source of this 'wall' is that when dealing with two long operands, when one wants to pick some digits from the second one, one has to jump above all digits constituting the first one, which can not be stored away: expandability forbids assignments to memory storage. One may envision some sophisticated schemes, dealing with this problem in less naive ways, trying to move big chunks of data higher up in the input stream and come back to it later, etc...; but each 'better' algorithm adds overhead for the smaller inputs. For example, I have another version of addition which is twice faster on inputs with 500 digits or more, but it is slightly less efficient for 50 digits or less. This 'wall' dissuaded me to look into implementing 'intelligent' multiplication which would be sub-quadratic in a model where storing and retrieving from memory do not cost much.

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

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

919914067865255595273732589057740055292398175703041081899663667259504481501699272746482593460001056542640300058109845216503196041486907675782281779922168025497895719241402384264558277131134550705242212400288788321682015883357692510873584673088098047157019845392593636091496592649985032312595749176777192766204722745602141259751008117769286305446477310746799556252091073975593865009068657662575498776171441652432689429290883797918610998608885523604923488703798270791878708904893553289897691454330944373029988201940516649351106728415714310870062864287097853457535790381940164468471006709045765905368997517124795294863441863741217489302505766961911637817182905140079945059782704396978288048741833805842680800859321347444047262674109422599447076018242969589181003363327404955184983007272535174065399989434573596999418901547834968039585130923242177182200778319745021042807585976157354417228688654492947577875997112116783179841166423074891264153269119019528429801546074063639085034073500149676874042508232228072337952772543978587402489918822307136945535226892532004437479089260244061349909313423374245012238285583475673105707091162020813890013911144763950765112962017292081212910951064466710102308545669055626970011798059483350648893271584285689129937135712902146542464209618059835531528993290954234094631002482875204705136558136258782510697494233038088362182817094859920054940217295603021711951258166194157319199140678652555952737325890057740055292398175703041081899663667...

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

\xinttheexpr trunc(.99^-100,200)\relax\dots: 2.731999026429026003846671721257837435505351642938572070833430572508246455518705343044814301378480614036805562476501925307034269685489153194616612271015920671913840348851485747943086470963920731779793 03...

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

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

-220045702773594816771390169652074193009609478853×(7^200-3^200)+1432589493627636931859130683268320465474416863387714089158381672478991921132820119127462437158039177754976857191287693144240605066991456336143205677696774891×(2^200-1)=1803403947125

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

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

$$22206980239027589097 = 2 \times 8169486210102119256 + 5868007818823350585$$
$$8169486210102119256 = 1 \times 5868007818823350585 + 2301478391278768671$$
$$5868007818823350585 = 2 \times 2301478391278768671 + 1265051036265813243$$
$$2301478391278768671 = 1 \times 1265051036265813243 + 1036427355012955428$$
$$1265051036265813243 = 1 \times 1036427355012955428 + 228623681252857815$$
$$1036427355012955428 = 4 \times 228623681252857815 + 121932630001524168$$

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

$$228623681252857815 = 1 \times 121932630001524168 + 106691051251333647$$
$$121932630001524168 = 1 \times 106691051251333647 + 15241578750190521$$
$$106691051251333647 = 7 \times 15241578750190521 + 0$$

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

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

The complete series, extended to infinity, has value $\frac{\pi^2}{144} - \frac{1}{162} = 0.062,366,079,945,836,595,346,844,45\ldots$ [10] I also used (this is a lengthier computation than the one above) **xintseries** to evaluate the sum with 100,000 terms, obtaining 16 correct decimal digits for the complete sum. The coefficient macro must be redefined to avoid a `\numexpr` overflow, as `\numexpr` inputs must not exceed `2^31-1`; my choice was:

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

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

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

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

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

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

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

The tex run creates a file `myfile-out.tex`, and then writes to it the quotient from the euclidean division of `2^{1000}` by `100!`. The number of digits is `\xintLen{\xintQuo{\xintPow{2}{1000}}{\xintFac{100}}}` which expands (in two steps) and tells us that `[2^{1000}/100!]` has 144 digits. This is not so many, let us print them here: `11481324964150750548227839387255106625980551778418617288366347806582654189470473797041953579887663048435826506006150374953170779311862777482960 1`.

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

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

[10] This number is typeset using the numprint package, with `\npthousandsep {,\hskip 1pt plus .5pt minus .5pt}`. But the breaking accross lines works only in text mode. The number itself was (of course...) computed initially with **xint**, with 30 digits of π as input. See how **xint may compute π from scratch**.

The \printnumber macro is not part of the package and would need additional thinking for more general use.[11] It may be used as \printnumber {\xintQuo{\xintPow {2}{1000}}{\xintFac{100}}}, or as \printnumber\mynumber if the macro \mynumber was previously defined via an \edef, as for example:

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

or as \expandafter\printnumber\expandafter{\mynumber}, if the macro \mynumber is defined by a \newcommand or a \def; using seven rather than three \expandafter's in \printnumber would allow to use it directly as \printnumber\mynumber when \mynumber has been defined via a \def or \newcommand using a chain of package macros.

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

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

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

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

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

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

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

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

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

6 Origins of the package

Package bigintcalc by Heiko Oberdiek already provides expandable arithmetic operations on "big integers", exceeding the TEX limits (of 2^{31}-1), so why another[13] one?

I got started on this in early March 2013, via a thread on the c.t.tex usenet group,

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

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

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

where Ulrich Diez used the previously cited package together with a macro (\ReverseOrder) which I had contributed to another thread.[14] What I had learned in this other thread thanks to interaction with Ulrich Diez and GL on expandable manipulations of tokens motivated me to try my hands at addition and multiplication.

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

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

The present package is the result of this initial questioning.

7 Expansions

By convention in this manual f-expansion ("full expansion" or "full first expansion") is the process of expanding repeatedly the first token seen until hitting against something not further expandable like an unexpandable TeX-primitive or an opening brace { or a character (inactive). For those familiar with LaTeX3 (which is not used by **xint**) this is what is called in its documentation full expansion. Technically, macro arguments in **xint** which are submitted to such a f-expansion are so via prefixing them with \romannumeral-`0. An explicit or implicit space token stops such an expansion and is gobbled.

Most of the package macros, and all those dealing with computations, are expandable in the strong sense that they expand to their final result via this f-expansion. Again copied from LaTeX3 documentation conventions, this will be signaled in the description of the
⋆ macro by a star in the margin. All[15] expandable macros of the **xint** packages completely expand in two steps.

Furthermore the macros dealing with computations, as well as many utilities from **xinttools**, apply this process of f-expansion to their arguments. Again from LaTeX3's conventions this will be signaled by a margin annotation. Some additional parsing which is done
Num f / f Frac by most macros of **xint** is indicated with a variant; and the extended fraction parsing done by most macros of **xintfrac** has its own symbol. When the argument has a priori to obey the TeX bound of 2147483647 it is systematically fed to a \numexpr..\relax hence the
num x expansion is then a *complete* one, signaled with an x in the margin. This means not only complete expansion, but also that spaces are ignored, infix algebra is possible, count registers are allowed, etc...

$*f$ The \xintApplyInline and \xintFor* macros from **xinttools** apply a special iterated f-expansion, which gobbles spaces, to all those items which are found *unbraced* from left to right in the list argument; this is denoted specially as here in the margin. Some other
$f \to * f$ Frac 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

[14] the \ReverseOrder could be avoided in that circumstance, but it does play a crucial rôle here.
[15] except \xintloop and \xintiloop.

parsing, this is signaled as here in the margin where the signification of the $*$ is thus a bit different from the previous case.

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

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

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

Summary of important expansion aspects:

1. the macros *f*-expand their arguments, this means that they expand the first token seen (for each argument), then expand, etc..., until something un-expandable such as a digit or a brace is hit against. This example
   ```
   \def\x{98765}\def\y{43210}\xintAdd {\x}{\x\y}
   ```
   is *not* a legal construct, as the `\y` will remain untouched by expansion and not get converted into the digits which are expected by the sub-routines of `\xintAdd`. It is a `\numexpr` which will expand it and an arithmetic overflow will arise as `9876543210` exceeds the TeX bounds.

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

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

   One can use naive `\if..\fi` things inside an `\xinttheexpr`-ession and cousins, as long as the test is expandable, for example
   ```
   \xintthenumexpr\ifnum3>2 143\else 33\fi 0^2\relax→2044900=1430^2
   ```

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

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

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

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

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

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

   If, for some reason, it is important to create a macro expanding in two steps to its final value, one may either do:
```
\def\AplusBC #1#2#3{\romannumeral-`0\xintAdd {#1}{\xintMul {#2}{#3}}}
```
or use the *lowercase* form of `\xintAdd`:
```
\def\AplusBC #1#2#3{\romannumeral0\xintadd {#1}{\xintMul {#2}{#3}}}
```
and then `\AplusBC` will share the same properties as do the other **xint** 'primitive' macros.

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

Since release `1.07` the `\xintNewExpr` command automatizes the creation of such expandable macros:
```
\xintNewExpr\AplusBC[3]{#1+#2*#3}
```
creates the `\AplusBC` macro doing the above and expanding in two expansion steps.

8 Input formats

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

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

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

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

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

2. the macro `\xintNum` normalizes into strict format an input having arbitrarily many minus and plus signs, followed by a string of zeros, then digits:
```
\xintNum {+-+-+----++-++----000000000009876543210}=-9876543210
```
Num f The extended integer format is thus for the arithmetic macros of **xint** which automatically parse their arguments via this `\xintNum`.

Frac *f*

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

   ```
   \xintAdd {+--0367.8920280/-++278.289287}{-109.2882/+270.12898}
   ```

   Incidentally this evaluates to

   ```
   =-129792033529284840/7517400124223726[-1]
   =-6489601676464242/3758700062111863 (irreducible)
   =-1.72655481129771093694248704898677881556360055242806...
   ```

   where the second line was produced with `\xintIrr` and the next with `\xintTrunc {50}` to get fifty digits of the decimal expansion following the decimal mark. Scientific notation is accepted for both numerator and denominator of a fraction, and is produced on output by `\xintFloat`:

   ```
   \xintAdd{10.1e1}{101.010e3}=101111/1[0]
   \xintFloatAdd{10.1e1}{101.010e3}=1.011110000000000e5
   \xintFloat{\xintiPow {2}{100}}=1.267650600228229e30
   ```

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

   ```
   \xintPRaw{\xintAdd{10.1e1}{101.010e3}}=101111
   \xintRaw{1.234e5/6.789e3}=1234/6789[2]
   \xintFloat[24]{1/66049}=1.51402746445820527184363e-5
   ```

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

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

A number can start directly with a decimal point:

```
\xintPow{-.3/.7}{11}=-177147/1977326743[0]
\xinttheexpr (-.3/.7)^11\relax=-177147/1977326743[0]
```

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

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

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

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

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

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

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

num *x*

1. 'short' integers, *i.e.* less than (or equal to) in absolute value 2,147,483,647. I will refer to this as the 'TEX' or '`\numexpr`' limit. This is the case for arguments which serve to count or index something. It is also the case for the exponent in the power function and for the argument to the factorial function.[20] When the argument exceeds the TEX bound (either positively or negatively), an error will originate from a `\numexpr` expression and it may sometimes be followed by a more specific error 'message' from a package macro.

f

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

Num *f*

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

Frac *f*

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

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

This format with a trailing `[n]` (possibly, `n=0`) is accepted on input but it presupposes that the numerator and denominator `A` and `B` are in the strict integer format described above. So `16000/289072[17]` or `3[-4]` are authorized and it is even possible to use `\A/\B[17]` if `\A` expands to `16000` and `\B` to `289072`, or `\A` if `\A` expands to `3[-4]`. However, NEITHER IMPORTANT! the numerator NOR the denominator may then have a decimal point. And, for this format,

[20] the bound has even been lowered for them but the float power function limits the exponent only to the TEX bound, and has a variant with no imposed limit on the exponent; but the result of the computation must in all cases be representable with a power of ten exponent obeying the TEX bound.

[21] A LATEX `\value{countername}` is accepted, if there is nothing else, especially before, in the macro argument.

ONLY the numerator may carry a UNIQUE minus sign (and no superfluous leading zeros; and NO plus sign).

This format with a power of ten represented by a number within square brackets is the output format used by (almost all) **xintfrac** macros dealing with fractions. It is allowed for user input but the parsing is minimal and it is mandatory to follow the above rules. This reduced flexibility, compared to the format without the square brackets, allows chaining package macros without too much speed impact, as they always output computation results in the `A/B[n]` form.

> All computations done by **xintfrac** on fractions are exact. Inputs containing decimal points or scientific parts do not make the package switch to a 'floating-point' mode. The inputs, however long, are always converted into exact internal representations.
>
> Floating point evaluations are done with special macros containing 'Float' in their names, or inside `\xintthefloatexpr`-essions.

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

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

9 Output formats

With package **xintfrac** loaded, the routines `\xintAdd`, `\xintSub`, `\xintMul`, `\xintPow`, `\xintSum`, `\xintPrd` are modified to allow fractions on input,[22] [23] [24] [25] and produce on output a fractional number `f=A/B[n]` where `A` and `B` are integers, with `B` positive, and `n` is a "short" integer (*i.e* less in absolute value than `2^{31}-9`). This represents `(A/B)` times `10^n`. The fraction `f` may be, and generally is, reducible, and `A` and `B` may well end up with zeros (*i.e.* `n` does not contain all powers of 10). Conversely, this format is accepted on input (and is parsed more quickly than fractions containing decimal points; the input may be a number without denominator).[26]

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

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

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

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

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

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

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

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

To convert the trailing [n] into explicit zeros either at the numerator or the denominator, use \xintRawWithZeros. In both cases the B is printed even if it has value 1. Conversely (sort of), the macro \xintREZ puts all powers of ten into the [n] (REZ stands for remove zeros). Here also, the B is printed even if it has value 1.

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

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

The macro \xintTrunc{N}{f} prints[27] the decimal expansion of f with N digits after the decimal point.[28] Currently, it does not verify that N is non-negative and strange things could happen with a negative N. A negative f is no problem, needless to say. When the original fraction is negative and its truncation has only zeros, it is printed as -0.0...0, with N zeros following the decimal point:

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

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

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

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

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

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

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

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

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

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

10 Multiple outputs

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

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

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

11 Use of count registers

When an argument to a macro is said in the documentation to have to obey the TEX bound, this means that it is fed to a \numexpr...\relax, hence it is submitted to a complete expansion which must delivers an integer, and count registers and even algebraic expressions with them like \mycountA+\mycountB*17-\mycountC/12+\mycountD are admissible arguments (the slash stands here for the rounded integer division done by \numexpr). This applies in particular to the number of digits to truncate or round with, to the indices of a series partial sum, ...

The macros dealing with long numbers or fractions allow (when not limited to the 'strict integer' format on input) *to some extent* the direct use of count registers and even infix algebra with them inside their arguments: a count register \mycountA or \count 255 is admissible as numerator or also as denominator, with no need to be prefixed by \the or \number. It is possible to have as argument an algebraic expression as would be acceptable by a \numexpr...\relax, under this condition: *each of the numerator and denominator is expressed with at most* eight *tokens*.[29] The slash for rounded division in a \numexpr should be written with braces {/} to not be confused with the **xintfrac** delimiter between numerator and denominator (braces will be removed internally). Example: \mycountA+\mycountB{/}17/1+\mycountA*\mycountB, or \count 0+\count 2{/}17/1+\count 0*\count 2, but in the latter case the numerator has the maximal allowed number of tokens (the braced slash counts for only one).

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

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

IMPORTANT!

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

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

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

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

12 Dimensions

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

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

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

13 `\ifcase`, `\ifnum`, ... constructs

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

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

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

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

If these tests are to be applied to standard TEX short integers, it is more efficient to use (under LATEX) the equivalent conditional tests from the etoolbox[30] package.

