The xintexpr and allied packages

Jean-François Burnol

jfbu (at) free (dot) fr

Part I.   The xintexpr package

Part II.   The macro layer for expandable computations: xintcore, xint, xintfrac,...

xintkernel

xinttools
xintcore
bnumexpr

xintbinhex
xintgcd
xint

xintfrac

xintseries
xintcfrac

xintexpr

polexpr

poormanlog
xintlog
xinttrig

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

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

polexpr is a separate (LaTeX only) package by the author which requires xintexpr.

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

poormanlog is a TeX and LaTeX package by the author which is loaded automatically by xintlog.

Part I.
The xintexpr package

1. Start here

- To use with `etex`, `pdftex`, ..., i.e. with TeX engines activating the eTeX extensions:

  ```
  \input xintexpr.sty
  \xinteval{sqrt(13, 60)}% get it with 60 digits (correctly rounded)
  ```

- To use with the LaTeX macro layer (`latex`, `pdflatex`, ...):

  ```
  \usepackage{xintexpr}
  % and here you have to wait for \begin{document}...
  % or rather you can start playing immediately:
  \typeout{\xinteval{sqrt(13, 60)}}
  ```

xintexpr is a package to do expandable computations, either exactly (i.e. with fractions) or in the sense of floating point numbers at an arbitrary (reasonable[1]) precision. It supports user definition of variables and functions. Its interface allows multiple comma separated expressions and nested structures:
```
\xinteval{1, 2, [3, [4, 5]], 6}
```
1, 2, [3, [4, 5]], 6 is impressive. No?

I am thinking about providing a console interface (using `rlwrap etex` as sadly the e-TeX binaries are not linked with the `readline` library) which would mimick sessions with Maple or Python.

For a quick impression of the package abilities, check (not yet, else you will never come back here) the tables of its built-in operators and functions.

The formatted source code is available in file sourcexint.pdf (texdoc sourcexint).

Warning: I don't have the time to maintain perfectly such large documentation. In preparing the 1.4 release I may have missed updating some bits which got randomly shuffled to new places (at least I did delete large sections, which was a hard decision to take, almost breaking the palimpsest quality of the document). Reports welcome.[2]

1.1. Features added since the 1.4 release

For bugfixes and possibly more details check CHANGES.html:

<div align="center">texdoc --list xint</div>

- The function zip().

- The function flat().

---

[1] Trigonometrical functions are currently supported only up to about 58 decimal digits, and generally speaking expansion starts taking really too much time at about 500 digits. It looks as if 100 digits is a reasonable upper limit for floating point (this is already a lot to fit on one single line of text) and future development of the package will not attempt to support floating point calculations efficiently beyond that limit. [2] Thanks to Jürgen Gilg for keeping the author motivated and helping proof-read the documentation.

- Chaining of comparison operators à la Python (no short-circuit, though) and l3fp.

- \xintPFloatE to specify like \xintFracToSciE does for \xinteval since 1.4 the separator to use between mantissa and exponent in the output of \xintfloateval output.

- \xintthespaceseparated (serves to provide suitable input to PS-Tricks \listplot).

- The optional argument [D] to \xintiexpr (or \xintieval) can be negative, with the same meaning as the non-negative case, i.e. rounding to an integer multiple of 1e-D (as formerly, for D positive the output uses fixed point notation with decimal digits and with D=0 the output is an integer with no decimal separator; with D<0 scientific notation is used[3]).

- The same applies to the functions trunc() and round(). And matching updates to \xintTrunc, \xintRound, \xintiTrunc, and \xintiRound.

- Support by add() and mul() for omit, abort and break().[4]

1.2. The 1.4 release of 2020/01/31

1.4 brought some new features (involving significant evolution of the xintexpr.sty source code) and a few (but important) breaking changes. See CHANGES.html which contains information which may not yet have been included into this PDF documentation.

The main new feature was (initial) support for nested structures. For a quick idea of already available related abilities check for example ndseq() or \xintdefufunc. See also \xintthealign. However, please grant the author a few decades to finish absorbing Python/NumPy.

The main breaking changes were:

- xintexpr 1.4 requires the \expanded primitive, which is provided by all major TeX engines since TeXLive 2019. The macro packages xint, xintfrac, xinttools et al. do not (yet) require \expanded.

  It is probable also \pdfstrcmp (\strcmp) will be required at some point but it has been provided by major TeX engines for a long time already.

- \xinteval (and \xintexpr) output does not use anymore the xintfrac ``raw'' format A/B[N]], rather it uses scientific notation AeN/B, dropping the exponent and/or denominator if they are respectively 0 and/or 1. This means that output can now be copied pasted directly to competing software on the market, such as Python or Maple. The output format of \xintfloatexpr (which uses macro \xintPFloat) was left un-modified although the prettifying done by it is not necessarily the best choice when displaying a nested structure via \xintthealign (perhaps next major release will reconsider that choice); and the way the zero value is output by \xintfloateval, currently 0. is yet to be chosen definitely. The used (*expandable*) macro for output can be specified by user.

- Syntax such as x*[a, b, c] or [a, b, c]+x for itemwise operation on «lists» has been (provisorily) dropped. Indeed, the brackets [...] are now genuine constructors of nestable structures, and implementing the feature (analogous to NumPy's concepts) will require overloading all scalar infix operators. Alternative already exist in the syntax for example seq(x*y, y = a,b,c). Actually in future x*[a, b, c] will be as [x*a, x*b, x*c] i.e. will keep the brackets, which prior to 1.4 on their own were no different from parentheses.

---

[3] This was introduced at 1.4a but due to a bug the e was by error of catcode 12, and \xintFracToSciE had no effect on it. Fixed at 1.4b   [4] Feature supposedly added at 1.4 but broken there.

1.3. License and installation instructions

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

Else, CTAN access provides xint.tds.zip which has all source code and documentation in a TDS-compliant archive, only waiting to be unzip -d <DIR> into some suitable hierarchical structure.

Else, etex xint.dtx extracts all source code. A Makefile is also provided with targets such as xint.pdf or sourcexint.pdf. Even if your system does not allow executing make, the rules it contains can be imitated manually (if possible using Latexmk).

Back to TeX distributions with a "texdoc" or similar utility,

```
texdoc --list xint
```

gives the choice to display one of:
- xint.pdf (this file),
- sourcexint.pdf (source code),
- README.md,
- CHANGES.html.

1.4. Printing big numbers on the page

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

```
\def\allowsplits #1{\ifx #1\relax \else #1\hskip 0pt plus 1pt\relax
                    \expandafter\allowsplits\fi}%
\def\printnumber #1{\expandafter\allowsplits \romannumeral-`0#1\relax }%
% \printnumber thus first ``fully'' expands its argument.
It may be used like this:
\printnumber{\xintiieval{100!^3}}\newline
```
81285103704665697929058034741394527800954175275203119077085794747670888482337305968567201883 75050478138776220712647125923141159206411609199354037545836490698436012619000519089702481351 07234498895796609463150334493880799668742586291763030205250590988746228607583652771623341365 91629009247695685942955467213561895127511100771737329147330105403484204308951158469957099274 14697054763835474153299936479805440000000000000000000000000000000000000000000000000000000000 00000000000000

The rendering here uses extra decoration.

2. xintexpr syntax reference and user guide

2.1. Oples and nut-ples: terminology for the 1.4 xint generation

Skip this on first reading, else you will never start using the package. SKIP THIS! (understood?)

In this section I will describe a mathematical terminology appropriate to understand the core functioning of the package in so far as it regards its numerical mode of operation. The description requires some adaptations to also cover the functioning during function declarations and this is not covered here.

We have *atoms*, which represent numeric data. In TeX syntax such *atoms* are always braced, more precisely, currently they look like {raw xintcore or raw xintfrac format within TeX braces}. Such TeX braces are not to be confused with set-theoretical braces: *atoms* are *elements* and not *sets*.

Our category C of «oples» is the smallest collection of *totally ordered finite sets* verifying these properties:

1. The empty set $\emptyset$ belongs to C.

2. Each singleton set whose element is an atom qualifies as an *ople*.

3. C is stable by concatenation.

4. If O is an *ople*, then the singleton {O} having O as unique element is also an *ople*.

Notes:

- We denote the empty set $\emptyset$ by *nil*. There is actually a built-in variable with this name. At 1.4, \xintexpr\relax is legal and also generates the *nil*.

- Concatenation is represented in the syntax by the comma. Thus repeated commas are like only one and nil is a neutral element.

- A singleton *ople* {atom} whose single element is an atom is called a *number*.

- The operation of constructing {O} from the *ople* O is called *bracing* (set theory, TeX), or *bracketing* (xintexpr input syntax, Python lists), or *packing* (as a reverse to Python's unpacking of sequence type objects).

- A braced *ople* is called a *nut-ple*. Among them {nil} is a bit special. It is called the *not-ple*. It is not nil!

It is perhaps important to reflect on the following:
The notation 3,5,7 can be seen in two distinct but related ways:

- each one of 3, 5, 7 is an *ople* (singleton) and 3,5,7 is their *union* or rather *concatenation* (order matters),

- or each one of 3, 5, 7 refers to an *atom* and 3,5,7 is an enumeration of the atoms of the *ople* it represents.

The second view is tempting, but recall that really the comma stands for *concatenation of totally ordered sets*, thus the first view is more correct. This first view maps to TeX notations where the value 3 is stored as {{3}}. But under \xintverbosetrue regime, the external brace pair, which is both a TeX brace pair and a set-theoretical notation gets removed. There only remains one, and what is shown is actually a view of an *atom*, where the braces are only TeX braces. But more complicated nested objects will have TeX braces representing also set-theoretical braces. If you are still here you can go on reading.

Each *ople* has a length which is its cardinality. The oples of length 1 are called *one-ples*. There are two types of *one-ples*:

- *numbers*,

- packed *oples*: the *nut-ples*.

As said before the *not-ple* {{}} is special. It can be input as []. Recall that a *number* as an *ople* is a singleton whose sole element is an *atom*. It is convenient to put the empty set nil on the same footing as *atoms*. Then the *not-ple* is analogous to an *empty number*.

We say that the empty set nil and *atoms* are *leaves*. Indeed, we can associate with any *ople* a tree. The root is the ople. In the case of the nil, there is nothing else than the root, which we then consider also a *leaf*. Else the children at top level are the successive items of the ople. Among the items some are *atoms* giving *leaves* of the tree, others are *nut-ples* which in turn have children. In the special case of the *not-ple* we consider it has a child, which is the empty set and this why we consider the empty set nil a *leaf*. We then proceed recursively. We thus obtain from the root *ople* a tree whose vertices are either *oples* or *leaves*. Only the empty set nil is both a *leaf* and an *ople*.

Considering the empty set nil as an *atom* fits with the xintexpr internal implementation based on TeX: nil is an empty pair of braces {}, whereas an *atom* is a braced representation of a numeric value using digits and other characters. We construct *oples* by putting one after the other such constituents and bracing them, and then repeating the process recursively.

Considering the empty set as an *atom* has also an impact on the definition of the *depth* (a.k.a as *maximal dimension*) of an *ople*. For example the *ople* {{}A_1A_2} with three elements, among them the empty set and two atoms is said to have depth 1, or to have maximal dimension 1. And {{∅}A_1A_2} is of depth 2 because it has a leaf (the empty set) which is a child of a child of the *ople*. NumPy *ndarrays* have a more restricted structure for example {{$A_{00}A_{01}$}{$A_{10}A_{11}$}} is a 2-dimensional array, where all leaves are at the same depth. When slicing empties the array from its atoms, NumPy keeps the shape information but prints the array as []. This will not be the case with xintexpr, which has no other way to indicate the shape than display it.

```
  \xinteval{[[],[]]}
```
[[], []]
```
  \xinteval{[[0,1],[10,11]][:,2:]}
```
[[], []]

«Set-theoretical» slicing of an *ople* means replacing it by a subset. This applies also if it is a *number*. Then it can be sliced only to itself or to the empty set (indeed it has only one element, which is an atom). Similarly the *not-ple* can only be sliced to give itself or the empty set. And more generally a *nut-ple* is a singleton so also can only be set-sliced to either the empty set or itself.

xintexpr extends «Python-like» slicing to act on *oples*:
- if they are not *nut-ples* set-theoretical slicing applies,
- if they are *nut-ples* (only case having a one-to-one correspondance in Python) then the slicing happens *within brackets*: i.e. the *nut-ple* is unpacked then the set-theoretical slicing is applied, then the result is *repacked* to produce a new *nut-ple*.