14 Assignments

It might not be necessary to maintain at all times complete expandability. For example why not allow oneself the two definitions `\edef\A {\xintQuo{100}{3}}` and `\edef\B {\xintRem {100}{3}}`. A special syntax is provided to make these things more efficient, as the package provides `\xintDivision` which computes both quotient and remainder at the same time:

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

gives `\meaning\A: macro:->`1148132496415075054822783938725510662598055177841861728836634780658265418947047379704195357988766304843582650600615037495317077931186277748296 01` and `\meaning\B: macro:->`5493629452133983225138128786223912807341050049847605059532189961231327664902288388132878702444582075129603152041054804964625083138567652624386837205668069376.

Another example (which uses a macro from the **xintgcd** package):

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In the example above, we deliberately broke all rules of complete expandability, but had we wanted to compute the sum of the digits, not the sum of the squares, we could just have written: \xintiiSum{\xintiPow{2}{100}}=115. Indeed, \xintiiSum is usually used on braced items as in

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

but in the previous example each digit of 2^{100} was treated as one item due to the rules of TEX for parsing macro arguments.

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

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

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

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

15 Utilities for expandable manipulations

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

1.06, \xintApplyUnbraced, since 1.06b, \xintloop and \xintiloop since 1.09g.[31]

As an example the following code uses only expandable operations:

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

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

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

Expandably computing primes is done in subsection 25.10.

16 A new kind of for loop

As part of the utilities coming with the **xinttools** package, there is a new kind of for loop, \xintFor. Check it out (subsection 25.17).

17 A new kind of expandable loop

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

18 Exceptions (error messages)

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

```
\xintError:ArrayIndexIsNegative
\xintError:ArrayIndexBeyondLimit
\xintError:FactorialOfNegativeNumber
\xintError:FactorialOfTooBigNumber
\xintError:DivisionByZero
\xintError:NaN
\xintError:FractionRoundedToZero
```

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

```
\xintError:NotAnInteger
\xintError:ExponentTooBig
\xintError:TooBigDecimalShift
\xintError:TooBigDecimalSplit
\xintError:RootOfNegative
\xintError:NoBezoutForZeros
\xintError:ignored
\xintError:removed
\xintError:inserted
\xintError:use_xintthe!
\xintError:bigtroubleahead
\xintError:unknownfunction
```

19 Common input errors when using the package macros

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

- using - to prefix some macro: `-\xintiSqr{35}/271`.[32]
- using one pair of braces too many `\xintIrr{{\xintiPow {3}{13}}/243}` (the computation goes through with no error signaled, but the result is completely wrong).
- using `[]` and decimal points at the same time `1.5/3.5[2]`, or with a sign in the denominator `3/-5[7]`. The scientific notation has no such restriction, the two inputs `1.5/-3.5e-2` and `-1.5e2/3.5` are equivalent: `\xintRaw{1.5/-3.5e-2}=-15/35[2]`, `\xintRaw{-1.5e2/3.5}=-15/35[2]`.
- specifying numerators and denominators with macros producing fractions when **xintfrac** is loaded: `\edef\x{\xintMul {3}{5}/\xintMul{7}{9}}`. This expands to `15/1[0]/63/1[0]` which is invalid on input. Using this `\x` in a fraction macro will most certainly cause a compilation error, with its usual arcane and undecipherable accompanying message. The fix here would be to use `\xintiMul`. The simpler alternative with package **xintexpr**: `\xinttheexpr 3*5/(7*9)\relax`.
- generally speaking, using in a context expecting an integer (possibly restricted to the TEX bound) a macro or expression which returns a fraction: `\xinttheexpr 4/2\relax` outputs `4/2[0]`, not 2. Use `\xintNum {\xinttheexpr 4/2\relax}` or `\xintthenumexpr 4/2\relax`.

20 Package namespace

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

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

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

public commands all start with \xint. Some other control sequences are used only as delimiters, and left undefined, they may have been defined elsewhere, their meaning doesn't matter and is not touched.

21 Loading and usage

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

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

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

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

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

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

This is for the loading of the packages.

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

All of those requirements are relaxed for tokens parsed inside an \xintexpr-ession (but arguments to macros which have been inserted in the expression must obey the rules, as it is the macro and not the parser which will get the tokens). In an \xintexpr-ession, the 'e' may be uppercased: 'E'.

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

22 Installation

```
Installation
============

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

obtain xint.tds.zip from CTAN:
  http://www.ctan.org/tex-archive/install/macros/generic/xint.tds.zip

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

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

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

obtain xint.dtx, xint.ins and the README from CTAN:
  http://www.ctan.org/tex-archive/macros/generic/xint

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

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

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

- with latex+dvipdfmx: latex xint.tex thrice then dvipdfmx xint.dvi
Ignore dvipdfmx warnings. In case the pdf file has problems with
fonts, use then rather pdflatex or xelatex.

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

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

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

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

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

```
      xintseries.sty |
       xintcfrac.sty |

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

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

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

23 The \xintexpr math parser (I)

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

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

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

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

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

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

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

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

• if such a formula is used thousands of times in a document (for plots?), this could impact some parts of the TeX program memory (for technical reasons explained in section 28). So, a utility `\xintNewExpr` is provided to do the work of translating an `\xintexpr`-ession with parameters into a chain of macro evaluations.[34]

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

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

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

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

This does the same thing as the hand-written version from the previous item. The use even thousands of times of such an \xintNewExpr-generated \formula has no memory impact.

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

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

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

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

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

• there is also \xintboolexpr ... \relax and \xinttheboolexpr ... \relax. Same as regular expression but the final result is converted to 1 if it is not zero. See also \xintifboolexpr (subsection 28.9) and the discussion of the bool and togl functions in section 23. Here is an example:

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

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

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

the precision. Default: 16 digits.

```
\xintthefloatexpr 2^100000\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:
1414213562373095048801688724209698078569671875376948073176 68[-59]
```

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

```
\xintFloat[60]{\xinttheexpr sqrt(2,60)\relax}:
1.414213562373095048801688724209698078569671875376948073176 68e0
```

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

```
\xintDigits:=48; \xintthefloatexpr 2^100000\relax:
9.990020930143845079440327643300335909804291390 54e30102
```

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

24 The \xintexpr math parser (II)

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

Note that 2^-10 is perfectly accepted input, no need for parentheses. And -2^-10^-5*3 does (-((2^(-10))^(-5)))*3.

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

The A/B[N] notation is the output format of most **xintfrac** macros,[35] but for user input in an \xintexpr..\relax such a fraction should be written with the scientific notation AeN/B (possibly within parentheses) or *braces* must be used: {A/B[N]}. The square brackets are *not parsable* if not enclosed in braces together with the fraction.

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

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

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

The minus sign as prefix has various precedence levels: \xintexpr -3-4*-5^-7\relax evaluates as (-3)-(4*(-(5^(-7)))) and -3^-4*-5-7 as (-((3^(-4))*(-5)))-7.

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

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

- Functions are at the same top level of priority.

functions with one (numeric) argument floor, ceil, reduce, sqr, abs, sgn, ?, !, not. The ?(x) function returns the truth value, 1 if x<>0, 0 if x=0. The !(x) is the logical not. The reduce function puts the fraction in irreducible form.

functions with one named argument bool,togl.

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

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

will return *NO* if executed in math mode (the computation is then $100 - 100 = 0$) and YES if not (the if conditional is described below; the \xintifboolexpr test automatically encapsulates its first argument in an \xintexpr and follows the first branch if the result is non-zero (see subsection 28.9)).

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

Similarly togl(name) returns 1 if the LATEX package etoolbox[36] has been used to define a toggle named name, and this toggle is currently set to true. Using togl in an \xintexpr..\relax without having loaded etoolbox will result in an error from \iftoggle being a non-defined macro. If etoolbox is loaded but togl is used on a name not recognized by etoolbox the error message will be of the type "ERROR: Missing \endcsname inserted.", with further information saying that \protect should have not been encountered (this \protect comes from the expansion of the non-expandable etoolbox error message).

When bool or togl is encountered by the \xintexpr parser, the argument enclosed in a parenthesis pair is expanded as usual from left to right, token by token, until the closing parenthesis is found, but everything is taken literally, no computations are performed. For example togl(2+3) will test the value of a toggle declared to etoolbox with name 2+3,

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

and not 5. Spaces are gobbled in this process. It is impossible to use togl on such names containing spaces, but \iftoggle{name with spaces}{1}{0} will work, naturally, as its expansion will pre-empt the \xintexpr scanner.

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

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

See also \xintifboolexpr, in this context.

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

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

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

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

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

- The three postfix operators:

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

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

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

is legal and computes 5+62^3=238333. Note though that it would be better practice to include here the 2^3 inside the branches. The contents of the branches may be arbitrary as long as once glued to what is next the syntax is respected: \xintexpr (3>2)?{5+(6}

{7-(1}2^3)\relax also works. Differs thus from the if conditional in two ways: the false branch is not at all computed, and the number scanner is still active on exit, more digits may follow.

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

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

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

25 Commands of the xinttools package

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

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

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

Contents

25.1 \xintReverseOrder

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

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

25.2 \xintRevWithBraces

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

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

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

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

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

25.3 \xintLength

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

25.5 \xintCSVtoList

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

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

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

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

Contiguous spaces and tab characters, are collapsed by TEX into single spaces. All such spaces around commas[37] are removed, as well as the spaces at the end of the list.[38] 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;

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

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

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.

f ⋆
n ⋆

25.6 \xintNthElt

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

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

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

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

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

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

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

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

In cases where x is larger (in absolute value) than the length of the list then \xintNthElt

returns nothing.

25.7 \xintListWithSep

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

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

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

25.8 \xintApply

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

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

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

```
\def\macro #1{\the\numexpr 9-#1\relax}
\xintApply\macro{\xintFac {20}}=7560797991823359999
```

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.

25.9 \xintApplyUnbraced

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

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

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

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

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

25.10 \xintSeq

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

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

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

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

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

The produced integers are with explicit litteral digits, so if used in `\ifnum` or other tests they should be properly terminated[39].

25.11 Completely expandable prime test

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

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

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

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

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

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

Our integers are short, but without \expandafter's with \@firstoftwo, or some other related techniques, direct use of \ifnum..\fi tests is dangerous. So to make the macro more efficient we are going to use the expandable tests provided by the package etoolbox[40]. The macro becomes:

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

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

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

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

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

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

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

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

[40] http://ctan.org/pkg/etoolbox

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 25.14) which is still expandable and another one (subsection 25.21) which is a close variant of the \IsPrime code above but with the \xintFor loop, thus breaking expandability. The xintiloop variant does not first evaluate the integer square root, the xintFor variant still does. I did not compare their efficiencies. Hmm, if one really needs to compute primes fast, sure I do applaud using **xint**, but, well, there is some slight overhead in using TEX for these things (something like a factor 1000? not tested...) compared to accessing to the CPU ressources via standard compiled code from a standard programming language...

funny private joke

Let us construct with this expandable primality test a table of the prime numbers up to 1000. We need to count how many we have in order to know how many tab stops one shoud add in the last row.[41] There is some subtlety for this last row. Turns out to be better to insert a \\ only when we know for sure we are starting a new row; this is how we have designed the \OneCell macro. And for the last row, there are many ways, we use again \xintApplyUnbraced but with a macro which gobbles its argument and replaces it with a tabulation character. The \xintFor* macro would be more elegant here.

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

We had to be careful to use 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).

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

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

There are 168 prime numbers up to 1000.

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

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

As this loop and `\xintiloop` will primarily be of interest to experienced TEX macro programmers, my description will assume that the user is knowledgeable enough. The iterated commands may contain `\par` tokens or empty lines.

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

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

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

nesting capabilities of \xintloop.[42]

```
\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...
\makeatother
\MatrixMultiplication\A\B\C \MatrixMultiplication\C\C\D % etc...
\[\begin{pmatrix}
    \A11&\A12&\A13&\A14\\
    \A21&\A22&\A23&\A24\\
```

[42] 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.

```
    \A31&\A32&\A33&\A34
  \end{pmatrix}
\times
  \begin{pmatrix}
    \B11&\B12&\B13\\
    \B21&\B22&\B23\\
    \B31&\B32&\B33\\
    \B41&\B42&\B43
  \end{pmatrix}
=
\begin{pmatrix}
    \C11&\C12&\C13\\
    \C21&\C22&\C23\\
    \C31&\C32&\C33
\end{pmatrix}\]
\[\begin{pmatrix}
    \C11&\C12&\C13\\
    \C21&\C22&\C23\\
    \C31&\C32&\C33
\end{pmatrix}^2 = \begin{pmatrix}
    \D11&\D12&\D13\\
    \D21&\D22&\D23\\
    \D31&\D32&\D33
\end{pmatrix}\]
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
\edef\z
{\xintiloop [10001+2]%
  {\xintiloop [3+2]%
   \ifnum\xintouteriloopindex<\numexpr\xintiloopindex*\xintiloopindex\relax
          \xintouteriloopindex,
          \expandafter\xintbreakiloop
   \fi
   \ifnum\xintouteriloopindex=\numexpr
        (\xintouteriloopindex/\xintiloopindex)*\xintiloopindex\relax
   \else
   \repeat
  }% no space here
 \ifnum \xintiloopindex < 10999 \repeat }%
```

`\meaning\z` macro:->10007, 10009, 10037, 10039, 10061, 10067, 10069, 10079, 10091, 10093, 10099, 10103, 10111, 10133, 10139, 10141, 10151, 10159, 10163, 10169, 10177, 10181, 10193, 10211, 10223, 10243, 10247, 10253, 10259, 10267, 10271, 10273, 10289, 10301, 10303, 10313, 10321, 10331, 10333, 10337, 10343, 10357, 10369, 10391, 10399, 10427, 10429, 10433, 10453, 10457, 10459, 10463, 10477, 10487, 10499, 10501, 10513, 10529, 10531, 10559, 10567, 10589, 10597, 10601, 10607, 10613, 10627, 10631, 10639, 10651, 10657, 10663, 10667, 10687, 10691, 10709, 10711, 10723, 10729, 10733, 10739,

10753, 10771, 10781, 10789, 10799, 10831, 10837, 10847, 10853, 10859, 10861, 10867, 10883, 10889, 10891, 10903, 10909, 10937, 10939, 10949, 10957, 10973, 10979, 10987, 10993, and we should have taken some steps to not have a trailing comma, but the point was to show that one can do that in an \edef! See also subsection 25.14 which extracts from this code its way of testing primality.

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

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

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

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

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

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

25.14 Another completely expandable prime test

The \IsPrime macro from subsection 25.11 checked expandably if a (short) integer was prime, here is a partial rewrite using \xintiloop. We use the etoolbox expandable conditionals for convenience, but not everywhere as \xintiloopindex can not be evaluated while being braced. This is also the reason why \xintbreakiloopanddo is delimited, and the next macro \SmallestFactor which returns the smallest prime factor examplifies that. One could write more efficient completely expandable routines, the aim here was only to illustrate use of the general purpose \xintiloop. A little table giving the first values of \SmallestFactor follows, its coding uses \xintFor, which is described later; none of this uses count registers.

```
\newcommand{\IsPrime}[1] % returns 1 if #1 is prime, and 0 if not
  {\ifnumodd {#1}
    {\ifnumless {#1}{8}
      {\ifnumequal{#1}{1}{0}{1}}% 3,5,7 are primes
      {\if
       \xintiloop [3+2]
       \ifnum#1<\numexpr\xintiloopindex*\xintiloopindex\relax
           \expandafter\xintbreakiloopanddo\expandafter1\expandafter.%
       \fi
       \ifnum#1=\numexpr (#1/\xintiloopindex)*\xintiloopindex\relax
       \else
       \repeat 00\expandafter0\else\expandafter1\fi
      }%
    }% END OF THE ODD BRANCH
    {\ifnumequal {#1}{2}{1}{0}}% EVEN BRANCH
}%
\catcode`_ 11
\newcommand{\SmallestFactor}[1] % returns the smallest prime factor of #1>1
  {\ifnumodd {#1}
    {\ifnumless {#1}{8}
      {#1}% 3,5,7 are primes
      {\xintiloop [3+2]
       \ifnum#1<\numexpr\xintiloopindex*\xintiloopindex\relax
           \xint_afterfi{\xintbreakiloopanddo#1.}%
       \fi
       \ifnum#1=\numexpr (#1/\xintiloopindex)*\xintiloopindex\relax
           \xint_afterfi{\expandafter\xintbreakiloopanddo\xintiloopindex.}%
       \fi
       \iftrue\repeat
      }%
     }% END OF THE ODD BRANCH
   {2}% EVEN BRANCH
}%
\catcode`_ 8
  \begin{tabular}{|c|*{10}c|}
    \hline
    \xintFor #1 in {0,1,2,3,4,5,6,7,8,9}\do {&\bfseries #1}\\
    \hline
    \bfseries 0&--&--&2&3&2&5&2&7&2&3\\
    \xintFor #1 in {1,2,3,4,5,6,7,8,9}\do
```

```
    {\bfseries #1%
      \xintFor #2 in {0,1,2,3,4,5,6,7,8,9}\do
      {&\SmallestFactor{#1#2}}\\}%
    \hline
  \end{tabular}
```

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

25.15 A table of factorizations

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

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

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

The table has been made into a float which appears on page 48. Here is now the code for factorization; the conditionals use the package provided \xint_firstoftwo and \xint_secondoftwo, one could have employed rather LaTeX's own \@firstoftwo and \@secondoftwo, or, simpler still in LaTeX context, the \ifnumequal, \ifnumless ..., utilities from the package etoolbox which do exactly that under the hood. Only TeX ac-

ceptable numbers are treated here, but it would be easy to make a translation and use the xint macros, thus extending the scope to big numbers; naturally up to a cost in speed.