With these conventions the *not-ple* for example is invariant under slicing: unpacking it gives the empty set, which has only the empty set as subset and repacking gives back the *not-ple*. Slicing a general *nut-ple* returns a *nut-ple* but now of course in general distinct from the first one.

The syntax for Python slicing is to postfix a variable or a parenthesized ople with [a:b]. See subsection 2.9 for more. There are never any out-of-range errors when slicing or indexing. All operations are licit and resolved by the nil, a.k.a. empty set.

«Set-theoretical» item indexing of an *ople* means reducing it to a subset which is a singleton. It is thus a special case of set-theoretical slicing (which is the general process of selecting a subset as replacement of a set).

xintexpr extends «Python-like» indexing to act on *oples*:

- if they are not *nut-ples* set-theoretical item indexing applies,
- if they are *nut-ples* (only case having a one-to-one correspondance in Python) then the meaning becomes *extracting*: i.e. the *nut-ple* is unpacked then the set-theoretical indexing is applied, but the result is *not repacked*.

For example when applied to the *not-ple* we always obtain the nil. Whereas as we saw slicing the *not-ple* always gives back the *not-ple*. Indexing is denoted in the syntax by postfixing by [N]. Thus for *nut-ples* (which are analogous to Python objects), there is genuine difference between the [N] extractor and the [N:N+1] slicer. But for *oples* which are either nil, a *number*, or of length at least 2, there is no difference.

Nested slicing is a concept from NumPy, which is extended by xintexpr to trees of varying depths. We have a chain of slicers and extractors. I will describe only the case of slicers and letting them act on a nut-ple. The first slicer gives back a new nut-ple. The second slicer will be applied to each of one of its remaining items. However some of them may be *atoms* or the empty set. In the NumPy context all leaves are at the same depth thus this can happen only when we have reached beyond the last dimension (axis). This is not permitted by NumPy and generates an error. xintexpr does not generate an error. But any attempt to slice an *atom* or the empty set (as element of its container) removes it. Recall we call them *leaves*. We can not slice leaves. We can only slice non-leaf items: such items are necessarily nut-ples. The procedure then applies recursively.

If we handle an extractor rather than a slicer, the procedure is similar: we can not extract out of an *atom* or the empty set. They are thus removed. Else we have a nut-ple. It is thus unpacked and replaced by the selected item. This item may be an atom or the empty set and any further slicer or extractor will remove them, or it is a nut-ple and the procedure applies with the next slicer/extractor.

xintexpr allows to apply such a [a:b,c:d,N,e:f,...] chain of slicing/extracting also to an *ople*, which is not a *nut-ple*. We simply apply the first step as has been described previously and successive steps will only get applied to either *nut-ples* or *leaves*, the latter getting silently removed by any attempted operation.

One last thing. In the syntax of xintexpr, variables as well as functions have a name and a value. The value is an ople. We can always use a variable whose value is an ople in a function call, it will occupy the place of as many arguments as its length indicates. But in a function declaration, the variables must stand for one-ples, i.e. either numbers or nut-ples.

The * unpacks a nut-ple. The last positional argument in a function declaration can have a special form *⟨name⟩. This means that ⟨name⟩ is a nut-ple which receives as items all arguments in the function call beyond the first ones corresponding to the function declaration.

In case things were too clear, let's try to add a bit of confusion with an extra word on *leaves*. When we discuss informally (particularly to compare with NumPy) an input such as

```
[[1, 2], [3, 4]]
```

we may well refer to 1, 2, 3, and 4 as being «the leaves of the 2d array». But obviously we have here numbers and previously we explained that a number is not a *leaf*, its *atom* is. Well, the point here is that we must make a difference between the input form as above and the actual constructed *ople* the parser will obtain out of it. In the input we do have numbers. The comma is a *concatenator*, it is not a separator for enumeration! The *ople* which corresponds to it has a TeX representation like this:

```
{{{1}{2}}{{3}{4}}}
```

where we don't have the *numbers* anymore (which would look like {{1}}, {{2}}, ...) but numeric *atoms* {1}, {2}, {3}, {4} where the braces are TeX braces and **not** set-theoretical braces (the other braces are both). Hence we should see the above as the ople {{A₀₀A₀₁}{A₁₀A₁₁}} with atoms A_{00} = {1}, ..., being the *leaves* of the tree associated to (or which is) the *ople*.

Numbers may be called the *leaves* of the **input**, but once parsed, the input becomes an *ople* which is (morally) a tree whose leaves are *atoms* (and the empty set).

I hope this is clear to everyone. If not, maybe time to say this section was absolutely not needed to understand the rest of the manual, but I needed to write it for my own satisfaction. Believe me, you need this section if you want to write the underlying software!

2.2. The three parsers

xintexpr provides three numerical expression parsers and two subsidiary ones. They are designed to be compatible with expansion only context. All computations ultimately rely on (and reduce to) usage of the \numexpr primitive from ε-TeX[5]. These ε-TeX extensions date back to 1999 and are by default incorporated into the pdftex etc... executables from major modern TeX installations for more than fifteen years now.

- \xinteval{⟨*expression*⟩} handles integers, decimal numbers, numbers in scientific notation and fractions. The algebraic computations are done *exactly*, and in particular / simply constructs fractions. Use // for floored division.

  ```
  \xinteval{add(x/(x+1), x = 1000..1014)}\par
  ```
  4648482709767835886400149017599415343/310206597612274815392155150733157360

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

  > By default, basic operations on fractions do not automatically reduce to smallest terms the output: A/B multiplied by C/D returns AC/BD, and A/B added to C/D uses lcm(B, D) as denominator.

  Arbitrarily long numbers are allowed in the input. The space character (contrarily to the situation inside \numexpr) and also the underscore character (as allowed in Python too) can serve to separate groups of digits for better readability. But the package currently provides no macros to let the output be formatted with such separators.

  Formatting of numeric output is apart from some minimal facilities such as \xintFrac, \xintDecToString, \xintPRaw, \xintFracToSci or \xintPFloat left to user macros or third-party packages[6].

  ```
  \xinteval{123_456_789_012^5}
  ```
  28679718616935524442942783005582105858543331562763768832

- \xintiieval{⟨*expression*⟩} does exact computations *on (big) integers only*. It is (of course) slightly faster than \xinteval for equivalent operations. The forward slash / does the *rounded* integer division to match behaviour of \numexpr. The // operator does floored division as in \xinteval. The /: is the associated modulo operator (we could easily let the catcode 12 % character be an alias, but using such an unusual percent character would be a bit cumbersome in a TeX workflow, if only for matters of syntax highlighting in TeX-aware text editors).

---

[5] It can handle only integers, and they must be at most $2^{31} - 1$ = 2147483647. Thus some work has to be done to handle arbitrarily big integers or arbitrary float precision.   [6] For example I hesitated whether to let \xintFracToSciE be actually a macro with one mandatory argument as this would give a hook to customize formatting the scientific exponent. But then, why not also wrap the mantissa or the denominator in hook macros? and should the / or the decimal separator also be customizable? It was reasonable to provide a way to use E in place of e for the scientific part, as E is accepted input in \xinteval or \xintfloateval. It looked however better to leave additional formatting to external utilities.

```
    \xintiieval{add((i/:7)?{omit}{i^5}, i=1000..1020)}% only add fifth powers of multiples of 7
3122939154402144
```

- `\xintfloateval{⟨expression⟩}` does floating point computations with a given precision P, as specified via a prior assignment `\xintDigits:=P\relax`. The `/` will compute the correct rounding of the exact fraction. Again `//` is floored division and `/:` its associated modulo (see also `divmod()`).

  ```
  \begingroup
  \xintDigits:=64\relax
  \xintfloateval{sqrt(3)}
  \endgroup
  ```
  1.732050807568877293527446341505872366942805253810380628055806979

  The default is with P=16 digits. The four basic operations and the square root realize *correct rounding.*[7]

  It can be used with an optional argument [Q] which means to do a final float rounding to mantissas of Q digits (this makes sense only if Q<P). ATTENTION: the optional argument [Q] is to be located *within* the braces at the start of the expression.

  When Q is negative it means to round to P+Q digits only. Current implementation of trigonometrical functions (`xinttrig`) is provisory and does not use guard digits, using [-2] will trim the last two, probably wrong, digits.

  On output, `\xintfloateval` uses `\xintPFloat` for each number. This can be modified (cf. `\xintfloatexprPrintOne`).

The user can define variables and functions. Definition of functions is either per parser (`\xintdeffunc`, `\xintdeffloatfunc`, ...), but there are some restrictions, or generic (`\xintNewFunction`) but the latter is only syntactic sugar for function-like disguise of a TeX macro having not done any pre-parsing.

Two derived parsers:

- `\xintieval{⟨expression⟩}` does all computations like `\xinteval` but rounds the result to the nearest integer. If there is an optional positive argument [D], the rounding is to:

  - if D>0: the nearest fixed point number with D digits after the decimal mark,
  - if D=0: the nearest integer,
  - New with 1.4a if D<0: the nearest multiple of 10^(-D) (this case is new with 1.4a and uses scientific notation).

  ATTENTION: the optional argument [D] is to be located *within* the braces at the start of the expression.

- `\xinttheboolexpr⟨expression⟩\relax` does all computations like `\xinteval` then converts all (non-empty) leaves[8] to True or False (cf. `\xintboolexprPrintOne`). There is no `\xintbooleval`.

These macros are wrappers for a more core syntax:
- `\xintexpr⟨expression⟩\relax`,
- `\xintiiexpr⟨expression⟩\relax`,
- `\xintfloatexpr⟨expression⟩\relax`,
- `\xintiexpr⟨expression⟩\relax`,
- `\xintboolexpr⟨expression⟩\relax`.

New with 1.4 This core syntax can be used directly in typesetting flow. In an `\edef` they expand to some braced nested data (all computations having been done) prefixed with some `\protected` «typesetter» macros. When using `\xinteval` (in contrast to `\xintexpr`), the protection of the «typesetter»

---

[7] when the inputs are already floating point numbers with at most P-digits mantissas.   [8] Currently, empty leaves are output using `\xintexprEmptyItem`, i.e. default to []. This may change.

10

is removed and its action gives (expandably, in two steps) explicit digits and other characters such as those of scientific notation or brackets.

It is possible to use the core syntax \xintexpr⟨*expression*⟩\relax also in so-called moving arguments, because when written out to a file the final expansion result only standard catcodes and thus will get retokenized and the typesetter macro (which being \protected is there intact in external file) will expand as expected.

One needs \xinteval et al. only if one really wants the final digits (and other characters), for example in a context where TEX expects a number or a dimension.

As alternative to \xinteval{⟨*expression*⟩}, an equivalent is \xintthe\xintexpr⟨*expression*⟩\relax. Similarly \xintthe can prefix all other core parsers. And one can also use \xinttheexpr as shortcut for \xintthe\xintexpr.

Throughout this documentation I will most of the time refer to \xinteval and \xintexpr. But beware that doing exact computations with fractions leads very quickly to very big results (and furthermore one needs to use explicitly the reduce() function to convert the fractions into smallest terms). Thus most probably what you want is \xintfloateval and \xintfloatexpr.

2.3. Expansion

As mentioned already, the parsers are compatible with expansion-only context.

Also, they expand the expression piece by piece: the normal mode of operation of the parsers is to unveil the parsed material token by token. Unveiling is a process combining brace removal (one level generally) and *f-expan*sion.