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

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

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

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

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

25.16 \xintApplyInline

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

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

```
\def\Macro #1{\advance\cnta #1 , \the\cnta}
\cnta 0
0\xintApplyInline\Macro {3141592653}.
```

Output: 0, 3, 4, 8, 9, 14, 23, 25, 31, 36, 39.

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

`\xintApplyInline` submits its second, token list parameter to an *f*-expansion. Then, each *unbraced* item will also be *f*-expanded. This provides an easy way to insert one list

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

inside another. *Braced* items are not expanded. Spaces in-between items are gobbled (as well as those at the start or the end of the list), but not the spaces *inside* the braced items.

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

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

was obtained from the following input:

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

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

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

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

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

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

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

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

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

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

But see \xintFor.

25.17 \xintFor, \xintFor*

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

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

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

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

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

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

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

> The non-starred variant \xintFor deals with comma separated values (*spaces before and after the commas are removed*) and the comma separated list may be a macro which is only expanded once (to prevent expansion of the first item \x in a list directly

input as \x,\y,... it should be input as {\x},\y,.. or <space>\x,\y,.., naturally all of that within the mandatory braces of the \xintFor #n in {list} syntax). The items are not expanded, if the input is <stuff>,\x,<stuff> then #1 will be at some point \x not its expansion (and not either a macro with \x as replacement text, just the token \x). Input such as <stuff>,,<stuff> creates an empty #1, the iteration is not skipped. An empty list does lead to the use of the replacement text, once, with an empty #1 (or #n). Except if the entire list is represented as a single macro with no parameters, it must be braced.

The starred variant \xintFor* deals with token lists (*spaces between braced items or single tokens are not significant*) and *f*-expands each *unbraced* list item. This makes it easy to simulate concatenation of various list macros \x, \y, ... If \x expands to {1}{2}{3} and \y expands to {4}{5}{6} then {\x\y} as argument to \xintFor* has the same effect as {{1}{2}{3}{4}{5}{6}}[43]. Spaces at the start, end, or in-between items are gobbled (but naturally not the spaces which may be inside *braced* items). Except if the list argument is a single macro with no parameters, it must be braced. Each item which is not braced will be fully expanded (as the \x and \y in the example above). An empty list leads to an empty result.

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

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

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

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

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

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

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

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

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

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

When inserted inside a macro for later execution the # characters must be doubled.[43] For example:

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

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

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

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

The iterated commands as well as the list items are allowed to contain explicit `\par` tokens. Neither `\xintFor` nor `\xintFor*` create groups. The effect is like piling up the iterated commands with each time `#1` (or `#2` ...) replaced by an item of the list. However, contrarily to the completely expandable `\xintApplyUnbraced`, but similarly to the non completely expandable `\xintApplyInline` each iteration is executed first before looking at the next `#1`[44] (and the starred variant `\xintFor*` keeps on expanding each unbraced item it finds, gobbling spaces).

25.18 \xintifForFirst, \xintifForLast

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

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

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

[43] 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.

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

25.19 \xintBreakFor, \xintBreakForAndDo

One may immediately terminate an \xintFor or \xintFor* loop with \xintBreakFor. As the criterion for breaking will be decided on a basis of some test, it is recommended to use for this test the syntax of ifthen[45] or etoolbox[46] or the xint own conditionals, rather than one of the various \if...\fi of TEX. Else (and this is without even mentioning all the various pecularities of the \if...\fi constructs), one has to carefully move the break after the closing of the conditional, typically with \expandafter\xintBreakFor\fi.[47]

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

25.20 \xintintegers, \xintdimensions, \xintrationals

If the list argument to \xintFor (or \xintFor*, both are equivalent in this context) is \xintintegers (equivalently \xintegers) or more generally \xintintegers[start+delta] (*the whole within braces*!)[48], then \xintFor does an infinite iteration where #1 (or #2, ..., #9) will run through the arithmetic sequence of (short) integers with initial value start and increment delta (default values: start=1, delta=1; if the optional argument is present it must contains both of them, and they may be explicit integers, or macros or count registers). The #1 (or #2, ..., #9) will stand for \numexpr <opt sign><digits>\relax, and the litteral representation as a string of digits can thus be obtained as \the#1 or \number#1. Such a #1 can be used in an \ifnum test with no need to be postfixed with a space or a \relax and one should *not* add them.

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

The graphic, with the code on its right[49], is for illustration only, not only because of pdf

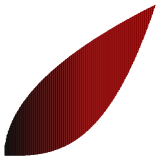
[45] http://ctan.org/pkg/ifthen

[46] http://ctan.org/pkg/etoolbox

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

[48] 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.

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



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

rendering artefacts when displaying adjacent rules (which do *not* show in dvi output as rendered by xdvi, and depend from your viewer), but because not using anything but rules it is quite inefficient and must do lots of computations to not confer a too ragged look to the borders. With a width of .5pt rather than .1pt for the rules, one speeds up the drawing by a factor of five, but the boundary is then visibly ragged. [50]

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

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

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

[50] to tell the whole truth we cheated and divided by 10 the computation time through using the following definitions, together with a horizontal step of .25pt rather than .1pt. The displayed original code would make the slowest computation of all those done in this document using the **xint** bundle macros!

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

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

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

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

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

25.21 Another table of primes

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

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

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

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

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

25.22 \xintForpair, \xintForthree, \xintForfour

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

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

```
\end{tabular}
```

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

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

25.23 \xintAssign

(f→∗x) ∗N `\xintAssign`⟨*braced things*⟩`\to`⟨*as many cs as they are things*⟩ defines (without checking if something gets overwritten) the control sequences on the right of `\to` to be the complete expansions of the successive braced things found on the left of `\to`. It is not expandable.

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

Special case: if after this initial expansion no brace is found immediately after `\xintAssign`, it is assumed that there is only one control sequence following `\to`, and this control *x N* sequence is then defined via `\edef` as the complete expansion of the material between `\xintAssign` and `\to`.

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

25.24 \xintAssignArray

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

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

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

25.25 \xintRelaxArray

N `\xintRelaxArray\myArray` (globally) sets to `\relax` all macros which were defined by

the previous \xintAssignArray with \myArray as array macro.

25.26 The Quick Sort algorithm illustrated

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

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

```
% THE QUICK SORT ALGORITHM EXPANDABLY
\input xintfrac.sty
% HELPER COMPARISON MACROS
\def\QSMore  #1#2{\xintifGt {#2}{#1}{{#2}}{ }}
% the spaces are there to stop the \romannumeral-‘0 originating
% in \xintapplyunbraced when it applies a macro to an item
\def\QSEqual #1#2{\xintifEq {#2}{#1}{{#2}}{ }}
\def\QSLess  #1#2{\xintifLt {#2}{#1}{{#2}}{ }}
%
\makeatletter
\def\QSfull   {\romannumeral0\qsfull }
\def\qsfull   #1{\expandafter\qsfull@a\expandafter{\romannumeral-‘0#1}}
\def\qsfull@a #1{\expandafter\qsfull@b\expandafter {\xintLength {#1}}{#1}}
\def\qsfull@b #1{\ifcase #1
                    \expandafter\qsfull@empty
                \or\expandafter\qsfull@single
                \else\expandafter\qsfull@c
                \fi
}%
\def\qsfull@empty  #1{ }  % the space stops the \QSfull \romannumeral0
\def\qsfull@single #1{ #1}
% for simplicity of implementation, we pick up the first item as pivot
\def\qsfull@c #1{\qsfull@ci #1\undef {#1}}
\def\qsfull@ci #1#2\undef {\qsfull@d {#1}}% #3 is the list, #1 its first item
\def\qsfull@d #1#2{\expandafter\qsfull@e\expandafter
                  {\romannumeral0\qsfull
                      {\xintApplyUnbraced {\QSMore {#1}}{#2}}}%
                  {\romannumeral0\xintapplyunbraced {\QSEqual {#1}}{#2}}%
                  {\romannumeral0\qsfull
                      {\xintApplyUnbraced {\QSLess {#1}}{#2}}}%
}%
\def\qsfull@e #1#2#3{\expandafter\qsfull@f\expandafter {#2}{#3}{#1}}%
\def\qsfull@f #1#2#3{\expandafter\space #2#1#3}
\makeatother
% EXAMPLE
\edef\z {\QSfull {{1.0}{0.5}{0.3}{1.5}{1.8}{2.0}{1.7}{0.4}{1.2}{1.4}%
```

```
              {1.3}{1.1}{0.7}{1.6}{0.6}{0.9}{0.8}{0.2}{0.1}{1.9}}}
\tt\meaning\z
\def\a {3.12345789123456789}\def\b {3.123456789123456788}
\def\c {3.12345789123456790}\def\d {3.123456789123456787}
\expandafter\def\expandafter\z\expandafter
  {\romannumeral0\qsfull {{\a}\b\c\d}}% \a is braced to not be expanded
\meaning\z
```

Output:

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

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

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

```
\def\DecoPivot #1{\begingroup\color{PIVOT}\advance\fboxsep-\fboxrule
                              \fbox{#1}\endgroup}
\def\DecoLEFTwithPivot #1{%
     \xintFor* ##1 in {#1} \do
     {\xintifForFirst {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}%
}
\def\DecoRIGHTwithPivot #1{%
     \xintFor* ##1 in {#1} \do
     {\xintifForFirst {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}%
}
%
\def\QSinitialize #1{\def\QS@list{\QSRr {#1}}%
                     \let\QSRr\DecoRIGHT
%                     \QS@list \par
\par\centerline{\QS@list}
}
\def\QSoneStep {\let\QSLr\DecoLEFTwithPivot
                \let\QSIr\DecoINERT
                \let\QSRr\DecoRIGHTwithPivot
%                    \QS@list
\centerline{\QS@list}
%                \par
                \def\QSLr {\noexpand\QS@a}%
                \let\QSIr\relax
                \def\QSRr {\noexpand\QS@a}%
                    \edef\QS@list{\QS@list}%
                \let\QSLr\relax
                \let\QSRr\relax
                    \edef\QS@list{\QS@list}%
                \let\QSLr\DecoLEFT
                \let\QSIr\DecoINERT
                \let\QSRr\DecoRIGHT
%                    \QS@list
\centerline{\QS@list}
%                \par
}
\begingroup\offinterlineskip
\small
\QSinitialize {{1.0}{0.5}{0.3}{1.5}{1.8}{2.0}{1.7}{0.4}{1.2}{1.4}%
               {1.3}{1.1}{0.7}{1.6}{0.6}{0.9}{0.8}{0.2}{0.1}{1.9}}
\QSoneStep
\QSoneStep
\QSoneStep
\QSoneStep
\QSoneStep
\endgroup
```

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

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

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

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

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

It is possible to modify this code to let it do \QSonestep repeatedly and stop automatically when the sort is finished.[51]

[51] http://tex.stackexchange.com/a/142634/4686

26 Commands of the xint package

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

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

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

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

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

Contents

26.1 \xintRev

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

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

26.2 \xintLen

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

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

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

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

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

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

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

26.3 \xintDigitsOf

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

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

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

26.4 \xintNum

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

```
\xintNum{+---++----+--0000000003679417894 79}=-367941789479
```

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

```
\xintNum{123.48/-0.03}=-4116
```

26.5 \xintSgn

Num *f* ⋆ `\xintSgn{N}` returns 1 if the number is positive, 0 if it is zero and -1 if it is negative.

f ⋆ Extended by **xintfrac** to fractions. `\xintiiSgn` skips the `\xintNum` overhead.

26.6 \xintOpp

Num *f* ⋆ `\xintOpp{N}` return the opposite `-N` of the number `N`. Extended by **xintfrac** to fractions. `\xintiOpp` is a synonym not modified by **xintfrac**[52], and `\xintiiOpp` skips the `\xintNum` overhead.

f ⋆

26.7 \xintAbs

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

f ⋆

26.8 \xintAdd

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

ff ⋆

26.9 \xintSub

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

ff ⋆

26.10 \xintCmp

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

26.11 \xintEq

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

26.12 \xintGt

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

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

26.13 \xintLt

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

26.14 \xintIsZero

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

26.15 \xintNot

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

26.16 \xintIsNotZero

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

26.17 \xintIsOne

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

26.18 \xintAND

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

26.19 \xintOR

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

26.20 \xintXOR

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

26.21 \xintANDof

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

26.22 \xintORof

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

26.23 \xintXORof

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

26.24 \xintGeq

Num Num
f f ⋆

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

26.25 \xintMax

Num Num
f f ⋆

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

26.26 \xintMaxof

$f\to *$ Num
f ⋆

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

26.27 \xintMin

Num Num
f f ⋆

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

26.28 \xintMinof

$f\to *$ Num
f ⋆

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

26.29 \xintSum

$*f$ ⋆

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

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

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

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

26.30 \xintMul

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

26.31 \xintSqr

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

26.32 \xintPrd

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

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

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

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

```
=\xintiiPrd {{\xintiPow {2}{200}}{\xintiPow {3}{100}}{\xintiPow {7}{100}}}
=267872793166157757576627951700754840232474026637401534897445961481542641296549949000044400724076572713000016531207640654562118014357199401590334353924402821243896682224892786298808438271613337 6
```

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

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

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

26.33 \xintPow

Num num
f x ⋆ `\xintPow{N}{x}` returns `N^x`. When `x` is zero, this is 1. If `N` is zero and `x<0`, if `|N|>1` and `x<0` negative, or if `|N|>1` and `x>999999999`, then an error is raised. `2^999999999` has 301,029,996 digits; each exact multiplication of two one thousand digits numbers already takes a few seconds, so needless to say this bound is completely irrealistic. Already `2^9999` has 3,010 digits,[53] so I should perhaps lower the bound to `99999`.

Extended by **xintfrac** to fractions (`\xintPow`) and to floats (`\xintFloatPow`). Negative exponents do not then cause errors anymore. The float version is able to deal with

[53] on my laptop `\xintiPow{2}{9999}` obtains all `3010` digits in about ten or eleven seconds. In contrast, the float versions for `8`, `16`, `24`, or even more significant figures, do their jobs in circa one hundredth of a second (`1.08b`). This is done without `log/exp` which are not (yet?) implemented in **xintfrac**. The LaTeX3 l3fp does this with `log/exp` and is ten times faster (`16` figures only).

things such as `2^999999999` without any problem. For example `\xintFloatPow[4]{2}{9999}=9.975e3009` and `\xintFloatPow[4]{2}{999999999}=2.306e301029995`.

`\xintiPow` is a synonym not modified by **xintfrac**, and `\xintiiPow` is an integer only variant skipping the `\xintNum` overhead.

f^num x ⋆

26.34 \xintSgnFork

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

26.35 \xintifSgn

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

26.36 \xintifZero

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

26.37 \xintifNotZero

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

26.38 \xintifTrueFalse

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

26.39 \xintifCmp

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

26.40 \xintifEq

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

26.41 \xintifGt

Num Num
f f nn ⋆

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

26.42 \xintifLt

Num Num
f f nn ⋆

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

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

26.43 \xintifOdd

Num
f nn ⋆

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

26.44 \xintFac

num
x ⋆

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

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

26.45 \xintDivision

Num Num
f f ⋆

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

ff ⋆

(even if `N` vanishes) and returns `{0}{0}`. The variant `\xintiiDivision` skips the overhead of parsing via `\xintNum`.

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

26.46 \xintQuo

Num Num
f f ⋆

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

ff ⋆

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

26.47 \xintRem

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

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

26.48 \xintFDg

Num
f ⋆ `\xintFDg{N}` returns the first digit (most significant) of the decimal expansion. The variant

f ⋆ `\xintiiFDg` skips the overhead of parsing via `\xintNum`.

26.49 \xintLDg

Num
f ⋆ `\xintLDg{N}` returns the least significant digit. When the number is positive, this is the same as the remainder in the euclidean division by ten. The variant `\xintiiLDg` skips the overhead of parsing via `\xintNum`.

26.50 \xintMON, \xintMMON

Num
f ⋆ `\xintMON{N}` returns `(-1)^N` and `\xintMMON{N}` returns `(-1)^{N-1}`.

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

f ⋆ The variants `\xintiiMON` and `\xintiiMMON` skip the overhead of parsing via `\xintNum`.

26.51 \xintOdd

Num
f ⋆ `\xintOdd{N}` is 1 if the number is odd and 0 otherwise. The variant `\xintiiOdd` skip the

f ⋆ overhead of parsing via `\xintNum`.

26.52 \xintiSqrt, \xintiSquareRoot

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

```
\xintiSqrt {200000000000000000000000000000000000}=1414213562373095048
\xintiSqrt {300000000000000000000000000000000000}=1732050807568877293
                  \xintiSqrt {\xintDSH {-120}{2}}=
   1414213562373095048801688724209698078569671875376948073176679
```

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

```
    \xintAssign\xintiSquareRoot {17000000000000000000000000}\to\A\B
                  \xintiSub{\xintiSqr\A}\B=\A^2-\B
      17000000000000000000000000=4123105625618^2-2799177881924
```

A rational approximation to $\sqrt{N}$ is $M - \frac{d}{2M}$ (this is a majorant and the error is at most `1/2M`; if `N` is a perfect square `k^2` then `M=k+1` and this gives `k+1/(2k+2)`, not `k`).

Package **xintfrac** has `\xintFloatSqrt` for square roots of floating point numbers.

> The macros described next are strictly for integer-only arguments. These arguments are *not* filtered via `\xintNum`.

26.53 \xintInc, \xintDec

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

26.54 \xintDouble, \xintHalf

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

26.55 \xintDSL

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

26.56 \xintDSR

$f\star$ `\xintDSR{N}` is decimal shift right, *i.e.* it removes the last digit (keeping the sign), equivalently it is the closest integer to `N/10` when starting at zero.

26.57 \xintDSH

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

26.58 \xintDSHr, \xintDSx

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

- if `N` is positive or zero, `Q` and `R` are the quotient and remainder in the euclidean division by `10^x` (obtained in a more efficient manner than using `\xintDivision`),
- if `N` is negative let `Q1` and `R1` be the quotient and remainder in the euclidean division by `10^x` of the absolute value of `N`. If `Q1` does not vanish, then `Q=-Q1` and `R=R1`. If `Q1` vanishes, then `Q=0` and `R=-R1`.
- for `x=0`, `Q=N` and `R=0`.

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

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

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

```
\meaning\M: macro:->12345676890000000000000000000.
\xintAssign\xintDSx {0}{-123004321}\to\Q\R
\meaning\Q: macro:->-123004321, \meaning\R: macro:->0.
\xintDSH {0}{-123004321}=-123004321, \xintDSHr {0}{-123004321}=0
\xintAssign\xintDSx {6}{-123004321}\to\Q\R
\meaning\Q: macro:->-123, \meaning\R: macro:->4321.
\xintDSH {6}{-123004321}=-123, \xintDSHr {6}{-123004321}=4321
\xintAssign\xintDSx {8}{-123004321}\to\Q\R
\meaning\Q: macro:->-1, \meaning\R: macro:->23004321.
\xintDSH {8}{-123004321}=-1, \xintDSHr {8}{-123004321}=23004321
\xintAssign\xintDSx {9}{-123004321}\to\Q\R
\meaning\Q: macro:->0, \meaning\R: macro:->-123004321.
\xintDSH {9}{-123004321}=0, \xintDSHr {9}{-123004321}=-123004321
```

26.59 \xintDecSplit

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

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

```
             \xintAssign\xintDecSplit {0}{-123004321}\to\L\R
\meaning\L: macro:->123004321, \meaning\R: macro:->.
             \xintAssign\xintDecSplit {5}{-123004321}\to\L\R
\meaning\L: macro:->1230, \meaning\R: macro:->04321.
             \xintAssign\xintDecSplit {9}{-123004321}\to\L\R
\meaning\L: macro:->, \meaning\R: macro:->123004321.
            \xintAssign\xintDecSplit {10}{-123004321}\to\L\R
\meaning\L: macro:->, \meaning\R: macro:->123004321.
           \xintAssign\xintDecSplit {-5}{-12300004321}\to\L\R
\meaning\L: macro:->12300, \meaning\R: macro:->004321.
          \xintAssign\xintDecSplit {-11}{-12300004321}\to\L\R
\meaning\L: macro:->12300004321, \meaning\R: macro:->.
          \xintAssign\xintDecSplit {-15}{-12300004321}\to\L\R
\meaning\L: macro:->12300004321, \meaning\R: macro:->.
```

26.60 \xintDecSplitL

num x f ⋆ `\xintDecSplitL{x}{N}` returns the first piece after the action of `\xintDecSplit`.

26.61 \xintDecSplitR

num x f ⋆ `\xintDecSplitR{x}{N}` returns the second piece after the action of `\xintDecSplit`.

27 Commands of the xintfrac package

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

Frac *f*

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

num *x*

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

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

To be certain to print an integer output without trailing [n] nor fraction slash, one should use either \xintPRaw {\xintIrr {f}} or \xintNum {f} when it is already known that f evaluates to a (big) integer. For example \xintPRaw {\xintAdd {2/5}{3/5}} gives a perhaps disappointing 25/25[54], whereas \xintPRaw {\xintIrr {\xintAdd {2/5}{3/5}}} returns 1. As we knew the result was an integer we could have used \xintNum {\xintAdd {2/5}{3/5}}=1.

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

Contents

[54] yes, \xintAdd blindly multiplies denominators...

27.1 \xintNum

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

27.2 \xintifInt

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

27.3 \xintLen

Frac *f* ⋆ The original macro is extended to accept a fraction on input.

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

27.4 \xintRaw

Frac *f* ⋆ This macro 'prints' the fraction f as it is received by the package after its parsing and expansion, in a form A/B[n] equivalent to the internal representation: the denominator B is always strictly positive and is printed even if it has value 1.

```
\xintRaw{\the\numexpr 571*987\relax.123e-10/\the\numexpr -201+59\relax e-7}=
                                -563577123/142[-6]
```

27.5 \xintPRaw

Frac *f* ⋆ PRaw stands for "pretty raw". It does *not* show the [n] if n=0 and does *not* show the B if B=1.

```
\xintPRaw {123e10/321e10}=123/321, \xintPRaw {123e9/321e10}=123/321[-1]
       \xintPRaw {\xintIrr{861/123}}=7 vz. \xintIrr{861/123}=7/1
```

See also \xintFrac (or \xintFwOver) for math mode. As is examplified above the \xintIrr macro which puts the fraction into irreducible form does not remove the /1 if the

fraction is an integer. One can use \xintNum for that, but there will be an error message if the fraction was not an integer; so the combination \xintPRaw{\xintIrr{f}} is the way to go.

27.6 \xintNumerator

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

```
\xintNumerator {178000/25600000[17]}=178000000000000000000
\xintNumerator {312.289001/20198.27}=312289001
\xintNumerator {178000e-3/256e5}=178000
\xintNumerator {178.000/25600000}=178000
```

As shown by the examples, no simplification of the input is done. For a result uniquely associated to the value of the fraction first apply \xintIrr.

27.7 \xintDenominator

Frac *f* ⋆ This returns the denominator corresponding to the internal representation of the fraction:[55]

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

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

27.8 \xintRawWithZeros

Frac *f* ⋆ This macro 'prints' the fraction f (after its parsing and expansion) in A/B form, with A as returned by \xintNumerator{f} and B as returned by \xintDenominator{f}.

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

27.9 \xintREZ

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

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

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

27.10 \xintFrac

Frac *f* ⋆ This is a LaTeX only command, to be used in math mode only. It will print a fraction, internally represented as something equivalent to A/B[n] as \frac {A}{B}10^n.

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

The power of ten is omitted when `n=0`, the denominator is omitted when it has value one, the number being separated from the power of ten by a `\cdot`. `$\xintFrac {178.000/25600000}$` gives $\frac{178000}{25600000}10^{-3}$, `$\xintFrac {178.000/1}$` gives $178000 \cdot 10^{-3}$, `$\xintFrac {3.5/5.7}$` gives $\frac{35}{57}$, and `$\xintFrac {\xintNum {\xintFac{10}/\xintiSqr{\xintFac {5}}}}$` gives 252. As shown by the examples, simplification of the input (apart from removing the decimal points and moving the minus sign to the numerator) is not done automatically and must be the result of macros such as `\xintIrr`, `\xintREZ`, or `\xintNum` (for fractions being in fact integers.)

27.11 \xintSignedFrac

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

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

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

27.12 \xintFwOver

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

27.13 \xintSignedFwOver

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

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

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

27.14 \xintIrr

Frac
f ⋆ This puts the fraction into its unique irreducible form:

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

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

Starting with release `1.08`, `\xintIrr` does not remove the trailing `/1` when the output is an integer. This was deemed better for various (stupid?) reasons and thus the output format is now *always* `A/B` with `B>0`. Use `\xintPRaw` on top of `\xintIrr` if it is needed to get rid of a possible trailing `/1`. For display in math mode, use rather `\xintFrac{\xintIrr {f}}` or `\xintFwOver{\xintIrr {f}}`.

27.15 \xintJrr

Frac f ⋆ This also puts the fraction into its unique irreducible form:

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

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

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

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

27.16 \xintTrunc

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

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

The digits printed are exact up to and including the last one. The identity \xintTrunc {x}{-f}=-\xintTrunc {x}{f} holds.[56]

27.17 \xintiTrunc

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

```
\xintiTrunc {16}{-803.2028/20905.298}=-3842101652892000
\xintiTrunc {10}{\xintPow {-11}{-11}}=0
\xintiTrunc {12}{\xintPow {-11}{-11}}=-3
```

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

27.18 \xintRound

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

```
\xintRound {16}{-803.2028/20905.298}=-0.0384210165289201
```

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

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

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

```
-0.00000000000350493899481392497604003313162598556370...
```

27.19 \xintiRound

num x Frac f ★

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

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

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

27.20 \xintFloor

Frac f ★

`\xintFloor {f}` returns the largest relative integer `N` with $N \leq f$.

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

27.21 \xintCeil

Frac f ★

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

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

27.22 \xintE

Frac f num x ★

`\xintE {f}{x}` multiplies the fraction `f` by `10^x`. The *second* argument `x` must obey the TEX bounds. Example:

```
\count 255 123456789 \xintE {10}{\count 255}->10/1[123456789]
```

Be careful that for obvious reasons such gigantic numbers should not be given to `\xintNum`, or added to something with a widely different order of magnitude, as the package always works to get the *exact* result. There is *no problem* using them for *float* operations:

```
\xintFloatAdd {1e1234567890}{1}=1.000000000000000e1234567890
```

27.23 \xintDigits, \xinttheDigits

The syntax `\xintDigits := D;` (where spaces do not matter) assigns the value of `D` to the number of digits to be used by floating point operations. The default is `16`. The maximal ★ value is `32767`. The macro `\xinttheDigits` serves to print the current value.

27.24 \xintFloat

[num x] Frac f ★

The macro `\xintFloat [P]{f}` has an optional argument `P` which replaces the current

value of \xintDigits. The (rounded truncation of the) fraction f is then printed in scientific form, with P digits, a lowercase e and an exponent N. The first digit is from 1 to 9, it is preceded by an optional minus sign and is followed by a dot and P-1 digits, the trailing zeros are not trimmed. In the exceptional case where the rounding went to the next power of ten, the output is 10.0...0eN (with a sign, perhaps). The sole exception is for a zero value, which then gets output as 0.e0 (in an \xintCmp test it is the only possible output of \xintFloat or one of the 'Float' macros which will test positive for equality with zero).

```
 \xintFloat[32]{1234567/7654321}=1.6129020457856418616360615134902e-1
\xintFloat[32]{1/\xintFac{100}}=1.07151028812546692318354675951926-158
```

The argument to \xintFloat may be an \xinttheexpr-ession, like the other macros; only its final evaluation is submitted to \xintFloat: the inner evaluations of chained arguments are not at all done in 'floating' mode. For this one must use \xintthefloatexpr.

27.25 \xintAdd

Frac Frac
f f ⋆

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

27.26 \xintFloatAdd

[num x] Frac Frac
f f ⋆

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

27.27 \xintSub

Frac Frac
f f ⋆

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

27.28 \xintFloatSub

[num x] Frac Frac
f f ⋆

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

27.29 \xintMul

Frac Frac
f f ⋆

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

27.30 \xintFloatMul

[num x] Frac Frac
f f ⋆

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

27.31 \xintSqr

Frac
f ⋆

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

27.32 \xintDiv

Frac Frac
f *f* ⋆

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

27.33 \xintFloatDiv

[num *x*] Frac Frac
f *f* ⋆

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

27.34 \xintFac

Num
f ⋆

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

27.35 \xintPow

Frac Num
f *f* ⋆

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

The exponent is allowed to be input as a fraction but it must simplify to an integer: `\xintPow {2/3}{10/2}=32/243[0]`. This integer will be checked to not exceed `999999999`; future releases will presumably lower this limit as even much much smaller values already create gigantic numerators and denominators which can not be computed exactly in a reasonable time. Indeed `2^999999999` has `301029996` digits.

27.36 \xintFloatPow

[num *x*] Frac num
f *x* ⋆

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

The exponent `x` will be fed to a `\numexpr`, hence count registers are accepted on input for this `x`. And the absolute value `|x|` must obey the TEX bound. For larger exponents use the slightly slower routine `\xintFloatPower` which allows the exponent to be a fraction simplifying to an integer and does not limit its size. This slightly slower routine is the one to which `^` is mapped inside `\xintthefloatexpr...\relax`.

The macro `\xintFloatPow` chooses dynamically an appropriate number of digits for the intermediate computations, large enough to achieve the desired accuracy (hopefully).

```
\xintFloatPow [8]{3.1415}{1234567890}=1.6122066e613749456
```

27.37 \xintFloatPower

[num x] Frac f Num f ⋆

`\xintFloatPower[P]{f}{g}` computes a floating point value `f^g` where the exponent `g` is not constrained to be at most the TEX bound `2147483647`. It may even be a fraction `A/B` but must simplify to a (possibly big) integer.

```
\xintFloatPower [8]{1.000000000001}{1e12}=2.7182818e0
\xintFloatPower [8]{3.1415}{3e9}=1.4317729e1491411192
```

Note that `3e9>2^31`. But the number following `e` in the output must at any rate obey the TEX `2147483647` bound.