For example a closing parenthesis does not have generally to be immediately visible, it may arise later from expansion. Even the ending \relax may arise from expansion. Even though the \xinteval user interface means that the package has at some point the entire expression in its hands, it immediately re-inserts it into token stream with an additional postfixed \relax and from this point on has lost any ways to manipulate formally again the whole thing; it can only re-discover it piece per piece.

This general behaviour has significative exceptions mostly related to «pseudo»-functions. A «pseudo»-function will grab some of its arguments via delimited macros. For example subs(expr1,x =expr2) needs to see the comma, equal sign and closing parenthesis. But it has mechanisms to allow expr1 and expr2 to possess their own commas and parentheses.

Inner semi-colons on the other hand currently always can originate from expansion. Defining functions or variables requires a visible semi-colon acting as delimiter of the expression, but inner semi-colons do not need to be hidden within braces or macros.

The expansion stops only when the ending \relax has been found (it is then removed from the token stream).

For catcode related matters see \xintexprSafeCatcodes.

A word of warning on the bracketed optional argument of respectively \xintfloatexpr and \xintiexpr. When defining macros which will hand over some argument to one of these two parsers, the argument may potentially start with a left square bracket [(e.g. argument could be [1, 2, 3]) and this will break the parser. The fix is to use in the macro definition \xintfloatexpr\empty. This extra \empty token will prevent the parser thinking there is an optional argument and it will then disappear during expansion.

If comparing to other languages able to handle floating point numbers or big integers, such as Python, one should take into account that what the xint packages manipulate are streams of ascii bytes, one per digit. At no time (due to expandability) is it possible to store intermediate results in an arithmetic CPU register; each elementary operation via \the\numexpr will output digit tokens (hence as many bytes), not things such as handles to memory locations where some numbers are stored as memory words. The process can never put aside things but can only possibly permute them with upcoming tokens, to use them later, or, via combinations of \expanded and \unexpanded or some other more antiquated means grab some tokens and shift the expansion to some distant locations to later come back. The process is a never-ending one-dimensional one...

2.4. \xintthealign and its customization

With \xintthealign one can get nested data use a TEX alignment in the output. Attention, this must be followed by \xintexpr et al., never by \xinttheexpr or \xinteval. Here is an example :

```
\xintthealign\xintexpr ndseq(1/(i+j), i = 1..10; j=1..10)\relax
```
```
[[ 1/2,   1/3,   1/4,   1/5,   1/6,   1/7,   1/8,   1/9,   1/10,  1/11  ],
 [ 1/3,   1/4,   1/5,   1/6,   1/7,   1/8,   1/9,   1/10,  1/11,  1/12  ],
 [ 1/4,   1/5,   1/6,   1/7,   1/8,   1/9,   1/10,  1/11,  1/12,  1/13  ],
 [ 1/5,   1/6,   1/7,   1/8,   1/9,   1/10,  1/11,  1/12,  1/13,  1/14  ],
 [ 1/6,   1/7,   1/8,   1/9,   1/10,  1/11,  1/12,  1/13,  1/14,  1/15  ],
 [ 1/7,   1/8,   1/9,   1/10,  1/11,  1/12,  1/13,  1/14,  1/15,  1/16  ],
 [ 1/8,   1/9,   1/10,  1/11,  1/12,  1/13,  1/14,  1/15,  1/16,  1/17  ],
 [ 1/9,   1/10,  1/11,  1/12,  1/13,  1/14,  1/15,  1/16,  1/17,  1/18  ],
 [ 1/10,  1/11,  1/12,  1/13,  1/14,  1/15,  1/16,  1/17,  1/18,  1/19  ],
 [ 1/11,  1/12,  1/13,  1/14,  1/15,  1/16,  1/17,  1/18,  1/19,  1/20  ]]
```

Changed at 1.4a! It is possible to customize the behaviour of \xintthealign. The helper macros, apart from \xintexpralignbegin and \xintexpralignend will be subjected to a complete (\expanded) expansion (once).[9] The package uses here \protected with no strong reason, as the replacement tokens are not expanding anyhow, but the idea is that this allows to define a macro in an \edef and later change the meaning of the auxiliary macros depending on what one wants to do with the expansion result. See also further down the LATEX example with a matrix environment, where \noexpand rather than \protected is used.

```
\protected\def\xintexpralignbegin      {\halign\bgroup\tabskip2ex\hfil##&&##\hfil\cr}%
\protected\def\xintexpralignend        {\crcr\egroup}%
\protected\def\xintexpralignlinesep    {,\cr}% separates "lines"
\protected\def\xintexpralignleftsep    {&}% at left of first item in a "line" (after brackets)
\protected\def\xintexpraligninnersep   {,&}% at the left of non-first items
\protected\def\xintexpralignrightsep   {&}% at right of last item in a "line" (before brackets)
\protected\def\xintexpralignleftbracket {[}%
\protected\def\xintexpralignrightbracket{]}%
```

Unstable! Although we will try to keep stable the way «regular arrays» are rendered, the \xintthealign macro (and its associated customizability) is considered work-in-progress and may experience breaking changes.

Use for example this for outputting to a file or a terminal:

```
% Better here without \protected.
% We assume here \newlinechar has the LaTeX setting.
\def\xintexpralignbegin      {}%
\def\xintexpralignend        {}%
\def\xintexpralignlinesep    {,^^J}% separates "lines"
\def\xintexpralignleftsep     { }% at left of first item in a "line" (after brackets)
\def\xintexpraligninnersep    {, }% at the left of non-first items
\def\xintexpralignrightsep    { }% at right of last item in a "line" (before brackets)
\def\xintexpralignleftbracket {[}%
\def\xintexpralignrightbracket{]}%
```

And here is an example using a pmatrix environment. But it will not break across pages, contrarily to the display produced by the default \xintthealign configuration which uses TEX's \halign.

```
\[
\def\xintexpralignbegin      {\begin{pmatrix}}%
\def\xintexpralignend        {\end{pmatrix}}%
\def\xintexpralignlinesep    {\noexpand\\}% needed to counteract an internal \expanded
\def\xintexpraligninnersep   {&}%
\let\xintexpralignleftbracket\empty  \let\xintexpralignleftsep\empty
\let\xintexpralignrightbracket\empty \let\xintexpralignrightsep\empty
% by default amsmath matrices can have 10 columns at most
```

---

[9] \xintexpralignend is expanded once, after the body has been submitted to exhaustive expansion, and prior to the expansion of \xintexpralignbegin.

```
% (cf amsmath documentation for what to do to allow more)
l.c.m.=\xintthealign\xintiiexpr  ndmap(lcm, 1..12; 1..10)\relax
\]
```

$$
\text{l.c.m.} = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
2 & 2 & 6 & 4 & 10 & 6 & 14 & 8 & 18 & 10 \\
3 & 6 & 3 & 12 & 15 & 6 & 21 & 24 & 9 & 30 \\
4 & 4 & 12 & 4 & 20 & 12 & 28 & 8 & 36 & 20 \\
5 & 10 & 15 & 20 & 5 & 30 & 35 & 40 & 45 & 10 \\
6 & 6 & 6 & 12 & 30 & 6 & 42 & 24 & 18 & 30 \\
7 & 14 & 21 & 28 & 35 & 42 & 7 & 56 & 63 & 70 \\
8 & 8 & 24 & 8 & 40 & 24 & 56 & 8 & 72 & 40 \\
9 & 18 & 9 & 36 & 45 & 18 & 63 & 72 & 9 & 90 \\
10 & 10 & 30 & 20 & 10 & 30 & 70 & 40 & 90 & 10 \\
11 & 22 & 33 & 44 & 55 & 66 & 77 & 88 & 99 & 110 \\
12 & 12 & 12 & 12 & 60 & 12 & 84 & 24 & 36 & 60
\end{pmatrix}
$$

2.5. Customization of typesetting of individual items

The way individual items are formatted (whether or not using \xintthealign) is also customizable:

```
\def\xintexprEmptyItem{[]}
\let\xintexprPrintOne\xintFracToSci
\def\xintFracToSciE{e}
\def\xintiiexprPrintOne #1{#1}
\def\xintfloatexprPrintOne#1#2{\xintPFloat[#1]{#2}}
\def\xintPFloatE{e}
\def\xintboolexprPrintOne#1{\xintiiifNotZero{#1}{True}{False}}
```

Attention! The above macros convert from xintexpr internal numeric data format to «printed» output; they are thus susceptible to require adjustments if the internal data format changes, which may happen at each release. Of course the default for \xintexprPrintOne etc... will be adjusted accordingly, but user custom definitions may break.

   Currently, this means that the macros used in place of \xintFracToSci and \xintPFloat must understand both the raw xintfrac format A/B[N] and the decimal format A.ddddd..dd.[10] The typesetter for \xintiiexpr simply prints ``as is'', but this may change in future.

   The used macros must be compatible with expansion-only context, but do not have to be *f-expan*dable.

   Note: when not using \xintthealign, output of nested structures uses left and right brackets, and commas and spaces in a non-customizable way, except via \xintexprEmptyItem. Use the \xintthealign interface for full customizability.

2.6. Built-in operators and their precedences

The entries of Table 1 are hyperlinked to the more detailed discussion at each level.

∞ At this highest level of precedence, one finds:

   functions and **variables** Functions (even the logic functions !() and ?() whose names consist of a single non-letter character) must be used with parentheses. These parentheses may arise from expansion after the function name is parsed (there are exceptions which are documented at the relevant locations.)

   * Python-like «unpacking» prefix operator. Sometimes one needs to use it as function *() (but I can't find an example right now) but most of the time parentheses are unneeded.

---

[10] Furthermore \xintieval{[D]...} with a *negative* D (feature added at 1.4a) relies on the \xintexprPrintOne ability to react to a catcode 12 e (it is always used with detokenized input). \xintFracToSci has this ability and uses catcode 11 e in output (or rather whatever \xintFracToSciE expands to).

∞: at this top level the syntax elements whose execution
is done prior to operators preceding them:
  • built-in or user-defined functions,
  • variables,
  • the * unpacking operator,
  • and intrinsic constituents of numbers: decimal mark .,
    e and E of scientific notation, hexadecimal prefix ".

Precedence	``Operators'' at this level
20	postfix ! and branching ?, ?? operators
-	minus sign as unary operator inherits the precedence of the infix operator it follows, if that precedence is higher than the one of binary + and -, else it inherits the latter
18	^ and ** are a priori synonymous (but see xintlog)
16	Tacit multiplication has an elevated precedence
14	*, /, // (floored division), and /: (associated modulo, alias 'mod')
12	+, -
10	<, >, ==, <=, >=, != (they can be chained)
8	Boolean conjunction && and its alias 'and'
6	Boolean disjunction || and its alias 'or'. Also 'xor' and .., ..[,].., and : have this precedence
4	the brackets for slicers and extractors [,]
3	the comma ,
2	the bracketers [,] construct nestable «arrays»
1	the parentheses (,), and the semi-colon ; in iter(), rseq(), and further structures

  • Actually operators have a left and a right precedence,
    which for most coincide. But for some there is a crucial
    distinction. The above table is indicative, and the ac-
    tual numerical levels used internally may change.
  • In case of equal precedence, the rule is left-
    associativity: the first encountered operation is
    executed first. Tacit multiplication has an ele-
    vated precedence level hence seemingly breaks left-
    associativity: (1+2)/(3+4)5 is computed as (1+2)/((3
    +4)*5) and x/2y is interpreted as x/(2*y) when using
    variables.

Table 1: Precedence levels

. is decimal mark; the number scanner treats it as an inherent, optional and unique component of a being formed number. `\xintexpr 0.^2+2^.0\relax` is interpreted as `0^2+2^0` and thus produces `1`.

Since release `1.2` an isolated decimal mark is illegal input in the xintexpr parsers (it remains legal as argument to the macros of xintfrac).

e scientific notation.

E scientific notation. For output, see `\xintFracToSciE`.