Inside an `\xintfloatexpr`-ession, `\xintFloatPower` is the function to which `^` is mapped. The exponent may then be something like `(144/3/(1.3-.5)-37)` which is, in disguise, an integer.

The intermediate multiplications are done with a higher precision than `\xintDigits` or the optional `P` argument, in order for the final result to hopefully have the desired accuracy.

27.38 \xintFloatSqrt

[num x] Frac f ⋆

`\xintFloatSqrt[P]{f}` computes a floating point approximation of $\sqrt{f}$, either using the optional precision `P` or the value of `\xintDigits`. The computation is done for a precision of at least 17 figures (and the output is rounded if the asked-for precision was smaller).

```
\xintFloatSqrt [50]{12.3456789e12}
≈ 3.513641828644462161665823116758077037159142718124 3e6
\xintDigits:=50;\xintFloatSqrt {\xintFloatSqrt {2}}
≈ 1.18920711500272106671749997056047591529297209246 38e0
```

27.39 \xintSum

f → ∗ Frac f ⋆

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

27.40 \xintPrd

f → ∗ Frac f ⋆

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

27.41 \xintCmp

Frac f Frac f ⋆

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

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

27.42 \xintIsOne

Frac f ⋆

See `\xintIsOne` (subsection 26.17).

27.43 \xintGeq

Frac Frac
f *f* ⋆

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

27.44 \xintMax

Frac Frac
f *f* ⋆

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

27.45 \xintMaxof

Frac
f → ∗ *f* ⋆

See \xintMaxof (subsection 26.26).

27.46 \xintMin

Frac Frac
f *f* ⋆

The macro is extended to fractions. The original, for (big) integers only, is available as \xintiMin.

27.47 \xintMinof

Frac
f → ∗ *f* ⋆

See \xintMinof (subsection 26.28).

27.48 \xintAbs

Frac
f ⋆

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

27.49 \xintSgn

Frac
f ⋆

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

27.50 \xintOpp

Frac
f ⋆

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

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

Frac Frac
f *f* ⋆
or Frac
f ⋆

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

All have variants whose names start with `xintii` rather than `xint`; these variants accept on input only integers in the strict format (they do not use \xintNum). They thus have

less overhead, and may be used when one is dealing exclusively with (big) integers. These variants are already available in xint, there is no need for xintfrac to be loaded.

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

```
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

28 Expandable expressions with the xintexpr package

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

The syntax is described in section 23 and section 24.

Contents

28.1 The \xintexpr expressions

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

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

1. it is allowed to occur only at the spots where numbers are legal,
2. the ⟨*expandable*⟩ contents is then completely expanded as if it were put in a \csname..\endcsname,[57] thus it escapes entirely the parser scope and infix notations will not be recognized except if the expanded macros know how to handle them by themselves,
3. and this complete expansion *must* produce a number or a fraction, possibly with decimal mark and trailing [n], the scientific notation is *not* authorized.

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

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

[57] well, actually it *is* put in such a \csname..\endcsname.

x ★ • \xinttheexpr⟨*expandable_expression*⟩\relax, or

x ★ • \xintthe\xintexpr⟨*expandable_expression*⟩\relax.

The computations are done *exactly*, and with no simplification of the result. The output format for the result can be coded inside the expression through the use of one of the functions round, trunc, float, reduce.[58] Here are some examples

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

- the expression may contain arbitrarily many levels of nested parenthesized sub-expressions,
- sub-contents giving numbers of fractions should be either
  1. parenthesized,
  2. a sub-expression \xintexpr...\relax,
  3. or within braces.
- an expression can not be given as argument to the other package macros, nor printed, for this one must use \xinttheexpr...\relax or \xintthe\xintexpr...\relax,
- one does not use \xinttheexpr...\relax as a sub-constituent of an \xintexpr...\relax (or \xinttheexpr...\relax) but simply \xintexpr...\relax,
- each xintexpression is completely expandable and obtains its result in two expansion steps.

28.2 \numexpr expressions, count and dimension registers

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

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

28.3 Catcodes and spaces

28.3.1 \xintexprSafeCatcodes

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

[58] In round and trunc the second optional parameter is the number of digits of the fractional part; in float it is the total number of digits of the mantissa.

28.3.2 \xintexprRestoreCatcodes

Restores the catcodes to the earlier state.

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

\xintexpr and \xintthexpr are very agnostic regarding catcodes: (unbraced) digits, binary operators, minus and plus signs as prefixes, parentheses, decimal point, may be indifferently of catcode letter or other or subscript or superscript, ..., it does not matter.

The characters +,-,*,/,^,!,&,|,?,:,<,>,=,(,) and the comma should not be active as everything is expanded along the way. If one of them is active, it can be prefixed with \string.

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

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

28.4 Expandability

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

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

28.5 Memory considerations

The parser creates an undefined control sequence for each intermediate computation (this does not refer to the intermediate steps needed in the evaluations of the \xintAdd, \xintMul, etc... macros corresponding to the infix operators, but only to each conversion of such an infix operator into a computation). So, a moderately sized expression might create 10, or 20 such control sequences. On my TEX installation, the memory available for such things is of circa 200,000 multi-letter control words. So this means that a document containing hundreds, perhaps even thousands of expressions will compile with no problem. But, if the package is used for computing plots[59], this may cause a problem.

There is a solution.[60]

A document can possibly do tens of thousands of evaluations only if some formulas are being used repeatedly, for example inside loops, with counters being incremented, or with data being fetched from a file. So it is the same formula used again and again with varying numbers inside.

With the \xintNewExpr command, it is possible to convert once and for all an expression containing parameters into an expandable macro with parameters. Only this initial

[59] 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.

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

definition of this macro actually activates the \xintexpr parser and will (very moderately) impact the hash-table: once this unique parsing is done, a macro with parameters is produced which is built-up recursively from the \xintAdd, \xintMul, etc... macros, exactly as it was necessary to do before the availability of the xintexpr package.

28.6 The \xintNewExpr command

The command is used as:

\xintNewExpr{\myformula}[n]{⟨*stuff*⟩}, where

- ⟨*stuff*⟩ will be inserted inside \xinttheexpr . . . \relax,
- n is an integer between zero and nine, inclusive, and tells how many parameters will \myformula have (it is *mandatory* even if n=0[61])
- the placeholders #1, #2, ..., #n are used inside ⟨*stuff*⟩ in their usual rôle.

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

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

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

> A "formula" created by \xintNewExpr is thus a macro whose parameters are given to a possibly very complicated combination of the various macros of xint and xintfrac; hence one can not use infix notation inside the arguments, as in for example \myformula {28^7-35^12} which would have been allowed by
>
> \def\myformula #1{\xinttheexpr (#1)^3\relax}
>
> One will have to do \myformula {\xinttheexpr 28^7-35^12\relax}, or redefine \myformula to have more parameters.

```
\xintNewExpr\DET[9]{ #1*#5*#9+#2*#6*#7+#3*#4*#8-#1*#6*#8-#2*#4*#9-#3*#5*#7 }
  \meaning\DET:macro:#1#2#3#4#5#6#7#8#9->\romannumeral-`0\xintSub{\xint
Sub{\xintSub{\xintAdd{\xintAdd{\xintMul{\xintMul{#1}{#5}}{#9}}{\xintMul
{\xintMul{#2}{#6}}{#7}}}{\xintMul{\xintMul{#3}{#4}}{#8}}}{\xintMul{\xin
tMul{#1}{#6}}{#8}}}{\xintMul{\xintMul{#2}{#4}}{#9}}}{\xintMul{\xintMul{
#3}{#5}}{#7}}
            \xintNum{\DET {1}{1}{1}{10}{-10}{5}{11}{-9}{6}}=0
           \xintNum{\DET {1}{2}{3}{10}{0}{-10}{21}{2}{-17}}=0
```

Remark: \meaning has been used within the argument to a \printnumber command, to avoid going into the right margin, but this zaps all spaces originally in the output from \meaning. Here is as an illustration the raw output of \meaning on the previous example:

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

```
  macro:#1#2#3#4#5#6#7#8#9->\romannumeral -‘0\xintSub {\xintSub {\xintSub
{\xintAdd {\xintAdd {\xintMul {\xintMul {#1}{#5}}{#9}}{\xintMul {\xintMul
{#2}{#6}}{#7}}}{\xintMul {\xintMul {#3}{#4}}{#8}}}{\xintMul {\xintMul {#1}{#6}}{#8}}}{\xin
Mul {\xintMul {#2}{#4}}{#9}}}{\xintMul {\xintMul {#3}{#5}}{#7}}
```

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

28.6.1 Use of conditional operators

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

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

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

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

Another example

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

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

28.6.2 Use of macros

For macros to be inserted within such a created xint-formula command, there are two cases:

- the macro does not involve the numbered parameters in its arguments: it may then be left as is, and will be evaluated once during the construction of the formula,
- it does involve at least one of the parameters as argument. Then:
  1. the whole thing (macro + argument) should be braced (not necessary if it is already included into a braced group),
  2. the macro should be coded with an underscore _ in place of the backslash \.
  3. the parameters should be coded with a dollar sign $1, $2, etc...

Here is a silly example illustrating the general principle (the macros here have equivalent functional forms which are more convenient; but some of the more obscure package macros of xint dealing with integers do not have functions pre-defined to be in correspondance with them):

```
\xintNewExpr\myformI[2]{ {_xintRound{$1}{$2}} - {_xintTrunc{$1}{$2}} }
\meaning\myformI:macro:#1#2->\romannumeral-`0\xintSub{\xintRound{#1}{#2
}}{\xintTrunc{#1}{#2}}
```

28.7 \xintnumexpr, \xintthenumexpr

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

28.8 \xintboolexpr, \xinttheboolexpr

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

28.9 \xintifboolexpr

x n n ⋆ `\xintifboolexpr{<expr>}{YES}{NO}` does `\xinttheexpr <expr>\relax` and then executes the `YES` or the `NO` branch depending on whether the outcome was non-zero or zero. The `<expr>` can be a pure logic expression using various `&` and `|`, with parentheses, the logic functions `all`, `any`, `xor`, the `bool` or `togl` operators, but it is not limited to them: the most general computation can be done, as we have here just a wrapper which tests if the outcome of the computation vanishes or not.

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

28.10 \xintifboolfloatexpr

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

28.11 \xintfloatexpr, \xintthefloatexpr

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

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

Note that `1.000000001` and `(1+1e-9)` will not be equivalent for `D=\xinttheDigits` set to nine or less. Indeed the addition implicit in `1+1e-9` (and executed when the closing

parenthesis is found) will provoke the rounding to 1. Whereas 1.000000001, when found as operand of one of the four elementary operations is kept with D+2 digits, and even more for the power function.

```
        \xintDigits:= 9; \xintthefloatexpr (1+1e-9)-1\relax=0.e0
 \xintDigits:= 9; \xintthefloatexpr 1.000000001-1\relax=1.00000000e-9
  For the fun of it: \xintDigits:=20;
     \xintthefloatexpr (1+1e-7)^1e7\relax=2.7182816925449662712e0
  \xintDigits:=36;
    \xintthefloatexpr ((1/13+1/121)*(1/179-1/173))/(1/19-1/18)\relax
              5.64487459334466559166166079096852897e-3
\xintFloat{\xintttheexpr ((1/13+1/121)*(1/179-1/173))/(1/19-1/18)\relax}
              5.64487459334466559166166079096852912e-3
```

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

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

We can try some crazy things:

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

Contrarily to some professional computing sofware which are our concurrents on this market, the 1.000000000000001 wasn't rounded to 1 despite the setting of \xintDigits; it would have been if we had input it as (1+1e-15).

28.12 \xintNewFloatExpr

This is exactly like \xintNewExpr except that the created formulas are set-up to use \xintthefloatexpr. The precision used for numbers fetched as parameters will be the one locally given by \xintDigits at the time of use of the created formulas, not \xintNewFloatExpr. However, the numbers hard-wired in the original expression will have been evaluated with the then current setting for \xintDigits.

28.13 \xintNewNumExpr

Like \xintNewExpr but using \xintthenumexpr.

28.14 \xintNewBoolExpr

Like \xintNewExpr but using \xintttheboolexpr.

28.15 Technicalities

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

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

The format of the output of `\xintexpr`⟨*stuff*⟩`\relax` is a `!` (with catcode 11) followed by `\XINT_expr_usethe` which prints an error message in the document and in the log file if it is executed, then a token doing the actual printing and finally a token `\.A/B[n]`. Using `\xinttheexpr` means zapping the first two things, the third one will then recover `A/B[n]` from the (presumably undefined, but it does not matter) control sequence `\.A/B[n]`.

I decided to put all intermediate results (from each evaluation of an infix operators, or of a parenthesized subpart of the expression, or from application of the minus as prefix, or of the exclamation sign as postfix, or any encountered braced material) inside `\csname...\endcsname`, as this can be done expandably and encapsulates an arbitrarily long fraction in a single token (left with undefined meaning), thus providing tremendous relief to the programmer in his/her expansion control.

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

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

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

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

28.16 Acknowledgements

I was greatly helped in my preparatory thinking, prior to producing such an expandable parser, by the commented source of the l3fp package, specifically the `l3fp-parse.dtx` file. Also the source of the `calc` package was instructive, despite the fact that here for `\xintexpr` the principles are necessarily different due to the aim of achieving expandability.

29 Commands of the xintbinhex package

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

The argument is first *f*-expanded. It then may start with an optional minus sign (unique, of category code other), followed with optional leading zeros (arbitrarily many, category code other) and then "digits" (hexadecimal letters may be of category code letter or other, and must be uppercased). The optional (unique) minus sign (plus sign is not allowed) is

kept in the output. Leading zeros are allowed, and stripped. The hexadecimal letters on output are of category code letter, and uppercased.

Contents

29.1 \xintDecToHex

f ⋆ Converts from decimal to hexadecimal.

```
\xintDecToHex{271828182845904523536028747135266249775724709369995957496696762772407663035354759457138217852516642742746639193200 3}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C6032936BF37DAC918814C63
```

29.2 \xintDecToBin

f ⋆ Converts from decimal to binary.

```
\xintDecToBin{271828182845904523536028747135266249775724709369995957496696762772407663035354759457138217852516642742746639193200 3}
->10001101010010011100101111100011001101001010010011010100101110000010100011111011111010000101010000001011110010001010011100011111000001011000101111100010000011011000100011100010010001011101011101111001010110101011101100000101110110011100011010010011100101111010000110110111001000110001100000001100101001001101101011111100110111110110010010001100010000001010011000110001100011
```

29.3 \xintHexToDec

f ⋆ Converts from hexadecimal to decimal.

```
\xintHexToDec{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C6032936BF37DAC918814C63}
->271828182845904523536028747135266249775724709369995957496696762772407663035354759457138217852516642742746639193200 3
```

29.4 \xintBinToDec

f ⋆ Converts from binary to decimal.

```
\xintBinToDec{10001101010010011100101111100011001101001010010011010100101110000010100011111011111010000101010000001011110010001010011100011111000001011000101111100010000011011000100011100010010001011101011101111001010110101011101100000101110110011100011010010011100101111010000110110111001000110001100000001100101001001101101011111100110111110110010010001100010000001010011000110001100011}
```

```
->271828182845904523536028747135266249775724709369995957496696762772
407663035354759457138217852516642742746639193200 3
```

29.5 \xintBinToHex

f ⋆ Converts from binary to hexadecimal.

```
\xintBinToHex{1000110101001001110010111110001100110100101001001101010
0101110000010100011111011111010000101010000000101111001000101001110000111
1100000101100010111111000100000110110001000111000100100010111010111011111
0010101101010111011000001011101100111000110100100111001011110100011011 0
1110011100100011011000110000000110010100100110110101111110011011111011 0
1011001001000110001000000010100110001100011}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F4
6DCE46C6032936BF37DAC918814C63
```

29.6 \xintHexToBin

f ⋆ Converts from hexadecimal to binary.