" prefix for hexadecimal input. Only uppercase letters, and one optional . separating integer and fractional hexadecimal parts. This functionality

requires to load explicitly package xintbinhex.

```
\xintexpr "FEDCBA9876543210\relax\newline
\xintexpr ".FEDCBA9876543210\relax\newline
\xintexpr 16^5-("F75DE.0A8B9+"8A21.F5746+16^-5)\relax
```
18364758544493064720
9955555555555555555941049661328179354313760995864868164062500000e-64
0

It is possible that in future the " prefix could be dropped in favour of `0x` prefix. This would free " to be used for input of «string»-like entities.

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

! computes the factorial of an integer. Attention that the boolean equality test `==` confuses the parser if following directly `!` (e.g. `3! == 10`) as spaces are ignored and `!=` will be intepreted as boolean inequality test, the second `=` causing then a low-level error. Use parentheses in such cases: `(3!)==10`.

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

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

- As unary operator, the minus sign inherits the precedence of the infix operator it follows (plus signs as unary operators are simply ignored).

```
\xintexpr -3-4*-5^-7, (-3)-(4*(-(5^(-7))))\relax\newline
\xintexpr -3^-4*-5-7, (-((3^(-4))*(-5)))-7\relax\newline
|2^-10| gives \xintexpr 2^-10\relax\space
```
-234371/78125, -234371/78125
-562/81, -562/81
2^-10 gives 1/1024 and is thus perfectly legal, no need for parentheses.

Note (`1.4b`): the above is what this documentation has always said, but it has also always been only partially true. I.e. it applies only when `-` follows an infix binary operator having at least the precedence level of `+` and `-`. When the unary `-` follows an infix operator (or operator word) of less precedence, its precedence will be set to the one for the infix operators `+` and `-`. «Seul sourcexint.pdf fait foi».

18

^

** Both compute powers in left associative way.

```
\xintiiexpr 2^2^3\relax
```

64

Half-integer exponents are allowed in `\xintfloateval` and use `sqrt`(). It is possible to allow arbitrary fractional exponents (subsection 4.1) but this currently achieves only a reduced precision. See `\xintFloatPower` and `xintlog` for additional information.

16 see Tacit multiplication.

14

`*` multiplication

`/` division: exact in `\xinteval`, correctly rounded in `\xintfloateval` (numerator and denominator are rounded before the division is done), and rounded to an integer (like `\numexpr` does: half-integers are rounded towards infinity of same sign) in `\xintiieval`. The division is left-associative:

```
\xintexpr reduce(100/50/2)\relax
```
1

`//` floored division

`/:` the associated modulo

Left-associativity applies generally to operators of same precedence.

```
\xintexpr 100000/:13, 100000 'mod' 13\relax\newline
\xintexpr 100000/:13/13\relax
```
4, 4
4/13

'mod' is same as `/:`.

Note: The enclosing (right) ticks are mandatory part of all such infix operator «words».

12

`+` addition

`-` subtraction. According to the general left-associativity rule in case of equal precedence, it is left associative:

```
\xintiiexpr 100-50-2\relax
```
48

10 Comparison operators are (as in Python) all at the same level of precedence, use parentheses for disambiguation.

`<` a<b evaluates to 1 if the strict inequality holds to 0 if not.

`>` a>b evaluates to 1 if the strict inequality holds to 0 if not.

`==` a==b evaluates to 1 if equality holds to 0 if not.

`<=` a<=b evaluates to 1 if left hand side is at most equal to right hand side, to 0 if not.

`>=` a>=b evaluates to 1 if left hand side is at least equal to right hand side, to 0 if not.

`!=` a!=b evaluates to 1 if they differ, to 0 if not.

New with 1.4b Comparisons can be chained arbitrarily, e.g., x < y <= z != t is equivalent to x < y 'and' y <= z 'and' z != t (and also to all(x<y, y<=z, z!=t)), except that if y and z involve computations, they are evaluated only once. Currently there is no short-circuit here, i.e. even if some intermediate comparison turns out false (in fact 0), all the remaining conditionals will still be evaluated.

```
\xintifboolexpr{1<=2!=3<4>1}{true}{\error}, \xintifboolexpr{1<=2>=3<4>1}{\error}{false}
```
true, false

8

&& logical conjunction. Evaluates to 1 if both sides are non-zero, to 0 if not.

'and' idem. The (right) ticks are mandatory. See also the all() multi-arguments function.

6

|| logical (inclusive) disjunction. Evaluates to 1 if one or both sides are non-zero, to 0 if not.

'or' idem. See also the any() multi-arguments function.

'xor' logical (exclusive) disjunction. See also the xor() multi-arguments function.

..

..[

**].. ** Syntax for arithmetic progressions. See subsection 2.8.

**: ** This is a separator involved in [a:b] Python-like slicing syntax.

4

[

**] ** Involved in Python-like slicing [a:b] and extracting [N] syntax. And its extension à la NumPy [a:b,N,c:d,...,:]. Ellipsis ... is not yet implemented. The «step» parameter as in [a:b:step] is not yet implemented.

3

**, ** The comma separates expressions (or function arguments).[11]

```
\xintiiexpr 2^3,3^4,5^6\relax
```
8, 81, 15625

2

[

**] ** The bracketers construct nestable «array-like» structures. Arbitrary (heterogeneous) nesting is allowed. For output related matters see \xintthealign (its usage is optional, without it rendering is «one-dimensional»). Output shape of non-homogeneous arrays is to be considered unstable at this time.

1

(

**) ** The parentheses serve as mandatory part of the syntax for functions, and to disambiguate precedences.[12] They do not construct any nested structure.

**; ** The semi-colon as involved as part of the syntax of iter(), rseq(), ndseq(), ndmap() has the same precedence as a closing parenthesis.

\relax This is the expression terminator for \xintexpr et al. It may arise from expansion during the parsing itself. As alternative use \xinteval et al. which proceed as macros expecting one mandatory argument.

The ; also serves as syntax terminator for \xintdefvar and \xintdeffunc. It can in this rôle not arise from expansion as the expression body up to it is fetched by a delimited macro. But this is done in a way which does not require any specific hiding for inner semi-colons as involved in the syntax of iter(), etc...

---

[11] The comma is really like a binary operator, which may be called "join". It has lowest precedence of all (apart the parentheses) because when it is encountered all postponed operations are executed in order to finalize its *first* operand; only a new comma or a closing parenthesis or the end of the expression will finalize its *second* operand. [12] It is not apt to describe the opening parenthesis as an operator, but the closing parenthesis is analogous to a postfix unary operator. It has lowest precedence which means that when it is encountered all postponed operations are executed to finalize its operand. The start of this operand was decided by the opening parenthesis.

2.7. Built-in functions

See Table 2 whose elements are hyperlinked to the corresponding definitions.

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

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

Table 2: Functions (click on names)

Miscellaneous notes:

☞ {
- since release 1.3d gcd() and lcm() are extended to apply to fractions too, and do NOT require the loading of xintgcd,
- The randomness related functions random(), qrand() and randrange() require that the TeX engine provides the \uniformdeviate or \pdfuniformdeviate primitive. This is currently the case for pdftex, (u)ptex, luatex, and also for xetex since TeXLive 2019.
- togl() is provided for the case etoolbox package is loaded,
- bool(), togl() use delimited macros to fetch their argument and the closing parenthesis must be explicit, it can not arise from on the spot expansion. The same holds for qint(), qfrac(), qfloat(), qraw(), random() and qrand().
- Also functions with dummy variables use delimited macros for some tasks. See the relevant explanations there.
- Functions may be called with *oples* as arguments as long as the total length is the number of arguments the function expects.

2.7.1. Functions with no argument

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

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

```
\pdfsetrandomseed 12345
\xintDigits:=37\relax
\xintthefloatexpr random()\relax\newline
\xintthefloatexpr random()\relax\par
```
0.2415544817596207455547929850209500042
0.2584863529993996627285461554203021352

qrand() returns a random float 0 <= x < 1 using 16 digits of precision (i.e. 10^{16}x is an in-
teger). This is provided when speed is a at premium as it is optimized for precision being
precisely 16.

```
% still with 37 digits as prevailing float precision
\xintthefloatexpr qrand(), random()\relax\newline
\xintDigits:=16\relax
\xintthefloatexpr qrand(), random()\relax\par
```
0.4883568991327765000000000000000000000, 0.09165461826072383107532471669335645230
0.9069127435402274, 0.9106687541716861

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

See further randrange(), which generates random integers.

Currently there is no uniform() function[13] but it can be created by user:

```
\xintdeffloatfunc uniform(a, b):= a + (b-a)*random();
\romannumeral\xintreplicate{10}%
{%
    \xintthefloatexpr uniform(123.45678, 123.45679)\relax\newline
}%
```
123.4567849497100
123.4567812033226
123.4567863308250
123.4567896366777
123.4567849656655
123.4567849908270
123.4567889123433
123.4567896262979
123.4567846543719
123.4567832664043

New with 1.4 **rbit()** returns a random 0 or 1.

2.7.2. Functions with one argument

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

```
\xinttheexpr num(3.1415^20), num(1e20)\relax
```
8764785276, 100000000000000000000 The output is an explicit integer with as many zeros are
as necessary. Even in float expressions, there will be an intermediate stage where all needed

---

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

digits are there, but then the integer is immediately reparsed as a float to the target precision, either because some operation applies to it, or from the output routine of \xint-floatexpr if it stood there alone. Hence, inserting something like num(1e10000) is costly as it really creates ten thousand zeros, even though later the whole thing becomes a float again. On the other hand naturally 1e10000 without num() would be simply parsed as a floating point number and would cause no specific overhead.

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

```
\xintthefloatexpr frac(-355/113), frac(-1129.218921791279)\relax
```
-0.1415929203539820, -0.2189217912790000

reduce(x) reduces a fraction to smallest terms

```
\xinttheexpr reduce(50!/20!/20!/10!)\relax
```
1415997888807961859400

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

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

```
\xinttheexpr preduce(10e3/2), reduce(10e3/2)\relax
```
5e3, 5000

abs(x) absolute value

sgn(x) sign. See also \xintifsgnexpr.

inv(x) inverse.

floor(x) floor function.

ceil(x) ceil function.

sqr(x) square.

ilog10(x) in \xintiiexpr the integer exponent a such that $10^a \leq abs(x) < 10^{a+1}$; returns (this may evolve in future) -2147450880 if x vanishes (i.e. 0x7fff8000).

```
\xintiieval{ilog10(1), ilog10(-1234567), ilog10(-123456789123456789), ilog10(2**31)}\par
```
0, 6, 17, 9

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

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

sqrtr(x) available *only* in \xintiiexpr, rounded square root.

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

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

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

not(x) logical not.

even(x) is the evenness of the truncation num(x).

```
\xintthefloatexpr [3] seq((x,even(x)), x=-5/2..[1/3]..+5/2)\relax
```

-2.50, 1.00, -2.17, 1.00, -1.83, 0., -1.50, 0., -1.17, 0., -0.833, 1.00, -0.500, 1.00, -0.167, 1.00, 0.167, 1.00, 0.500, 1.00, 0.833, 1.00, 1.17, 0., 1.50, 0., 1.83, 0., 2.17, 1.00, 2.50, 1.00

odd(x) is the oddness of the truncation num(x).

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

isint(x) evaluates to 1 if x is an integer, to 0 if not. See ifint().

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

isone(x) evaluates to 1 if x is 1, to 0 if not. See ifone().

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

qint(x) belongs with qfrac(), qfloat(), qraw() to a special category:

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

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

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

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

   ```
   \def\x{....many many many ... digits}\def\y{....also many many many digits...}
   \xinttheiiexpr qint(\x)*qint(\y)+qint(\y)^2\relax\par
   ```

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

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

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

21

```
   \xintthefloatexpr 1234567812345679/1234567812345678-0.5\relax\newline
   \xintthefloatexpr \xintexpr12345678123456785001/12345678123456784999\relax-0.5\newline
```
0.5000000000000000
0.5000000000000010
0.5000000000000010
0.5000000000000000

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

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

New with 1.4 **float_(x)** is like float() but does not accept the latter second optional argument. This is provided as a utility to speed-up things in some contexts, particularly when converting function definitions done via \xintdeffunc (see explanations there) to functions usable in \xintfloateval.

Do not use! (1.4) **nuple(x)** is currently same as [...]. Reserved for possible alternative meaning in future.

```
   \xinteval{nuple(1,2,3)}
```
[1, 2, 3]

New with 1.4 **unpack(x)** is alternative for * unpacking operator.

```
   \xinteval{unpack([1,2,3])}
```
1, 2, 3

New with 1.4b **flat(ople)** removes all nesting to produce a (non-bracketed) ople having the same leaves (some possibly empty) but located at depth 1.

```
   \xinteval{flat([[[[1,[],3],[4,[[[5,6,[]],[8,9],[[],11]],12],[13,14]]], [[],16]]], [])}
```
1, [], 3, 4, 5, 6, [], 8, 9, [], 11, 12, 13, 14, [], 16, []

unstable? I almost delayed indefinitely release because I was hesitating on the name: perhaps better with flattened(), but long names add (negligible, but still) overhead compared to short names. For this reason, consider that name may change.

2.7.3. Functions with an alphanumeric argument

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

```
   \xintifboolexpr{25*4-if(bool(mmode),100,75)}{YES}{NO}
```
will return NO if executed in math mode (the computation is then 100-100 = 0) and YES if not (the if() conditional is described below; the \xintifboolexpr test automatically encapsulates its first argument in an \xintexpr and follows the first branch if the result is non-zero (see subsection 15.14)).

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

togl(name) returns 1 if the LaTeX package etoolbox[14] 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

---

[14] https://ctan.org/pkg/etoolbox

type ``ERROR: Missing `\endcsname` inserted.'', with further information saying that `\protect` should have not been encountered (this `\protect` comes from the expansion of the non-expandable `etoolbox` error message).

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

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

A straight `\ifsometest{YES}{NO}` would do the same more efficiently, the point of `\ifsometest10` is to allow arbitrary boolean combinations using the (described later) `&&` and `||` 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.

2.7.4. Functions with one mandatory and a second but optional argument

round(x[, n]) Rounds its first argument to an integer multiple of `10^(-n)` (i.e. it *quantizes*). The case of negative `n` is new with `1.4a`. Positive `n` corresponds to conversion to a fixed point number with `n` digits after decimal mark.

```
\xinteval{round(-2^30/3^5,12), round(-2^30/3^5,-3)}
```
-4418690.633744855967, -4419e3

trunc(x[, n]) Truncates its first argument to an integer multiple of `10^(-n)`. The case of negative `n` is new with `1.4a`.

```
\xinteval{trunc(-2^30/3^5,12), trunc(-2^30/3^5,-3)}
```
-4418690.633744855967, -4418e3

float(x[, n]) Rounds its first argument to a floating point number, with a precision given by the second argument, which must be positive.

```
\xinteval{float(-2^30/3^5,12), float(-2^30/3^5, 1)}
```
-441869063374e-5, -4e6

For this example and earlier ones if the parser had been `\xintfloateval`, not `\xinteval`, the first argument (here `2^30/3^5`) would already have been computed as floating point number with numerator and denominator rounded separately first to the prevailing precision. To avoid that, use `\xintexpr...\relax` wrapper. Then the rounding or truncation will be applied to an exact fraction.

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

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

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

```
\xintfloateval{ilog10(99999999/10000000, 8), ilog10(-999999995/100000000, 8),
                                    ilog10(-999999995/100000000, 9)}\newline
\xinteval{ilog10(-999999995/100000000), ilog10(-999999995/100000000, 8)}
```
0., 1.000000000000000, 0.
0, 1

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

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

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

```
\xinttheexpr sqrt(2,31)\relax\ and \xinttheiiexpr sqrt(num(2e60))\relax
```
1414213562373095048801688724210e-30 and 1414213562373095048801688724209

There is an integer only variant for \xintiiexpr.

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

```
\xinttheexpr factorial (100,32)\relax, {\xintDigits:=32\relax \xintthefloatexpr
                                   factorial (100)\relax}\newline
\xinttheexpr factorial (50)\relax\newline
\xinttheexpr factorial (50, 32)\relax
```
93326215443944152681699238856267e126, 9.3326215443944152681699238856267e157
30414093201713378043612608166064768844377641568960512000000000000
30414093201713378043612608166065e33

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

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

Attention that the arguments are first converted to integers using \xintNum (i.e. truncated towards zero).

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

```
\pdfsetrandomseed 12345
\xinttheiiexpr randrange(10**20)\relax\newline
\xinttheiiexpr randrange(1234*10**16, 1235*10**16)\relax\newline
\printnumber{\xinttheiiexpr randrange(10**199,10**200)\relax}\par
```
12545314555479298502
12341249468233524155
38724271496566552250944896366777081662436330824968873373120332258200044549497099786643312
910668754171686190691274354022744800916546182607238310753247166933564523488356899132776523
9525848635299939966 2728

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

2.7.5. Functions with two arguments

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

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

mod(f, g) computes `f - g*floor(f/g)`. Hence its output is a general fraction or floating point number or integer depending on the used parser.

Prior to `1.2p` it computed `f - g*trunc(f/g)`.

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

```
\xinttheexpr mod(11/7,1/13), reduce(((11/7)//(1/13))*1/13+mod(11/7,1/13)),
mod(11/7,1/13)- (11/7)/:(1/13), (11/7)//(1/13)\relax\newline
\xintthefloatexpr mod(11/7,1/13)\relax\par
```
3/91, 11/7, 0, 20
0.03296703296703260

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

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

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

```
\noindent\xinttheexpr 29/5 /: 3/5, 11/3 /: 17/19\relax\newline
        \xinttheexpr (29/5) /: (3/5), (11/3) /: (17/19)\relax\newline
        \xintthefloatexpr 29/5 /: 3/5, 11/3 /: 17/19, 11/57\relax\newline
        \xintthefloatexpr (29/5) /: (3/5), (11/3) /: (17/19), 5/57\relax\newline
        5/57 = \xinttheexpr trunc(5/57, 20)\relax\dots\newline
```
14/25, 11/57
2/5, 5/57
0.5600000000000000, 0.1929824561403509, 0.1929824561403509
0.4000000000000000, 0.08771929824561420, 0.08771929824561404
5/57 = 0.08771929824561403508...
Regarding some details of behaviour in `\xintfloatexpr`, see discussion of `divmod` function next.

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

Changed at 1.4!

In `\xintfloatexpr...\relax` the modulo is rounded to the prevailing precision. The quotient is like in the other parsers an exact integer. It will be rounded as soon as it is used in further operations, or via the global output routine of `\xintfloatexpr`. *Those examples behave as in 1.3f because assignments to multiple variables tacitly unpack if this is necessary.*

```
\xintdefvar Q, R := divmod(3.7, 1.2);%
\xinttheexpr Q, R, 1.2Q + R\relax\newline
\xintdefiivar Q, R := divmod(100, 17);%
\xinttheiiexpr Q, R, 17Q + R\relax\newline
\xintdeffloatvar Q, R := divmod(100, 17e-20);%
\xintthefloatexpr Q, R, 17e-20 * Q + R\relax\newline
% show Q exactly, although defined as float it can be used in iiexpr:
```

```
  \xinttheiiexpr Q\relax\ (we see it has more than 16 digits)\par
  \xintunassignvar{Q}\xintunassignvar{R}%
```
3, 1e-1, 37e-1
5, 15, 100
5.882352941176471e20, 9.000000000000000e-20, 100.0000000000000
588235294117647058823 (we see it has more than 16 digits)

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

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

```
  \xintDigits := 4\relax
  \xintthefloatexpr if(12345678//23=537000, 1, 0), 12345678//23\relax\newline
  \xintthefloatexpr if(float(12345678//23)=537000, 1, 0)\relax\par
  \xintDigits := 16\relax
```
0., 537000.
1.000

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

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

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

```
  \xinttheexpr seq(binomial(20, i), i=0..20)\relax
```
1, 20, 190, 1140, 4845, 15504, 38760, 77520, 125970, 167960, 184756, 167960, 125970, 77520, 38760, 15504, 4845, 1140, 190, 20, 1

```
  \printnumber{\xintthefloatexpr seq(binomial(100, 50+i), i=-5..+5)\relax}%
```
6.144847121413618e28, 7.347099819081500e28, 8.441348728306404e28, 9.320655887504988e28, 9↲
.891308288780803e28, 1.008913445455642e29, 9.891308288780803e28, 9.320655887504988e28, 8.↲
441348728306404e28, 7.347099819081500e28, 6.144847121413618e28

The arguments must be (expand to) short integers.

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

```
  \xinttheexpr seq(pfactorial(20, i), i=20..30)\relax
```
1, 21, 462, 10626, 255024, 6375600, 165765600, 4475671200, 125318793600, 3634245014400, 109027350432000

The arguments must (expand to) short integers. See *subsection 8.36* for the behaviour if the arguments are negative.

New with 1.4 **ndfillraw(TEX-macro, n-uple)** The second argument is `[N1, N2, ..., Nk]`. The construct fills an `N1x`↲ `N2x...xNk` hyperrectangular nested list by evaluating the given `macro` as many times as needed. The expansion result goes directly into internal data and must thus comply with what is expected internally for an individual numeric leaf (at `1.4`, `xintfrac` raw format worked for `\xi`↲

Do not use! `ntexpr` or `\xintfloatexpr`, but not `\xintiiexpr`, and this may have changed since). This is an experimental function serving to generate either constant or random arrays. Attention that **TEX-macro** stands here for any expandable TEX *macro*, and an `\xintexpr`-ession at this location thus requires an explicit `\xinteval` wrapping.

26

2.7.6. Functions with 3 or 4 arguments

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

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

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

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

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

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

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

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

2.7.7. Functions with an arbitrary number of arguments

At 1.4 all(), any(), xor(), `+`(), `*`(), max(), min(), gcd(), lcm(), first(), last(), reversed() and len() admit:

- at least two arguments, and then operate as expected in backward compatible way,

☞ {
- or only one argument, which then *must* be a nut-ple, i.e. a variable or explicit bracketed list. In the case of reversed() the output is a nut-ple if the input was one.

Notice that this is breaking change as the functions do not work anymore with a single argument being a number (or give funny non-documented results depending on internal data representation).

Do not use! **qraw(stuff)** It injects directly tokens to represent internally numerical data. Will break at any release modifying the internal data format specifications (which are not always documented).

all(x, y, ...) inserts a logical AND in-between its arguments and evaluates the resulting logical assertion (as with all functions, all arguments are evaluated).