```
\xintHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B576
0BB38D272F46DCE46C6032936BF37DAC918814C63}
->1000110101001001110010111110001100110100101001001101010010111000000
1010001111101111010000101010000000101111001000101001110001111100000 1
0110001011111000100000110110001000111000100100010111010111011111001 01
0110101011101100000101110110011100011010010011100101111010001101101 1
1001110010001101100011000000011001010010011011010111111001101111101 1
0101100100100011000100000010100110001100011
```

29.7 \xintCHexToBin

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

```
\xintCHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B57
60BB38D272F46DCE46C6032936BF37DAC918814C63}
->1000110101001001110010111110001100110100101001001101010010111000000
1010001111101111010000101010000000101111001000101001110001111100000 1
0110001011111000100000110110001000111000100100010111010111011111001 01
0110101011101100000101110110011100011010010011100101111010001101101 1
1001110010001101100011000000011001010010011011010111111001101111101 1
0101100100100011000100000010100110001100011
```

30 Commands of the xintgcd package

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

Since release 1.09a the macros filter their inputs through the \xintNum macro, so one can use count registers, or fractions as long as they reduce to integers.

Contents

30.1 \xintGCD

Num Num
f f ⋆

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

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

30.2 \xintGCDof

Num
$f\to *$ 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.

30.3 \xintLCM

Num Num
f f ⋆

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

30.4 \xintLCMof

Num
$f\to *$ f ⋆

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

30.5 \xintBezout

Num Num
f f ⋆

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

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

30.6 \xintEuclideAlgorithm

Num Num
f f ⋆

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

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

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

The first token is the number of steps, the second is N, the third is the GCD, the fourth is M then the first quotient and remainder, the second quotient and remainder, ... until the final quotient and last (zero) remainder.

30.7 \xintBezoutAlgorithm

Num Num
f *f* ⋆

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

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

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

30.8 \xintTypesetEuclideAlgorithm

Num Num
f *f*

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

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

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

30.9 \xintTypesetBezoutAlgorithm

Num Num
f *f*

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

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

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

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

31 Commands of the xintseries package

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

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

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

Contents

31.1 \xintSeries

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

`\xintSeries{A}{B}{\coeff}` computes $\sum_{\texttt{n=A}}^{\texttt{n=B}}$ `\coeff{n}`. The initial and final indices must obey the `\numexpr` constraint of expanding to numbers at most `2^31-1`. The `\coeff` macro must be a one-parameter f-expandable command, taking on input an explicit number `n` and producing some number or fraction `\coeff{n}`; it is expanded at the time it is needed.[62]

[62] `\xintiiMON` is like `\xintMON` but does not parse its argument through `\xintNum`, for efficiency; other macros of this type are `\xintiiAdd`, `\xintiiMul`, `\xintiiSum`, `\xintiiPrd`, `\xintiiMMON`, `\xintiiLDg`, `\xintiiFDg`, `\xintiiOdd`, . . .

```
\def\coeff #1{\xintiiMON{#1}/#1.5} % (-1)^n/(n+1/2)
\edef\w {\xintSeries {0}{50}{\coeff}} % we want to re-use it
\edef\z {\xintJrr {\w}[0]} % the [0] for a microsecond gain.
% \xintJrr preferred to \xintIrr: a big common factor is suspected.
% But numbers much bigger would be needed to show the greater efficiency.
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} = \xintFrac\z \]
```

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}} = \frac{1739093382873709404321127921016266022787 14}{1100274671593900030252799172260397290505 75}$$

For info, before action by `\xintJrr` the inner representation of the result has a denominator of `\xintLen {\xintDenominator\w}`=117 digits. This troubled me as `101!!` has only 81 digits: `\xintLen {\xintQuo {\xintFac {101}}{\xintiMul {\xintiPow {2}{50}}{ \xintFac{50}}}}`=81. The explanation lies in the too clever to be efficient `#1.5` trick. It leads to a silly extra `5^{51}` (which has 36 digits) in the denominator. See the explanations in the next section.

Note: as soon as the coefficients look like factorials, it is more efficient to use the `\xintRationalSeries` macro whose evaluation will avoid a denominator build-up; indeed the raw operations of addition and subtraction of fractions blindly multiply out denominators. So the raw evaluation of $\sum_{n=0}^{N} 1/n!$ with `\xintSeries` will have a denominator equal to $\prod_{n=0}^{N} n!$. Needless to say this makes it more difficult to compute the exact value of this sum with `N=50`, for example, whereas with `\xintRationalSeries` the denominator does not get bigger than 50!.

For info: by the way $\prod_{n=0}^{50} n!$ is easily computed by **xint** and is a number with 1394 digits. And $\prod_{n=0}^{100} n!$ is also computable by **xint** (24 seconds on my laptop for the brute force iterated multiplication of all factorials, a specialized routine would do it faster) and has 6941 digits (this means more than two pages if printed...). Whereas 100! only has 158 digits.

```
\def\coeffleibnitz #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}
\cnta 1
\loop  % in this loop we recompute from scratch each partial sum!
% we can afford that, as \xintSeries is fast enough.
\noindent\hbox to 2em{\hfil\texttt{\the\cnta.} }%
         \xintTrunc {12}
                    {\xintSeries {1}{\cnta}{\coeffleibnitz}}\dots
\endgraf
\ifnum\cnta < 30 \advance\cnta 1 \repeat
```

1. 1.000000000000...
2. 0.500000000000...
3. 0.833333333333...
4. 0.583333333333...
5. 0.783333333333...
6. 0.616666666666...
7. 0.759523809523...
8. 0.634523809523...
9. 0.745634920634...
10. 0.645634920634...
11. 0.736544011544...
12. 0.653210678210...
13. 0.730133755133...
14. 0.658705183705...
15. 0.725371850371...
16. 0.662871850371...
17. 0.721695379783...
18. 0.666139824228...
19. 0.718771403175...
20. 0.668771403175...
21. 0.716390450794...
22. 0.670935905339...
23. 0.714414166209...
24. 0.672747499542...

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

31.2 \xintiSeries

num num f ⋆ `\xintiSeries{A}{B}{\coeff}` computes $\sum_{n=A}^{n=B}$ `\coeff{n}` where `\coeff{n}` must f-expand to a (possibly long) integer in the strict format.

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

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

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} \approx 1.58059930649352504123678950695672641448 10$$

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

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

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} \approx 1.58059930649352504123678950695672641448 04$$

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

This shows indeed that our sum of truncated terms estimated wrongly the 39th and 40th digits of the exact result[63] and that the sum of rounded terms fared a bit better.

31.3 \xintRationalSeries

num num Frac Frac
x x f f ⋆

`\xintRationalSeries{A}{B}{f}{\ratio}` evaluates $\sum_{n=A}^{n=B} F(n)$, where `F(n)` is specified indirectly via the data of `f=F(A)` and the one-parameter macro `\ratio` which must be such that `\macro{n}` expands to `F(n)/F(n-1)`. The name indicates that `\xintRationalSeries` was designed to be useful in the cases where `F(n)/F(n-1)` is a rational function of `n` but it may be anything expanding to a fraction. The macro `\ratio` must be an expandable-only compatible command and expand to its value after iterated full expansion of its first token. `A` and `B` are fed to a `\numexpr` hence may be count registers or arithmetic expressions built with such; they must obey the TEX bound. The initial term `f` may be a macro `\f`, it will be expanded to its value representing `F(A)`.

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

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

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

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

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

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

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

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

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

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

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

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

$\sum_{n=0}^{0}(.57)^n/n! = 1.0000000000000000000000000000000000000000000000000\dots$
$\sum_{n=0}^{10}(.57)^n/n! = 1.76826705137947002480668058035714285714285714285714\dots$
$\sum_{n=0}^{20}(.57)^n/n! = 1.76826705143373515162089324271187082272833005529082\dots$
$\sum_{n=0}^{30}(.57)^n/n! = 1.76826705143373515162089339282382144915484884979430\dots$
$\sum_{n=0}^{40}(.57)^n/n! = 1.76826705143373515162089339282382144915485219867776\dots$
$\sum_{n=0}^{50}(.57)^n/n! = 1.76826705143373515162089339282382144915485219867776\dots$

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

Here is a slightly more complicated evaluation:

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

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

31.4 \xintRationalSeriesX

num num Frac Frac
x x f f f ⋆

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

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

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

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

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

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

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

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

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

E(L(1[-1]))=435534952734304993753128478305695755446525998418916420656308534427154141471013807206588202981046013155342233701289165089056830056936564478988779520000000000/39594086612242519324387557078266845776303882240000000000000000000[-90] (length of numerator: 155)

E(L(12[-2]))=443453770054417465442109252347264824711893599160411729603882584198084153226108070707505890096280305971037133280203464123715588771418838065898295901413463294640275999939742200930346362653264 3

541704863984316744555312271367954598414044364800000000/395940866122
425193243875570782668457763038822400000000000000000000[-180] (length of numerator: 245)

E(L(123[-3]))=444641592651941777154254148848854866198954971552616390074295913531792113850864779762350800814416981762774148663052493217566759754097977420731516373336789722730765496139079185229545102248282391199621029237793811740122110919735433161132757168955864017710881850585395079859843831617966207195391567803471832147436302936555630100480000000039594086612242519324387557078266845776303882240000000000000000000[-270] (length of numerator: 335)

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

E(L(1/7))=518138516117322604916074833164833344883840590133006168125125346674309133532553948047136691585715900449768925914489452341864351924224000000000/453371201621089791788096627821377652892232653817581525466548360950870896010226899427964653421154077863588092639042087157760000000000000000000[0] (length of numerator: 141; length of denominator: 141)

E(L(1/71))=164799489177219556498025955806107098256158101756209369864657152282149780083067798039175325186850716609293467854603842163754716919123274624394132188208895310089982001627351524910000588238596565380887916286153347403881434316800000000/162510607383091507102283159265830434485606350979982865517923046004017115844425486049111273926394712850261667426510159483544917475146636033045963798199826115486814955381536472641379276308916890414267771321449447424000000000000000000000[0] (length of numerator: 232; length of denominator: 232)

E(L(1/712))=209623173880163120675481637897216200283968902248203238943136902264182865559717266406341976325767001357109452980607391271438079195073959301528254006087908156888129567520269011715459969154688799089625738271433856535377918700884980798641197021855117078629780316835353043067415753497212012899985019017494798220551782400000000/2093291722337673799732719862311619975662927884547744846526034295741465968177583093786412050480958301357075221213896546903011983961080605724903426024563430558292203346913309844190901402018394162270065876670575550330002721292096217682473000829618103432600036119035084894266166648343032219206471638591733760000000000000000000[0] (length of numerator: 322; length of denominator: 322)

For info the last fraction put into irreducible form still has 288 digits in its denominator.[64] 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

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

10 from the log and 9 from the exp, so 90. One more digit in the input means 90 more digits in the numerator of the output: obviously we can not go on composing such partial sums of series and hope that **xint** will joyfully do all at the speed of light! Briefly said, imagine that the rules of the game make the programmer like a security guard at an airport scanning machine: a never-ending flux of passengers keep on arriving and all you can do is re-shuffle the first nine of them, organize marriages among some, execute some, move children farther back among the first nine only. If a passenger comes along with many hand luggages, this will slow down the process even if you move him to ninth position, because sooner or later you will have to digest him, and the children will be big too. There is no way to move some guy out of the file and to a discrete interrogatory room for separate treatment or to give him/her some badge saying "I left my stuff in storage box 357".

Hence, truncating the output (or better, rounding) is the only way to go if one needs a general calculus of special functions. This is why the package **xintseries** provides, besides `\xintSeries`, `\xintRationalSeries`, or `\xintPowerSeries` which compute *exact* sums, also has `\xintFxPtPowerSeries` for fixed-point computations.

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

31.5 \xintPowerSeries

num num Frac Frac
x x f f ★

`\xintPowerSeries{A}{B}{\coeff}{f}` evaluates the sum $\sum_{n=A}^{n=B}$ `\coeff{n}` $\cdot$ `f`^{n}. The initial and final indices are given to a `\numexpr` expression. The `\coeff` macro (which, as argument to `\xintPowerSeries` is expanded only at the time `\coeff{n}` is needed) should be defined as a one-parameter expandable command, its input will be an explicit number.

The `f` can be either a fraction directly input or a macro `\f` expanding to such a fraction. It is actually more efficient to encapsulate an explicit fraction `f` in such a macro, if it has big numerators and denominators ('big' means hundreds of digits) as it will then take less space in the processing until being (repeatedly) used.

This macro computes the *exact* result (one can use it also for polynomial evaluation). Starting with release `1.04` a Horner scheme for polynomial evaluation is used, which has the advantage to avoid a denominator build-up which was plaguing the `1.03` version. [65]

> Note: as soon as the coefficients look like factorials, it is more efficient to use the `\xintRationalSeries` macro whose evaluation, also based on a similar Horner scheme, will avoid a denominator build-up originating in the coefficients themselves.

```
\def\geom #1{1[0]} % the geometric series
\def\f {5/17[0]}
\[ \sum_{n=0}^{n=20} \Bigl(\frac 5{17}\Bigr)^n
 =\xintFrac{\xintIrr{\xintPowerSeries {0}{20}{\geom}{\f}}}
 =\xintFrac{\xinttheexpr (17^21-5^21)/12/17^20\relax}\]
```

$$\sum_{n=0}^{n=20}\Bigl(\frac{5}{17}\Bigr)^n = \frac{57576611593776579768853 41}{40642314066475725224016 01} = \frac{69091933912531895722624092}{48770776879770870268819212}$$

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

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

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

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

```
\cnta 1 % previously declared count
\loop   % in this loop we recompute from scratch each partial sum!
% we can afford that, as \xintPowerSeries is fast enough.
\noindent\hbox to 2em{\hfil\texttt{\the\cnta.} }%
         \xintTrunc {12}
             {\xintPowerSeries {1}{\cnta}{\coefflog}{\f}}\dots
\endgraf
\ifnum \cnta < 30 \advance\cnta 1 \repeat
```

1. 0.500000000000...	11. 0.693109245355...	21. 0.693147159757...
2. 0.625000000000...	12. 0.693129590407...	22. 0.693147170594...
3. 0.666666666666...	13. 0.693138980431...	23. 0.693147175777...
4. 0.682291666666...	14. 0.693143340085...	24. 0.693147178261...
5. 0.688541666666...	15. 0.693145374590...	25. 0.693147179453...
6. 0.691145833333...	16. 0.693146328265...	26. 0.693147180026...
7. 0.692261904761...	17. 0.693146777052...	27. 0.693147180302...
8. 0.692750186011...	18. 0.693146988980...	28. 0.693147180435...
9. 0.692967199900...	19. 0.693147089367...	29. 0.693147180499...
10. 0.693064856150...	20. 0.693147137051...	30. 0.693147180530...