> `\xinteval{all(1,1,1), all([1,0,1]), all([1,1,1])}`
> 1, 0, 1

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

> `\xinteval{any(0,0,0), any([1,0,1]), any([0,0,0])}`
> 0, 1, 0

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

> `\xinteval{xor(1,1,1), xor([1,0,1]), xor([1,1,1])}`
> 1, 0, 1

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

> `\xinttheexpr `+`(1,3,19), `+`(1**2,3**2,sqr(19)), `+`([1**2,3**2,sqr(19)])\relax`
> 23, 371, 371

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

```
\xinttheexpr `*`(1,3,19), `*`(1^2,3^2,19^2), `*`([1^2,3^2,19^2])\relax
```
57, 3249, 3249

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

```
\xinttheexpr max(1,3,19), min([1,3,19])\relax
```
19, 1

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

```
\xinttheexpr min(1,3,19), min([1,3,19])\relax
```
1, 1

gcd(x, y, ...) computes the positive generator of the fractional ideal of rational numbers $x\mathbb{Z} + y\mathbb{Z} + \dots \subset \mathbb{Q}$. When the inputs are integers it is advantageous to use a sub \xintiiexpr-ession, as the integer-only macro is more efficient (about 6X) than the one accepting general fractional inputs. Notice that this may require some num() wrapper when using variables, as they may well be in fraction format, and \xintiiexpr accepts only strict integers. Since 1.3d, this function and lcm() are available whether or not package xintgcd is loaded. Note that like other operations with fractions it does not always produce a fraction in irreducible format. This example shows also how to reduce an n-uple to its primitive part: (this example should be revisited)

```
\xinttheexpr gcd(7/300, 11/150, 13/60)\relax\newline
$(7/300, 11/150, 13/60)\to
(\xinttheexpr subs(seq(reduce(x/D), x = 7/300, 11/150, 13/60), D=gcd(7/300, 11/150, 13/60))\relax)$\newline
\xintexpr gcd([7/300, 11/150, 13/60])\relax\par
```
1/300
$(7/300, 11/150, 13/60) \to (7, 22, 65)$
1/300

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

lcm(x, y, ...) computes the positive generator of the fractional ideal of rational numbers $x\mathbb{Z} \cap y\mathbb{Z} \cap \dots \subset \mathbb{Q}$. When the inputs are integers it is advantageous to use a sub \xintiiexpr-ession, as the integer-only macro is more efficient (about 9X) than the one accepting general fractional inputs.

```
\xinttheexpr lcm([7/300, 11/150, 13/60])\relax
```
1001/30

first(x, y, ...) first item of the list or nut-ple argument:

```
\xintiiexpr first([last(-7..3), [58, 97..105]])\relax
```
3

last(x, y, ...) last item of the list or nut-ple argument:

```
\xintiiexpr last([-7..3, 58, first(97..105)])\relax
```
97

reversed(x, y, ...) reverses the order of the comma separated list or inside a nut-ple:

```
\xintiieval{reversed(reversed(1..5), reversed([1..5]))}
```
[5, 4, 3, 2, 1], 1, 2, 3, 4, 5

The above is correct as xintexpr functions may produce oples and this is the case here.

len(x, y, ...) computes the number of items in a comma separated list or inside a nut-ple (at first level only: it is not a counter of leaves).

```
\xinttheiiexpr len(1..50, [101..150], 1001..1050), len([1..10])\relax
```
101, 10

New with
1.4b

zip(*nutples) behaves similarly to the Python function of the same name: i.e. it produces *an ople of nut-ples, where the i-th nut-ple contains the i-th element from each of the argument nut-ples. The ople ends when the shortest input nut-ple is exhausted. With a single nut-ple argument, it returns an ople of 1-nutples. With no arguments, it returns the empty ople.*

As there is no exact match in xintexpr of the concept of «iterator» object,[15] there is a significant difference here that (for example) the zip(x,x,x) Python idiom to cluster the iterator x into successive chunks of length 3 does not apply. Consider for this reason even the name of

unstable?

the function as work-in-progress, susceptible to change.

```
\xintiieval{zip([1..9], [0, 1, 2], [11..29], [111..139])}
```
[1, 0, 11, 111], [2, 1, 12, 112], [3, 2, 13, 113]

See also \xintthespaceseparated for some possible usage in combination with flat().

2.7.8. Functions requiring dummy variables

The pseudo-functions subs(), seq(), subsm(), subsn(), iter(), add(), mul(), rseq(), iterr(), rrseq(), iterr(), ndseq(), ndmap(), ndfillraw() use delimited macros for some tasks:

- for all of them, whenever a <varname>= chunk must be parsed into a (non-assigned) variable name, then the equal sign must be visible,

- and if the syntax is with ,<varname>= the initial comma also must be visible (spaces do not matter),

- for all of them but ndmap() and ndfillraw() the final closing parenthesis must be visible.

Although delimited macros involving commas are used to locate ,<varname= this is done in a way silently ignoring commas located inside correctly balanced parentheses. Thus, as the examples will show, nesting works as expected.

The semi-colons involved in the syntax may arise from expansion alone. For rseq(), iter(), rrseq() and iterr() the ,<varname>= part may also be created from the expansion which will generate the initial comma separated values delimited by a semi-colon.

Prior to 1.4, semi-colons needed to be braced or otherwise hidden when located in an expression parsed by \xintdefvar or \xintdeffunc, to not be confused with the expression terminator.

New with
1.4

This is not needed anymore.

seq(), rseq(), iter(), rrseq(), iterr() and also add(), mul(), but not subs() admit the omit, abort, and break() keywords. This is a new feature at 1.4 for add() and mul().

In the case of a potentially infinite list generated by the <integer>++ syntax, use of abort or of break() is mandatory, naturally.

All lowercase and uppercase Latin letters are pre-configured for usage as dummy variables. In Unicode engines one can use \xintnewdummy to turn any letter into a usable dummy variable.

New with
1.4

And since 1.4, \xintnewdummy works (in all engines) to turn a multi-letter word into a dummy variable. In the descriptions, varname stands for such a dummy variable, either single-letter or word.

subs(expr, varname=values) for variable substitution.

```
\xinttheexpr subs(subs(seq(x*z,x=1..10),z=y^2),y=10)\relax\newline
```
100, 200, 300, 400, 500, 600, 700, 800, 900, 1000

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

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

---

[15] Speaking of iterators, I have some ideas about this: as \xintexpr does not have the global expression in its hands it is difficult to organize globally expandably the idea of iterator, but locally via syntax like the one for seq() this is feasible. When one thinks about it, seq() is closely related to the iterator idea.

of efficiency. But subs() is helpless in function definitions: all places where a variable is substituted will receive the complete recipe to compute the variable, rather than evaluate only once.

One should rather define auxiliary functions to compute intermediate results. Or one can use seq(). See the documentation of \xintdeffunc.

add(expr, varname=values) addition

```
\xintiiexpr add(x^3,x=1..20), add(x(x+1), x=1,3,19)\relax\newline
\xintiiexpr add(x^3, x = 1..[2]..20)\relax\newline        % add only odd cubes
\xintiiexpr add((odd(x))?{x^3}{omit}, x = 1..20)\relax\par % add only odd cubes
```
44100, 394
19900
19900

At 1.4 (fixed at 1.4a), the keywords omit (as in example above), abort and break() are allowed. The meaning of break() is specific: its argument serves as last operand for the addition, not as ultimate value.

```
\xintiiexpr add((x>10)?{break(1000)}{x}, x = 1..15)\relax
```
1055

The @ special variable holds the so-far accumulated value. Initially its value is zero.

```
\xintiiexpr add(1 + @, i=1..10)\relax % iterates x <- 2x+1
```
1023

See `+`() for syntax simply adding items of a list without usage of a dummy variable.

mul(expr, varname=values) multiplication

```
\xintiiexpr mul(x^2, x = 1, 3, 19, 37..50)\relax
```
2171846665384874110852122798021721110872064000000000

The @ special variable holds the so-far accumulated value. Initially its value is one.

At 1.4 (fixed at 1.4a), the keywords omit, abort and break() are allowed. The meaning of break() is specific: its argument serves as last operand for the multiplication, not as ultimate value.

```
\xintiieval{mul((i==100)?{break(i^4)}{i}, i = 98, 99, 100)}
```
970200000000

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

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

```
\xintiiexpr seq(x(x+1)(x+2)(x+3),x=1..10), `*`(seq(3x+2,x=1..10))\relax
```
24, 120, 360, 840, 1680, 3024, 5040, 7920, 11880, 17160, 1162274713600

```
\smallskip
\leavevmode\vbox{\xintthealign\xintiiexpr [seq([seq(i^2+j^2, i=0..j)], j=0..10)]\relax}
```
```
[[  0      ],
 [  1,    2      ],
 [  4,    5,    8      ],
 [  9,    10,   13,   18     ],
 [  16,   17,   20,   25,   32     ],
 [  25,   26,   29,   34,   41,   50     ],
 [  36,   37,   40,   45,   52,   61,   72     ],
 [  49,   50,   53,   58,   65,   74,   85,   98     ],
 [  64,   65,   68,   73,   80,   89,   100,  113,  128    ],
 [  81,   82,   85,   90,   97,   106,  117,  130,  145,  162    ],
 [  100,  101,  104,  109,  116,  125,  136,  149,  164,  181,  200  ]]
```

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

```
    \printnumber {\xintthefloatexpr subs(rseq (1; @/2+y/2@, i=1..10),y=1000)\relax }\newline
```
1.000000000000000, 500.5000000000000, 251.2490009990010, 127.6145581634591, 67.7253273608↲
2604, 41.24542607499115, 32.74526934448864, 31.64201586865079, 31.62278245070105, 31.62277↲
660168434, 31.62277660168379

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

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

```
    \printnumber{\xintthefloatexpr rseq(1,10^6;
                 (sqrt(@[0]*@[1]),(@[0]+@[1])/2), i=1..7)\relax }
```
1.000000000000000, 1.000000000000000e6, 1000.000000000000, 500000.5000000000, 22360.69095↲
533499, 250500.2500000000, 74842.22521066670, 136430.4704776675, 101048.3052657827, 10563↲
6.3478441671, 103316.8617608946, 103342.3265549749, 103329.5933734841, 103329.5941579348,
103329.5937657094, 103329.5937657095

Prior to 1.4 the above example had to be written with [@]. This is still possible (@ stands
for an ople with two items, bracketing then extracting is like extracting directly), but it
is leaner to drop the extra «packing».

iter(initial value; expr, varname=values) is exactly like rseq, except that it only prints the last
iteration.

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

```
  \xintDigits:= 87\relax
  % Below 83 is 87-3-1 (3 guard digits, target 84=1+83 digits) and 43 is 84/2+1.
  \xintdeffloatfunc BS(a, b, t, p):= 0.5*(a+b), sqrt(a*b), t-p*sqr(a-b), \xintiiexpr 2p\relax;
  \xinteval
  {trunc(% I feel truncation is better than rounding to display decimals of $\pi$
    \xintfloatexpr
      iter(1, sqrt(0.5), 1, 1; % initial values
        (@[0]-@[1]<2e-43)?% stopping criteria; takes into account that the
                          % exit computation (break() argument) doubles
                          % number of exact digits (roughly)
        {break(sqr(@[0]+@[1])/@[2])}     % ... do final computation,
        {BS(@)}, % else do iteration
         i=1++)  % This generates infinite iteration. The i is not used.
    \relax
  , 83)% closing parenthesis of trunc()
  }...% some dots following end of \xinteval argument
  \xintDigits:=16\relax
```
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862...
You can try with \xintDigits:=1004\relax and 2e-501 in place of \xintDigits:=87\relax and 2e↲
-43, but be patient for some seconds for the result. Of course don't truncate the final result
to only 83 fractional decimal digits but 1000... and better to wrap the whole thing in \message
or \immediate\write128 or \edef because it will then run in the right margin.

Prior to 1.4 the above example had to use notation such as [@][0]; this would still work but
@[0] is leaner.

rrseq(initial values; expr, varname=values) recursive sequence with multiple initial terms. Say,
there are K of them. Then @1, ..., @4 and then @@(n) up to n=K refer to the last K values.
Notice the difference with rseq() for which @ refers to a list of items in case the initial
value is a list and not a single item.[16] Using rrseq() with @1 etc... accessors may be perhaps

---

[16] Prior to 1.4, one could use @ in rrseq() and iterr() as an alias to @1. This undocumented feature is dropped and @ will break
rrseq() and iterr().

a bit more efficient than using `rseq()` with a list as staring value and constructs such as `@[₂0]`, `@[1]` (or rather `@[-1]`, `@[-2]` to mimick what `@1`, `@2`, `@3`, `@4` and `@@(integer)` do in `rrseq()`.

```
\xinttheiiexpr rrseq(0,1; @1+@2, i=2..30)\relax
```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040

```
\xinttheiiexpr rseq(1; 2@, i=1..10)\relax
```
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024

```
\xinttheiiexpr rseq(1; 2@+1, i=1..10)\relax
```
1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047

```
\xinttheiiexpr rseq(2; @(@+1)/2, i=1..5)\relax
```
2, 3, 6, 21, 231, 26796

```
\xinttheiiexpr rrseq(0,1,2,3,4,5; @1+@2+@3+@4+@@(5)+@@(6), i=1..20)\relax
```
0, 1, 2, 3, 4, 5, 15, 30, 59, 116, 229, 454, 903, 1791, 3552, 7045, 13974, 27719, 54984, 109065, 216339, 429126, 851207, 1688440, 3349161, 6643338

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

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

```
\xinttheiiexpr iterr(0,1; @1+@2, i=2..5, 6..10)\relax
% the iterated over list is allowed to have disjoint defining parts.
```
55

New with 1.4 **subsm(expr, var1=value1; var2=value2; ....; varN=valueN[;])** Simultaneous substitutions. The assigned values must not involve the variables. An optional final semi-colon is allowed.

```
\xintiieval{subsm(x+2y+3z+4t, x=1; y=10; z=100; t=1000;)}
```
4321

New with 1.4 **subsn(expr, var1=value1; var2=value2; ....; varN=valueN[;])** Simultaneous substitutions. The assigned values may involve all variables located further to its right. An optional final semi-colon is allowed.

```
\xintiieval{subsn(x+y+z+t, x=20y; y=20z; z=20t; t=1)}
```
8421

New with 1.4 **ndmap(function, values1; values2; ....; valuesN[;])** Construction of a nested list (a priori having N dimensions) from function values. The function must be an N-variable function (or a function accepting arbitrarily many arguments), but it is not constrained to produce only scalar values. Only in the latter case is the output really an N-dimensional «ndlist» type object. An optional final semi-colon in the input before the closing parenthesis is allowed.

```
\xintdeffunc foo(a,b,c,d) = a+b+c+d;
\begin{multicols}{2}
\xintthealign\xintexpr ndmap(foo, 1000,2000,3000; 100,200,300; 10,20,30; 1,2,3)\relax
\end{multicols}
```

```
[[[[ 1111,  1112,  1113  ],                         [ 1331,  1332,  1333  ]]],
   [ 1121,  1122,  1123  ],              [[[ 2111,  2112,  2113  ],
   [ 1131,  1132,  1133  ]],                [ 2121,  2122,  2123  ],
  [[ 1211,  1212,  1213  ],                 [ 2131,  2132,  2133  ]],
   [ 1221,  1222,  1223  ],               [[ 2211,  2212,  2213  ],
   [ 1231,  1232,  1233  ]],                [ 2221,  2222,  2223  ],
  [[ 1311,  1312,  1313  ],                 [ 2231,  2232,  2233  ]],
   [ 1321,  1322,  1323  ],               [[ 2311,  2312,  2313  ],
```

```
            [  2321,  2322,  2323  ],                    [  3221,  3222,  3223  ],
            [  2331,  2332,  2333  ]]],                   [  3231,  3232,  3233  ]],
        [[[  3111,  3112,  3113  ],                    [[  3311,  3312,  3313  ],
            [  3121,  3122,  3123  ],                    [  3321,  3322,  3323  ],
            [  3131,  3132,  3133  ]],                    [  3331,  3332,  3333  ]]]]
         [[  3211,  3212,  3213  ],
```

New with
1.4
ndseq(expr, var1=values1; var2=values2; ....; varN = valuesN[;]) Constructs a nested list (a pri-
ori having N dimensions) from substitutions in an expression involving N (dummy) variables.
The expression is not constrained to produce only scalar values. Only in the latter case is
the output really an N-dimensional «ndlist» type object. An optional final semi-colon in the
input before the closing parenthesis is allowed.

```
\begin{multicols}{2}
\xintthealign\xintexpr ndseq(a+b+c+d, a=1000,2000,3000; b=100,200,300; c=10,20,30; d=1,2,3;)\relax
\end{multicols}% in case of page break, this makes amusing zigzag rendering
```

```
[[[[  1111,  1112,  1113  ],                    [  2231,  2232,  2233  ]],
    [  1121,  1122,  1123  ],                  [[  2311,  2312,  2313  ],
    [  1131,  1132,  1133  ]],                   [  2321,  2322,  2323  ],
  [[  1211,  1212,  1213  ],                      [  2331,  2332,  2333  ]]],
    [  1221,  1222,  1223  ],                  [[[  3111,  3112,  3113  ],
    [  1231,  1232,  1233  ]],                    [  3121,  3122,  3123  ],
  [[  1311,  1312,  1313  ],                      [  3131,  3132,  3133  ]],
    [  1321,  1322,  1323  ],                   [[  3211,  3212,  3213  ],
    [  1331,  1332,  1333  ]]],                    [  3221,  3222,  3223  ],
 [[[  2111,  2112,  2113  ],                       [  3231,  3232,  3233  ]],
    [  2121,  2122,  2123  ],                   [[  3311,  3312,  3313  ],
    [  2131,  2132,  2133  ]],                    [  3321,  3322,  3323  ],
  [[  2211,  2212,  2213  ],                       [  3331,  3332,  3333  ]]]]
    [  2221,  2222,  2223  ],
```

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

abort it is a pseudo-variable which indicates to stop here and now.

omit it is a pseudo-variable which says to omit this value and go to next one.

break(stuff) says to abort and insert stuff as last value.

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

```
\xinttheiiexpr iter(1;(@>10^40)?{break(@)}{2@},i=1++)\relax
```
1088903574147003083082798743781658276659 is the smallest power of 2 with at least fourty one
digits.

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

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

33

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

Note: the above example, up to 1.3f used break(i) in place of current break(@1, i). This syntax looks in retrospect as having been a bug. Starting with 1.4 the example does have to be written with break(@1, i), as break(i) conforming to intuition will only print the last i value. And if one also wants the previous Fibonacci number one only has to use break(@2, @1, i) for example.

2.8. Generators of arithmetic progressions

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

  The .. has lower precedence than the arithmetic operations.

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

  The step of replacing a by its ceil and b by its floor is a kind of silly overhead, but a and b are allowed to be themselves the result of computations and there is no notion of «int» type in \xinteval. The solution is, when a and b are given explicit integers to temporarily switch to the \xintiiexpr parser:

  `\xintexpr \xintiiexpr 1..10\relax\relax`
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10

  On the other hand integers from \xintexpr 1..10\relax are already in raw xintfrac format for example 3/1[0] which speeds up their usage in the macros internally involved in computations... thus perhaps what one gains on one side is lost on the other side.

- a..[d]..b generates «real» numbers along arithmetic progression of reason d. It does *not* replace a by its ceil, nor b by its floor. The generated list is empty if b-a and d are of opposite signs; if d=0 or if a=b the list expands to single element a.

  `\xintexpr 1.5..[1.01]..11.23\relax`
  15e-1, 251e-2, 352e-2, 453e-2, 554e-2, 655e-2, 756e-2, 857e-2, 958e-2, 1059e-2

At 1.4, this generator behaves in \xintfloatexpr exactly as in \xintexpr, i.e. *exactly*. This is breaking change.

  `\xintDigits:=6;`
  `\xintexpr\xintfloatexpr 100..[1.23456]..110\relax\relax`
  `\xintDigits:=16;`
  100, 10123456e-5, 10246912e-5, 10370368e-5, 10493824e-5, 10617280e-5, 10740736e-5, 10864192e-5, 10987648e-5

  This demonstration embedded the float expression in the exact parser only to avoid the rounding to the prevailing precision on output, thus we can see that internally additions are done exactly and not with 6 digits mantissas (in this example).

2.9. Python slicing and indexing of one-dimensional sequences

There are some breaking changes in the syntax at 1.4, because previously xintexpr had no real notion of a list or sequence type. It now does, and even allows nesting.

We denote here by *list* or *sequence* a general *ople*, either given as a variable or explicitly. In the former case the parentheses are optional.

- (list)[n] returns the n+1th item if n>=0. If n<0 it enumerates items from the tail. Items are numbered as in Python, the first element corresponding to n=0.

  `\xintexpr (0..10)[6], (0..10)[-1], (0..10)[23*18-22*19]\relax`

34

```
6, 10, 7
```

This also works for singleton *oples* which are in fact a *number*:

```
\xintexpr (7)[0], (7)[-1], 9, (7)[-2], 9\relax
```

```
7, 7, 9, 9
```

In the example above the parentheses serve to disambiguate from the raw xintfrac format such as 7[-1] which, although discouraged, is accepted on input. And we used a trick to show that (7)[-2] returns nil.

The behaviour changes for singleton *oples* which are not *numbers*. They are thus *nut-ples*, or equivalently they are the bracketing (bracing, packing) of another *ople*. In this case, the meaning of the syntax for item indexing is, as in Python, item *extraction*:

```
\xintexpr [0,1,2,3,4,5][2], [0,1,2,3,4,5][-3]\relax\newline
\xintexpr [0,[1,2,3,4,5],6][1][-1]\relax
```

```
2, 3
5
```

- (list)[:n] produces the first n elements if n>0, or suppresses the last |n| elements if n<0.

  ```
  \xintiiexpr (0..10)[:6]\relax\ and \xintiiexpr (0..10)[:-6]\relax
  ```

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

  As above, the meaning change for *nut-ples* and fits with expectations from Python regarding its sequence types:

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

  ```
  [0, 1, 2, 3, 4, 5] and [0, 1, 2, 3, 4]
  ```

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

  ```
  \xintiiexpr (0..10)[6:]\relax\ and \xintiiexpr (0..10)[-6:]\relax
  ```

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

  As above, the meaning change for *nut-ples* and fit with expectations from Python with *tuple* or *list* types:

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

  ```
  [6, 7, 8, 9, 10] and [5, 6, 7, 8, 9, 10]
  ```

- Finally, (list)[a:b] also works according to the Python ``slicing'' rules (inclusive of negative indices). Notice though that stepping is currently not supported.

  ```
  \xinttheiiexpr (1..20)[6:13]\relax\ = \xinttheiiexpr (1..20)[6-20:13-20]\relax\newline
  \xinttheiiexpr [1..20][6:13]\relax\ = \xinttheiiexpr [1..20][6-20:13-20]\relax
  ```

  ```
  7, 8, 9, 10, 11, 12, 13 = 7, 8, 9, 10, 11, 12, 13
  [7, 8, 9, 10, 11, 12, 13] = [7, 8, 9, 10, 11, 12, 13]
  ```

- It is naturally possible to execute such slicing operations one after the other (the syntax is simplified compared to before 1.4):

  ```
  \xintexpr (1..50)[13:37][10:-10]\relax\newline
  \xintexpr (1..50)[13:37][10:-10][-1]\relax
  ```

  ```
  24, 25, 26, 27
  27
  ```

2.10. NumPy like nested slicing and indexing for arbitrary oples and nut-ples

New with
1.4

This is entirely new with 1.4.

I will give one illustrative example and refer to the NumPy documentation for more.

Notice though that our interpretation of the syntax is more general than NumPy's concepts (of basic slicing/indexing):

- slicing and itemizing apply also to non-bracketed objects i.e. *oples*,

- the leaves do not have to be all at the same depth,

- there are never any out-of-range index errors: out-of-range indices are silently ignored.

```
\begin{multicols}{3}
\xintdefvar myArray = ndseq(a+b+c, a=100,200,300; b=40,50,60; c=7,8,9);
myArray = \xintthealign\xintexpr myArray\relax
\columnbreak
mySubArray = \xintthealign\xintexpr myArray[0:2,0:2,0:2]\relax
myExtractedSubArray = \xintthealign\xintexpr myArray[0:2,0:2,0:2][0]\relax
\columnbreak
myExtractedSubArray = \xintthealign\xintexpr myArray[0:2,0:2,0:2][0,1]\relax
\noindent
firstExtractedScalar = \xintexpr myArray[0:2,0:2,0:2][0,1,0]\relax\newline
secondExtractedScalar = \xintexpr myArray[0,1,0]\relax\par
\end{multicols}
```

```
 myArray =                      mySubArray =                    myExtractedSubArray =
[[[ 147,  148,  149 ],         [[[ 147,  148 ],                [ 157,  158 ]
  [ 157,  158,  159 ],           [ 157,  158 ]],               firstExtractedScalar = 157
  [ 167,  168,  169 ]],         [[ 247,  248 ],                secondExtractedScalar = 157
 [[ 247,  248,  249 ],           [ 257,  258 ]]]
  [ 257,  258,  259 ],
  [ 267,  268,  269 ]],
 [[ 347,  348,  349 ],          myExtractedSubArray =
  [ 357,  358,  359 ],         [[ 147,  148 ],
  [ 367,  368,  369 ]]]          [ 157,  158 ]]
```

As said before, *stepping* is not yet implemented. Also the NumPy extension to Python for item selection (i.e. via a tuple of comma separated indices) is not yet implemented.