```
%\def\coeffarctg #1{1/\the\numexpr\xintMON{#1}*(2*#1+1)\relax }%
\def\coeffarctg  #1{1/\the\numexpr\ifodd #1 -2*#1-1\else2*#1+1\fi\relax }%
% the above gives (-1)^n/(2n+1). The sign being in the denominator,
%             **** no [0] should be added ****,
% else nothing is guaranteed to work (even if it could by sheer luck)
% NOTE in passing this aspect of \numexpr:
%         ****  \numexpr -(1)\relax does not work!!! ****
\def\f {1/25[0]}% 1/5^2
\[\mathrm{Arctg}(\frac15)\approx
            \frac15\sum_{n=0}^{15} \frac{(-1)^n}{(2n+1)25^n}
= \xintFrac{\xintIrr {\xintDiv
            {\xintPowerSeries {0}{15}{\coeffarctg}{\f}}{5}}}\]
```

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

31.6 \xintPowerSeriesX

num num Frac Frac
x x f f

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

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

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

31.7 \xintFxPtPowerSeries

num num ★
Frac Frac num ★
x x
f f x

\xintFxPtPowerSeries{A}{B}{\coeff}{f}{D} computes $\sum_{n=A}^{n=B}$ \coeff{n} · f^n with each term of the series truncated to D digits after the decimal point. As usual, A and B are completely expanded through their inclusion in a \numexpr expression. Regarding D it will be similarly be expanded each time it is used inside an \xintTrunc. The one-parameter macro \coeff is similarly expanded at the time it is used inside the computations. Idem for f. If f itself is some complicated macro it is thus better to use the variant \xintFxPtPowerSeriesX which expands it first and then uses the result of that expansion.

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

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

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

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

```
\def\coeffexp #1{1/\xintFac {#1}[0]}% 1/n!
\def\f {-1/2[0]}% [0] for faster input parsing
\cnta 0 % previously declared \count register
\noindent\loop
$\xintFxPtPowerSeries {0}{\cnta}{\coeffexp}{\f}{20}$\\
\ifnum\cnta<19 \advance\cnta 1 \repeat\par
% One should **not** trust the final digits, as the potential truncation
% errors of up to 10^{-20} per term accumulate and never disappear! (the
% effect is attenuated by the alternating signs in the series). We  can
% confirm that the last two digits (of our evaluation of the nineteenth
% partial sum) are wrong via the evaluation with more digits:
\xintFxPtPowerSeries {0}{19}{\coeffexp}{\f}{25}= 0.6065306597126334236037992
```

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

$$\texttt{\textbackslash xintPowerSeries \{0\}\{19\}\{\textbackslash coeffexp\}\{\textbackslash f\}} = \frac{3868274616003639731775 7}{6377706640314571161600 0} = 0.6065306597126334236037991 52126\ldots$$

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

31.8 \xintFxPtPowerSeriesX

$\overset{\text{num}}{\underset{f}{\text{Frac}}}\ \overset{\text{num}}{\underset{f}{\text{Frac}}}\ \overset{x}{\underset{x}{\text{num}}}\ x\ \star$

`\xintFxPtPowerSeriesX{A}{B}{\coeff}{\f}{D}` computes, exactly as `\xintFxPtPowerSeries`, the sum of `\coeff{n}·\f^n` from `n=A` to `n=B` with each term of the series being *truncated* to `D` digits after the decimal point. The sole difference is that `\f` is first expanded and it is the result of this which is used in the computations.

Let us illustrate this on the numerical exploration of the identity

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

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

```
\cnta 0
\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}% (-1)^{n-
1}/n
\def\coeffalt #1{\the\numexpr\ifodd#1 -1\else1\fi\relax [0]}%   (-1)^n
\loop
```

```
\noindent \hbox to 2.5cm {\hss\texttt{D(\the\cnta/100): }}%
\xintAdd {\xintFxPtPowerSeriesX {1}{10}{\coefflog}{\the\cnta [-2]}{5}}
         {\xintFxPtPowerSeriesX {1}{10}{\coefflog}
             {\xintFxPtPowerSeriesX {1}{10}{\coeffalt}{\the\cnta [-2]}{5}}
          {5}}\endgraf
\ifnum\cnta < 49 \advance\cnta 7 \repeat
```

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

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

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

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

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

31.9 \xintFloatPowerSeries

$[{}^{\text{num}}_{f}\,x_{\text{Frac}}^{\text{num}}\,{}^{\text{num}}_{f}\,x^{\text{Frac}}_{f}\,\star]$

`\xintFloatPowerSeries[P]{A}{B}{\coeff}{f}` computes $\sum_{\texttt{n=A}}^{\texttt{n=B}}$ `\coeff{n}` $\cdot$ `f`$^{\texttt{n}}$ with a floating point precision given by the optional parameter `P` or by the current setting of `\xintDigits`.

In the current, preliminary, version, no attempt has been made to try to guarantee to the final result the precision `P`. Rather, `P` is used for all intermediate floating point evaluations. So rounding errors will make some of the last printed digits invalid. The operations done are first the evaluation of `f^A` using `\xintFloatPow`, then each successive power

is obtained from this first one by multiplication by f using \xintFloatMul, then again with \xintFloatMul this is multiplied with \coeff{n}, and the sum is done adding one term at a time with \xintFloatAdd. To sum up, this is just the naive transformation of \xintFxPtPowerSeries from fixed point to floating point.

```
\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}%
\xintFloatPowerSeries [8]{1}{30}{\coefflog}{-1/2[0]}
                                -6.9314718e-1
```

31.10 \xintFloatPowerSeriesX

[num x num num / Frac f Frac f x ★]

\xintFloatPowerSeriesX[P]{A}{B}{\coeff}{f} is like \xintFloatPowerSeries with the difference that f is expanded once and for all at the start of the computation, thus allowing efficient chaining of such series evaluations.

```
\def\coeffexp #1{1/\xintFac {#1}[0]}% 1/n! (exact, not float)
\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}%
\xintFloatPowerSeriesX [8]{0}{30}{\coeffexp}
    {\xintFloatPowerSeries [8]{1}{30}{\coefflog}{-1/2[0]}}
                                5.0000001e-1
```

31.11 Computing $\log 2$ and π

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

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

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

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

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

```
\def\coefflog #1{1/#1[0]}% 1/n
\def\xa {13/256[0]}%  we will compute log(1-13/256)
\def\xb {1/9[0]}%     we will compute log(1-1/9)
\def\LogTwo #1%
%  get log(2)=-2log(1-13/256)- 5log(1-1/9)
```

```
{% we want to use \printnumber, hence need something expanding in two steps
 % only, so we use here the \romannumeral0 method
  \romannumeral0\expandafter\LogTwoDoIt \expandafter
    % Nb Terms for 1/9:
  {\the\numexpr #1*150/143\expandafter}\expandafter
    % Nb Terms for 13/256:
  {\the\numexpr #1*100/129\expandafter}\expandafter
    % We print #1 digits, but we know the ending ones are garbage
  {\the\numexpr #1\relax}% allows #1 to be a count register
}%
\def\LogTwoDoIt #1#2#3%
%  #1=nb of terms for 1/9, #2=nb of terms for 13/256,
{% #3=nb of digits for computations, also used for printing
 \xinttrunc {#3} % lowercase form to stop the \romannumeral0 expansion!
 {\xintAdd
  {\xintMul {2}{\xintFxPtPowerSeries {1}{#2}{\coefflog}{\xa}{#3}}}
  {\xintMul {5}{\xintFxPtPowerSeries {1}{#1}{\coefflog}{\xb}{#3}}}%
 }%
}%
\noindent $\log 2 \approx \LogTwo {60}\dots$\endgraf
\noindent\phantom{$\log 2$}${}\approx{}$\printnumber{\LogTwo {65}}\dots\endgraf
\noindent\phantom{$\log 2$}${}\approx{}$\printnumber{\LogTwo {70}}\dots\endgraf
```

$\log 2 \approx 0.693147180559945309417232121458176568075500134360255254120484\ldots$

$\approx 0.693147180559945309417232121458176568075500134360255254120680$
00711...

$\approx 0.693147180559945309417232121458176568075500134360255254120680$
0094933723...

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

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

Let us turn now to Pi, computed with the Machin formula. Again the numbers of terms to keep in the two arctg series were roughly estimated, and some experimentations showed

that removing the last three digits was enough (at least for D=0-100 range). And the algorithm does print the correct digits when used with D=1000 (to be convinced of that one needs to run it for D=1000 and again, say for D=1010.) A theoretical analysis could help confirm that this algorithm always gets better than 10^{-D} precision, but again, strings of zeros or nines encountered in the decimal expansion may falsify the ending digits, nines may be zeros (and the last non-nine one should be increased) and zeros may be nine (and the last non-zero one should be decreased).

```
% pi = 16 Arctg(1/5) - 4 Arctg(1/239) (John Machin's formula)
\def\coeffarctg #1{\the\numexpr\ifodd#1 -1\else1\fi\relax/%
                                        \the\numexpr 2*#1+1\relax [0]}%
% the above computes (-1)^n/(2n+1).
% Alternatives:
% \def\coeffarctg #1{1/\the\numexpr\xintiiMON{#1}*(2*#1+1)\relax }%
% The [0] can *not* be used above, as the denominator is signed.
% \def\coeffarctg #1{\xintiiMON{#1}/\the\numexpr 2*#1+1\relax [0]}%
\def\xa {1/25[0]}%      1/5^2, the [0] for faster parsing
\def\xb {1/57121[0]}% 1/239^2, the [0] for faster parsing
\def\Machin #1{% \Machin {\mycount} is allowed
    \romannumeral0\expandafter\MachinA \expandafter
     % number of terms for arctg(1/5):
    {\the\numexpr (#1+3)*5/7\expandafter}\expandafter
     % number of terms for arctg(1/239):
    {\the\numexpr (#1+3)*10/45\expandafter}\expandafter
     % do the computations with 3 additional digits:
    {\the\numexpr #1+3\expandafter}\expandafter
     % allow #1 to be a count register:
    {\the\numexpr #1\relax }}%
\def\MachinA #1#2#3#4%
% #4: digits to keep after decimal point for final printing
% #3=#4+3: digits for evaluation of the necessary number of terms
% to be kept in the arctangent series, also used to truncate each
% individual summand.
{\xinttrunc {#4} % must be lowercase to stop \romannumeral0!
 {\xintSub
  {\xintMul {16/5}{\xintFxPtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}}
  {\xintMul {4/239}{\xintFxPtPowerSeries {0}{#2}{\coeffarctg}{\xb}{#3}}}%
 }}%
\[ \pi = \Machin {60}\dots \]
```

$$\pi = 3.14159265358979323846264338327950288419716939937510582097494 4\ldots$$

Here is a variant\MachinBis, which evaluates the partial sums *exactly* using \xintPowerSeries, before their final truncation. No need for a "+3" then.

```
\def\MachinBis #1{% #1 may be a count register,
% the final result will be truncated to #1 digits post decimal point
    \romannumeral0\expandafter\MachinBisA \expandafter
     % number of terms for arctg(1/5):
```

```
    {\the\numexpr #1*5/7\expandafter}\expandafter
     % number of terms for arctg(1/239):
    {\the\numexpr #1*10/45\expandafter}\expandafter
      % allow #1 to be a count register:
    {\the\numexpr #1\relax }}%
\def\MachinBisA #1#2#3%
{\xinttrunc {#3} %
 {\xintSub
   {\xintMul {16/5}{\xintPowerSeries {0}{#1}{\coeffarctg}{\xa}}}
   {\xintMul{4/239}{\xintPowerSeries {0}{#2}{\coeffarctg}{\xb}}}%
}}%
```

Let us use this variant for a loop showing the build-up of digits:

```
\cnta 0 % previously declared \count register
\loop
\MachinBis{\cnta} \endgraf  % Plain's \loop does not accept \par
\ifnum\cnta < 30 \advance\cnta 1 \repeat
```

	3.141592653589793
3.	3.1415926535897932
3.1	3.14159265358979323
3.14	3.141592653589793238
3.141	3.1415926535897932384
3.1415	3.14159265358979323846
3.14159	3.141592653589793238462
3.141592	3.1415926535897932384626
3.1415926	3.14159265358979323846264
3.14159265	3.141592653589793238462643
3.141592653	3.1415926535897932384626433
3.1415926535	3.14159265358979323846264338
3.14159265358	3.141592653589793238462643383
3.141592653589	3.1415926535897932384626433832
3.1415926535897	3.14159265358979323846264338327
3.14159265358979	3.141592653589793238462643383279

You want more digits and have some time? Copy the `\Machin` code to a Plain TEX or LATEX document loading **xintseries**, and compile:

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

This will create a file with the correct first 1000 digits of π after the decimal point. On my laptop (a 2012 model) this took about 42 seconds last time I tried (and for 200 digits it is less than 1 second). As mentioned in the introduction, the file pi.tex by D. Roegel shows that orders of magnitude faster computations are possible within TEX, but recall our constraints of complete expandability and be merciful, please.

Why truncating rather than rounding? One of our main competitors on the market of scientific computing, a canadian product (not encumbered with expandability constraints, and having barely ever heard of TEX ;-), prints numbers rounded in the last digit. Why didn't we follow suit in the macros `\xintFxPtPowerSeries` and `\xintFxPtPowerSeriesX`? To round at `D` digits, and excluding a rewrite or cloning of the division algo-

rithm which anyhow would add to it some overhead in its final steps, **xintfrac** needs to truncate at D+1, then round. And rounding loses information! So, with more time spent, we obtain a worst result than the one truncated at D+1 (one could imagine that additions and so on, done with only D digits, cost less; true, but this is a negligeable effect per summand compared to the additional cost for this term of having been truncated at D+1 then rounded). Rounding is the way to go when setting up algorithms to evaluate functions destined to be composed one after the other: exact algebraic operations with many summands and an f variable which is a fraction are costly and create an even bigger fraction; replacing f with a reasonable rounding, and rounding the result, is necessary to allow arbitrary chaining.

But, for the computation of a single constant, we are really interested in the exact decimal expansion, so we truncate and compute more terms until the earlier result gets validated. Finally if we do want the rounding we can always do it on a value computed with D+1 truncation.

32 Commands of the xintcfrac package

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

Contents

32.1 Package overview

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

cated. With the amsmath macro \cfrac one can display such a continued fraction as

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

Here is a concrete example:

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

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

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

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

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

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

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

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

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

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

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

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

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

The left hand side was obtained with the following code:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
143+1/2+1/5+-1/4+-1/4+-1/4+-1/3+1/2+1/2+1/6+-1/22+1/2+1/10+-1/5+
-1/11+-1/3+1/4+-1/2+1/2+1/4+-1/2+1/23+1/3+1/8+-1/6+-1/9
```

. If we apply \xintGCtoF to this generalized continued fraction, we discover that the original fraction was reducible:

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

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

```
\xintGCtoF {143+1/2+...+-1/6}=32812488771062672 9/2287346221788023
```

and indeed:

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

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

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

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

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

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

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

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

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

Next, we use the following code:

```
$\xintFrac{49171/18089}\to{}$
```

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

It produces:

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

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

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

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

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

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

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

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

There are also \xintGCntoF and \xintGCntoGC which allow the same for generalized

fractions. The following initial portion of a generalized continued fraction for π:

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

was obtained with this code:

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

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

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

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

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

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

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

- a comma separated list of the first 36 coefficients is produced by \xintCntoCs,

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

1. $2.000000000000000000000000000000\cdots = 2$
2. $3.000000000000000000000000000000\cdots = 3$
3. $2.666666666666666666666666666666\cdots = \frac{8}{3}$
4. $2.750000000000000000000000000000\cdots = \frac{11}{4}$
5. $2.714285714285714285714285714285\cdots = \frac{19}{7}$
6. $2.718750000000000000000000000000\cdots = \frac{87}{32}$
7. $2.717948717948717948717948717948\cdots = \frac{106}{39}$
8. $2.718309859154929577464788732394\cdots = \frac{193}{71}$
9. $2.718279569892473118279569892473\cdots = \frac{1264}{465}$
10. $2.718283582089552238805970149253\cdots = \frac{1457}{536}$
11. $2.718281718281718281718281718281\cdots = \frac{2721}{1001}$
12. $2.718281835205992509363295880149\cdots = \frac{23225}{8544}$
13. $2.718281822943949711891042430591\cdots = \frac{25946}{9545}$
14. $2.718281828735695726684725523798\cdots = \frac{49171}{18089}$
15. $2.718281828445401318035025074172\cdots = \frac{517656}{190435}$
16. $2.718281828470583721777828930962\cdots = \frac{566827}{208524}$
17. $2.718281828458563411277850606202\cdots = \frac{1084483}{398959}$
18. $2.718281828459065114074529546648\cdots = \frac{13580623}{4996032}$
19. $2.718281828459028013207065591026\cdots = \frac{14665106}{5394991}$
20. $2.718281828459045851404621084949\cdots = \frac{28245729}{10391023}$
21. $2.718281828459045213521983758221\cdots = \frac{410105312}{150869313}$
22. $2.718281828459045254624795027092\cdots = \frac{438351041}{161260336}$
23. $2.718281828459045234757560631479\cdots = \frac{848456353}{312129649}$
24. $2.718281828459045235379013372772\cdots = \frac{14013652689}{5155334720}$
25. $2.718281828459045235343535532787\cdots = \frac{14862109042}{5467464369}$
26. $2.718281828459045235360753230188\cdots = \frac{28875761731}{10622799089}$

27. $2.71828182845904523536027459394\cdots = \frac{534625820200}{196677847971}$
28. $2.71828182845904523536029912091\cdots = \frac{563501581931}{207300647060}$
29. $2.71828182845904523536028717990\cdots = \frac{1098127402131}{403978495031}$
30. $2.71828182845904523536028747861\cdots = \frac{22526049624551}{8286870547680}$
31. $2.71828182845904523536028746472\cdots = \frac{23624177026682}{8690849042711}$
32. $2.71828182845904523536028747150\cdots = \frac{46150226651233}{16977719590391}$
33. $2.71828182845904523536028747134\cdots = \frac{1038929163353808}{382200680031313}$
34. $2.71828182845904523536028747135\cdots = \frac{1085079390005041}{399178399621704}$
35. $2.71828182845904523536028747135\cdots = \frac{2124008553358849}{781379079653017}$
36. $2.71828182845904523536028747135\cdots = \frac{52061284670617417}{19152276311294112}$

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

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

Numerator = 5689640388718962675975238923158078752938890176679174460572320245471922969611182301752438601749953108177313670124170860974963432938290617086097496343293829061708609749634329382906

Denominator = 3311238176697376193062563608163567533654688237293144381562056154632466597285818654613376920631489160195506145705925533766114264521722392553376611426452172239255337661142645217223

Expansion = 1.7182818284590452353602874713526624977572470936999595749666967627724076630353547594571382178525166427427466391932003059921817413596629043572900334295260595630738132328627943490763233829880753195251019011573834187930702154089149934884167509244761460668082264800168477411 8...

One can also use a centered continued fraction: we get more digits but there are also more computations as the numerators may be either 1 or -1.

32.2 \xintCFrac

Frac
f ⋆

`\xintCFrac{f}` is a math-mode only, LaTeX with `amsmath` only, macro which first com-

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

32.3 \xintGCFrac

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

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

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

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

32.4 \xintGCtoGCx

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

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

Plain TEX+amstex users may be interested in:

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

32.5 \xintFtoCs

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

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

$$-\frac{5262046}{89233}=[-59,33,27,100]$$

32.6 \xintFtoCx

Frac *n f* ⋆ \xintFtoCx{sep}{f} returns the list of the coefficients of the simple continued fraction of f, withing group braces and separated with the help of sep.

```
$$\xintFtoCx {+\cfrac1\\ }{f}\endcfrac$$
```

will display the continued fraction in \cfrac format, with Plain TEX and amstex.

32.7 \xintFtoGC

Frac *f* ⋆

\xintFtoGC{f} does the same as \xintFtoCx{+1/}{f}. Its output may thus be used in the package macros expecting such an 'inline format'. This continued fraction is a *simple* one, not a *generalized* one, but as it is produced in the format used for user input of generalized continued fractions, the macro was called \xintFtoGC rather than \xintFtoC for example.

```
566827/208524=\xintFtoGC {566827/208524}
```

566827/208524=2+1/1+1/2+1/1+1/1+1/4+1/1+1/1+1/6+1/1+1/1+1/8+1/1+1/1+1/11

32.8 \xintFtoCC

Frac *f* ⋆

\xintFtoCC{f} returns the 'centered' continued fraction of f, in 'inline format'.

```
566827/208524=\xintFtoCC {566827/208524}
```

566827/208524=3+-1/4+-1/2+1/5+-1/2+1/7+-1/2+1/9+-1/2+1/11

```
\[\xintFrac{566827/208524} = \xintGCFrac{\xintFtoCC{566827/208524}}\]
```

$$\frac{566827}{208524} = 3 - \cfrac{1}{4 - \cfrac{1}{2 + \cfrac{1}{5 - \cfrac{1}{2 + \cfrac{1}{7 - \cfrac{1}{2 + \cfrac{1}{9 - \cfrac{1}{2 + \cfrac{1}{11}}}}}}}}}$$

32.9 \xintFtoCv

Frac *f* ⋆

\xintFtoCv{f} returns the list of the (braced) convergents of f, with no separator. To be treated with \xintAssignArray or \xintListWithSep.

```
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintFtoCv{5211/3748}}}\]
```

$$1 \to \frac{3}{2} \to \frac{4}{3} \to \frac{7}{5} \to \frac{25}{18} \to \frac{32}{23} \to \frac{57}{41} \to \frac{317}{228} \to \frac{374}{269} \to \frac{691}{497} \to \frac{5211}{3748}$$

32.10 \xintFtoCCv

Frac *f* ⋆

\xintFtoCCv{f} returns the list of the (braced) centered convergents of f, with no separator. To be treated with \xintAssignArray or \xintListWithSep.

```
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintFtoCCv{5211/3748}}}\]
```

$$1 \to \frac{4}{3} \to \frac{7}{5} \to \frac{32}{23} \to \frac{57}{41} \to \frac{374}{269} \to \frac{691}{497} \to \frac{5211}{3748}$$

32.11 \xintCstoF

f ⋆ `\xintCstoF{a,b,c,d,...,z}` computes the fraction corresponding to the coefficients, which may be fractions or even macros expanding to such fractions (in two steps). The final fraction may then be highly reducible.

```
\[\xintGCFrac {-1+1/3+1/-5+1/7+1/-9+1/11+1/-13}
=\xintSignedFrac{\xintCstoF {-1,3,-5,7,-9,11,-13}}
=\xintSignedFrac{\xintGCtoF {-1+1/3+1/-5+1/7+1/-9+1/11+1/-13}}\]
```

$$-1+\cfrac{1}{3+\cfrac{1}{-5+\cfrac{1}{7+\cfrac{1}{-9+\cfrac{1}{11+\cfrac{1}{-13}}}}}}=-\frac{75887}{118187}=-\frac{75887}{118187}$$

```
\xintGCFrac{{1/2}+1/{1/3}+1/{1/4}+1/{1/5}}=
\xintFrac{\xintCstoF {1/2,1/3,1/4,1/5}}
```

$$\frac{1}{2}+\cfrac{1}{\frac{1}{3}+\cfrac{1}{\frac{1}{4}+\frac{1}{5}}}=\frac{159}{66}$$

A generalized continued fraction may produce a reducible fraction (`\xintCstoF` tries its best not to accumulate in a silly way superfluous factors but will not do simplifications which would be obvious to a human, like simplification by 3 in the result above).

32.12 \xintCstoCv

f ⋆ `\xintCstoCv{a,b,c,d,...,z}` returns the list of the corresponding convergents. It is allowed to use fractions as coefficients (the computed convergents have then no reason to be the real convergents of the final fraction). When the coefficients are integers, the convergents are irreducible fractions, but otherwise it is not necessarily the case.

```
\xintListWithSep:{\xintCstoCv{1,2,3,4,5,6}}
1/1:3/2:10/7:43/30:225/157:1393/972
\xintListWithSep:{\xintCstoCv{1,1/2,1/3,1/4,1/5,1/6}}
1/1:3/1:9/7:45/19:225/159:1575/729
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintCstoCv
{\xintPow {-.3}{-5},7.3/4.57,\xintCstoF{3/4,9,-1/3}}}}\]
```

$$\frac{-100000}{243}\to\frac{-72888949}{177390}\to\frac{-2700356878}{6567804}$$

32.13 \xintCstoGC

f ⋆ `\xintCstoGC{a,b,..,z}` transforms a comma separated list (or something expanding to such a list) into an 'inline format' continued fraction `{a}+1/{b}+1/...+1/{z}`. The coefficients are just copied and put within braces, without expansion. The output can then be used in `\xintGCFrac` for example.

```
\[\xintGCFrac {\xintCstoGC {-1,1/2,-1/3,1/4,-1/5}}
=\xintSignedFrac {\xintCstoF {-1,1/2,-1/3,1/4,-1/5}}\]
```

$$-1+\cfrac{1}{\frac{1}{2}+\cfrac{1}{\frac{-1}{3}+\cfrac{1}{\frac{1}{4}+\cfrac{1}{\frac{-1}{5}}}}}=-\frac{145}{83}$$

32.14 \xintGCtoF

f ⋆ `\xintGCtoF{a+b/c+d/e+f/g+.....+v/w+x/y}` computes the fraction defined by the inline generalized continued fraction. Coefficients may be fractions but must then be put within braces. They can be macros. The plus signs are mandatory.

```
\[\xintGCFrac {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}} =
\xintFrac{\xintGCtoF {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}}} =
\xintFrac{\xintIrr{\xintGCtoF
                   {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintFac {6}}}}\]
```

$$1+\cfrac{3375\cdot 10^{-3}}{\frac{1}{7}-\cfrac{\frac{3}{5}}{720}}=\frac{88629000}{3579000}=\frac{29543}{1193}$$

```
\[ \xintGCFrac{{1/2}+{2/3}/{4/5}+{1/2}/{1/5}+{3/2}/{5/3}} =
   \xintFrac{\xintGCtoF {{1/2}+{2/3}/{4/5}+{1/2}/{1/5}+{3/2}/{5/3}}} \]
```

$$\frac{1}{2}+\cfrac{\frac{2}{3}}{\frac{4}{5}+\cfrac{\frac{1}{2}}{\frac{1}{5}+\cfrac{\frac{3}{2}}{\frac{5}{3}}}}=\frac{4270}{4140}$$

The macro tries its best not to accumulate superfluous factor in the denominators, but doesn't reduce the fraction to irreducible form before returning it and does not do simplifications which would be obvious to a human.

32.15 \xintGCtoCv

f ⋆ `\xintGCtoCv{a+b/c+d/e+f/g+.....+v/w+x/y}` returns the list of the corresponding convergents. The coefficients may be fractions, but must then be inside braces. Or they may be macros, too.

The convergents will in the general case be reducible. To put them into irreducible form, one needs one more step, for example it can be done with \xintApply\xintIrr.

```
\[\xintListWithSep{,}{\xintApply\xintFrac
                {\xintGCtoCv{3+{-2}/{7/2}+{3/4}/12+{-56}/3}}}\]
\[\xintListWithSep{,}{\xintApply\xintFrac{\xintApply\xintIrr
                {\xintGCtoCv{3+{-2}/{7/2}+{3/4}/12+{-56}/3}}}}\]
```

$$3, \frac{17}{7}, \frac{834}{342}, \frac{1306}{542}$$

$$3, \frac{17}{7}, \frac{139}{57}, \frac{653}{271}$$

32.16 \xintCntoF

$\overset{\text{num}}{x}\, f\star$ \xintCntoF{N}{\macro} computes the fraction f having coefficients c(j)=\macro{j} for j=0,1,...,N. The N parameter is given to a \numexpr. The values of the coefficients, as returned by \macro do not have to be positive, nor integers, and it is thus not necessarily the case that the original c(j) are the true coefficients of the final f.

```
\def\macro #1{\the\numexpr 1+#1*#1\relax}\xintCntoF {5}{\macro}
72625/49902[0]
```

32.17 \xintGCntoF

$\overset{\text{num}}{x}\, ff\star$ \xintGCntoF{N}{\macroA}{\macroB} returns the fraction f corresponding to the inline generalized continued fraction a0+b0/a1+b1/a2+....+b(N-1)/aN, with a(j)=\macroA{j} and b(j)=\macroB{j}. The N parameter is given to a \numexpr.

$$1+\cfrac{1}{2-\cfrac{1}{3+\cfrac{1}{1-\cfrac{1}{2+\cfrac{1}{3-\cfrac{1}{1}}}}}} = \frac{39}{25}$$

There is also \xintGCntoGC to get the 'inline format' continued fraction. The previous display was obtained with:

```
\def\coeffA #1{\the\numexpr #1+4-3*((#1+2)/3)\relax }%
    \def\coeffB #1{\xintMON{#1}}% (-1)^n
\[\xintGCFrac{\xintGCntoGC {6}{\coeffA}{\coeffB}}
 = \xintFrac{\xintGCntoF {6}{\coeffA}{\coeffB}}\]
```

32.18 \xintCntoCs

$\overset{\text{num}}{x}\, f\star$ \xintCntoCs{N}{\macro} produces the comma separated list of the corresponding coefficients, from n=0 to n=N. The N is given to a \numexpr.

```
\def\macro #1{\the\numexpr 1+#1*#1\relax}
```

```
\xintCntoCs {5}{\macro}->1,2,5,10,17,26
\[\xintFrac{\xintCntoF {5}{\macro}}=\xintCFrac{\xintCntoF {5}{\macro}}\]
```

$$\frac{72625}{49902} = 1 + \cfrac{1}{2 + \cfrac{1}{5 + \cfrac{1}{10 + \cfrac{1}{17 + \cfrac{1}{26}}}}}$$

32.19 \xintCntoGC

$^{\text{num}}_{x}f\star$ `\xintCntoGC{N}{\macro}` evaluates the `c(j)=\macro{j}` from `j=0` to `j=N` and returns a continued fraction written in inline format: `{c(0)}+1/{c(1)}+1/...+1/{c(N)}`. The parameter `N` is given to a `\numexpr`. The coefficients, after expansion, are, as shown, being enclosed in an added pair of braces, they may thus be fractions.

```
\def\macro #1{\the\numexpr\ifodd#1 -1-#1\else1+#1\fi\relax/%
              \the\numexpr 1+#1*#1\relax}
\edef\x{\xintCntoGC {5}{\macro}}\meaning\x
macro:->{1/1}+1/{-2/2}+1/{3/5}+1/{-4/10}+1/{5/17}+1/{-6/26}
\[\xintGCFrac{\xintCntoGC {5}{\macro}}\]
```

$$1 + \cfrac{1}{\frac{-2}{2} + \cfrac{1}{\frac{3}{5} + \cfrac{1}{\frac{-4}{10} + \cfrac{1}{\frac{5}{17} + \cfrac{1}{\frac{-6}{26}}}}}}$$

32.20 \xintGCntoGC

$^{\text{num}}_{x}ff\star$ `\xintGCntoGC{N}{\macroA}{\macroB}` evaluates the coefficients and then returns the corresponding `{a0}+{b0}/{a1}+{b1}/{a2}+...+{b(N-1)}/{aN}` inline generalized fraction. `N` is givent to a `\numexpr`. As shown, the coefficients are enclosed into added pairs of braces, and may thus be fractions.

```
  #1*#1*#1+1\relax}%
\def\bn #1{\the\numexpr \xintiiMON{#1}*(#1+1)\relax}%
$\xintGCntoGC {5}{\an}{\bn}}=\xintGCFrac {\xintGCntoGC {5}{\an}{\bn}} =
\displaystyle\xintFrac {\xintGCntoF {5}{\an}{\bn}}$\par
```

$$1 + 1/2 + -2/9 + 3/28 + -4/65 + 5/126 = 1 + \cfrac{1}{2 - \cfrac{2}{9 + \cfrac{3}{28 - \cfrac{4}{65 + \cfrac{5}{126}}}}} = \frac{5797655}{3712466}$$

32.21 \xintiCstoF, \xintiGCtoF, \xintiCstoCv, \xintiGCtoCv

The same as the corresponding macros without the 'i', but for integer-only input. Infinitesimally faster; to notice the higher efficiency one would need to use them with an input having (at least) hundreds of coefficients.

32.22 \xintGCtoGC

f ⋆ `\xintGCtoGC{a+b/c+d/e+f/g+.....+v/w+x/y}` expands (with the usual meaning) each one of the coefficients and returns an inline continued fraction of the same type, each expanded coefficient being enclosed withing braces.

```
  {6}+\xintCstoF {2,-7,-5}/16}} \meaning\x
macro:->{1}+{3375/1[-3]}/{1/7}+{-3/5}/{720}+{67/36}/{16}
```

To be honest I have, it seems, forgotten why I wrote this macro in the first place.

This documentation has been compiled without the source code. To produce the documentation with the source code included, run "tex xint.dtx" to generate xint.tex (if not already available), then thrice latex on xint.tex and finally dvipdfmx on xint.dvi (ignore the dvipdfmx warnings; see also section 22).