2.11. Tacit multiplication

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

(1.) *encounters a count or dimen or skip register or variable or an ε-T$_E$X expression,* or

(2.) *encounters a sub-\xintexpression,* or

(3.) *encounters an opening parenthesis,* or

(4.) *encounters a letter (which is interpreted as signaling the start of either a variable or a function name),* or

(5.) (of course, only when in state "looking for an operator") *encounters a digit.*

☞ {
   For example, if x, y, z are variables all three of (x+y)z, x(y+z), (x+y)(x+z) will create a tacit multiplication.
   Furthermore starting with release 1.2e, whenever tacit multiplication is applied, in all cases it *always* ``ties'' more than normal multiplication or division, but still less than power. Thus x/2y is interpreted as x/(2y) and similarly for x/2max(3,5) but x^2y is still interpreted as (x^2)*y and 2n! as 2*n!.
```
   \xintdefvar x:=30;\xintdefvar y:=5;%
   \xinttheexpr (x+y)x, x/2y, x^2y, x!, 2x!, x/2max(x,y)\relax
```

```
1050, 30/10, 4500, 265252859812191058636308480000000, 5305057196243821172726169600000000,
30/60
   Since 1.2q tacit multiplication is triggered also in cases such as (1+2)5 or 10!20!30!.
   \xinttheexpr (10+7)5, 4!4!, add(i, i=1..10)10, max(x, y)100\relax
85, 576, 550, 3000
   The ``tie more'' rule applies to all cases of tacit multiplication. It impacts only situa-
tions when a division was the last seen operator, as the normal rule for the xintexpr parsers
is left-associativity in case of equal precedence.
   \xinttheexpr 1/(3)5, (1+2)/(3+4)(5+6), 2/x(10), 2/10x, 3/y\xintiiexpr 5+6\relax, 1/x(y)\relax\
   differ from\newline\xinttheexpr 1/3*5, (1+2)/(3+4)*(5+6), 2/x*(10), 2/10*x,
                             3/y*\xintiiexpr 5+6\relax, 1/x*(y)\relax\par
1/15, 3/77, 2/300, 2/300, 3/55, 1/150 differ from
5/3, 33/7, 20/30, 60/10, 33/5, 5/30
```

Note that y\xinttheiiexpr 5+6\relax would have tried to use a variable with name y11 rather
than doing y*11: tacit multiplication works only in front of sub-\xintexpressions, not in front
of \xinttheexpressions which are unlocked into explicit digits.
   Here is an expression whose meaning is completely modified by the ``tie more'' property of tacit
multiplication:
   `\xintdeffunc e(z):=1+z(1+z/2(1+z/3(1+z/4)));`
will be parsed as
   `\xintdeffunc e(z):=1+z*(1+z/(2*(1+z/(3*(1+z/4)))));`
which is not at all the presumably hoped for:
   `\xintdeffunc e(z):=1+z*(1+(z/2)*(1+(z/3)*(1+(z/4))));`

2.12. User defined variables

Since release 1.1 it is possible to make an assignment to a variable name and let it be known to the
parsers of xintexpr. Since 1.2p simultaneous assignments are possible.
   `\xintdefvar myPi:=3.141592653589793238462643;%`
   `\xintdefvar x_1, x_2, x_3 := 10, 20, 30;%`
   `\xintdefiivar List := seq(x(x+1)/2, x=0..10);% seq produces an «open» list`
   `\xintdefiivar Nuple := ndmap(sqr, List);% ndmap produces a «bracketed» list`
   `\xintdefiivar FourthPowers := ndmap(sqr, *Nuple);% "unpacking" is needed here.`
   `$x_1 = \xinteval{x_1}, x_2 = \xinteval{x_2}, x_3 = \xinteval{x_3}$\newline`
   `$\pi^{100}\approx\xintfloateval{myPi^100}$ is evaluated \fbox{after} having rounded myPi`
   `to the prevailing float precision (which here is the default \xinttheDigits)\newline`
   `$\xintDigits:=20\relax \pi^{100}\approx\xintfloateval{myPi^100}$ (this one first`
   `rounded the variable to 20 digits before evaluating its 100th power)\newline`
   `Open List: \xintiieval{List}\newline`
   `Nuple: \xintiieval{Nuple}\newline`
   `FourthPowers: \xintiieval{FourthPowers}\par`
$x_1 = 10, x_2 = 20, x_3 = 30$
$\pi^{100} \approx 5.187848314319574e49$ is evaluated after having rounded myPi to the prevailing float pre-
cision (which here is the default 16)
$\pi^{100} \approx 5.1878483143196131983e49$ (this one first rounded the variable to 20 digits before evaluat-
ing its 100th power)
Open List: 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55
Nuple: [0, 1, 9, 36, 100, 225, 441, 784, 1296, 2025, 3025]
FourthPowers: [0, 1, 81, 1296, 10000, 50625, 194481, 614656, 1679616, 4100625, 9150625]
   By the way xinttrig defines indeed a variable Pi, but its value can be modified at user level,
with no impact whatsoever on the trigonometrical functions.
   Here is another example with simultaneous assignments:
   `\xintdefiivar A, B := 1500, 135;%`

37

```
\xintloop
\xintifboolexpr{B}
    {\xintdefiivar A, B := B, A 'mod' B;\iftrue}
    {\iffalse}
\repeat
The last non zero remainder is \xintiiexpr A\relax.
```
The last non zero remainder is 15.

    Note1: simultaneous assignments are more costly in terms of memory impact.

    Note2: in case of simultaneous assignments, the right hand side will be automatically unpacked if necessary.

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

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

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

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

```
\xintdefvar x\xintListWithSep{, x}{\xintSeq{0}{10}} := seq(2**i, i = 0..10);%
```
This defines x0, x1, ..., x10 for future usage.

    Legal variable names are composed of letters, digits, _ and @ and characters. A variable name must start with a letter. Variable names starting with a @ or _ are reserved for internal usage.

    As x_1x_2 or even x_1x are licit variable names, and as the parser does not trace back its steps, input syntax must be x_1*x_2 if the aim is to multiply such variables.

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

    Since 1.4 even assigned variables can be used in the signature of function declarations.

    Using \xintdefvar, \xintdefiivar, or \xintdeffloatvar means that the variable value will be computed using respectively \xintexpr, \xintiiexpr or \xintfloatexpr. It can then be used in all three parsers, as long as the parser understands the format. Currently this means that variables using \xintdefvar or \xintdeffloatvar can be used freely either with \xintexpr or \xintfloatexpr but not with \xintiiexpr, and variables defined via \xintdefiivar can be used in all parsers.

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

```
\xintdeffloatvar e:=2.7182818284590452353602874713526624977572470936999595749669676;%
\xintexpr        e\relax\newline         % shows the recorded value
\xintfloatexpr   e\relax\newline         % typesetter rounds on output
\xintexpr \xintfloatexpr e\relax\relax\newline      % here we see all digits again
\xintexpr \xintfloatexpr [16]e\relax\relax\newline % this forces rounding
\xintexpr \xintfloatexpr 0+e\relax\relax\newline   % rounding here done by addition
\xintexpr float(e)\relax\newline % one more way to force rounding
\xintifboolfloatexpr{e == e+0}{\error}{Different! Comparisons do not pre-round.}\par
```
2718281828459045235360287471352662497757247093699959574966 9676e-61

2.718281828459045

2718281828459045235360287471352662497757247093699959574966 9676e-61

2718281828459045e-15

2718281828459045e-15

2718281828459045e-15

Different! Comparisons do not pre-round.

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

```
Package xintexpr Info: (on line 1)
    Variable "myPi" defined with value {3141592653589793238462643[-24]}.
Package xintexpr Info: (on line 2)
    Variable "x_1" defined with value {10}.
Package xintexpr Info: (on line 2)
    Variable "x_2" defined with value {20}.
Package xintexpr Info: (on line 2)
    Variable "x_3" defined with value {30}.
Package xintexpr Info: (on line 3)
    Variable "List" defined with value {0}{1}{3}{6}{10}{15}{21}{28}{36}{45}{55}
.
Package xintexpr Info: (on line 4)
    Variable "Nuple" defined with value {{0}{1}{9}{36}{100}{225}{441}{784}{1296
}{2025}{3025}}.
Package xintexpr Info: (on line 5)
    Variable "FourthPowers" defined with value {{0}{1}{81}{1296}{10000}{50625}{
194481}{614656}{1679616}{4100625}{9150625}}.
```

Prior to 1.4 individual (scalar) values would not have been printed to the log with the braces.

2.12.1. \xintunassignvar

Variable declarations obey the current scope. To let a (multi-letter) name be unknown to (all parsers of) xintexpr without waiting the end of the scope one issues \xintunassignvar{⟨*variable*⟩}.

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

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

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

2.12.2. \xintnewdummy

Any catcode 11 character can serve as a dummy variable, via this declaration:
```
\xintnewdummy{<character>}
```
For example with XeTeX or LuaLaTeX the following works:
```
% use a Unicode engine
\input xintexpr.sty
\xintnewdummy ξ% or any other letter character !
\xinttheexpr add(ξ, ξ=1..10)\relax
\bye
```
Under \xintglobaldefstrue regime the effect of \xintnewdummy is global.

New with 1.4   Starting with 1.4, it is allowed to use \xintnewdummy with multi-letter names (obeying the condition for being a variable name).

2.12.3. \xintensuredummy, \xintrestorevariable

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

39

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

Changed at 1.4!   \xintrestorevariable was formerly called \xintrestorelettervar.

2.13. User defined functions

2.13.1. \xintNewFunction

This is syntactic sugar which allows to use notation of functions for what is nothing more in disguise than a TeX macro. Here is an example:
```
\xintNewFunction {foo}[3]{add(mul(x+i, i=#1..#2),x=1..#3)}
```
We now have a genuine function foo() of three variables which can be used in *all three parsers*.
```
\xintexpr seq(foo(0, 3, j), j= 1..10)\relax
```
24, 144, 504, 1344, 3024, 6048, 11088, 19008, 30888, 48048

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

2.13.2. \xintdeffunc

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

> - The ending semi-colon is allowed to be of active catcode, as \xintdeffunc temporarily resets catcodes before parsing the expression. But this will fail if the whole thing is inside a macro definition. Then the used semi-colon must be the standard one.
>
> - Semi-colons used inside the expression need not be hidden inside braces. (new with 1.4)
>
> - The colon before the equal sign is optional and its (reasonable) catcode does not matter.

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

- The function names are composed of letters, digits, underscores or @ signs. A function name must start with a letter. It may be a single letter (see subsubsection 2.13.7).

New with 1.4
- The variable names used in the function signature may be multi-letter words. It is also allowed for them to already be in use for previously declared variables. Their meanings will get restored for usage after the function declaration.

- A function can have at most nine arguments. It can be defined as a function with no arguments.

- Recursive definitions are possible; for them to not generate error or fall in infinite loops, **Changed at 1.4!** the use of the short-circuit conditionals ? and ?? is *mandatory*.

Changed at 1.4!
- If a function is used in another definition it will check if it is applied to numerical arguments and if this is the case will expand fully. Prior to 1.4 one needed deprecated \xintdeffunc for this. But the latter is now but an alias for \xintdeffunc, the two have been merged.

- The previous item has an exception for functions with no arguments; they never expand immediately in other function definitions (else they would be almost like variables). This provides a way to define functions with parameters: simply let their definition use some functions with no arguments.

- A function declared via \xintdeffunc remains unknown to \xintfloatexpr (or \xintfloateval). See \xintdeffloatfunc, \xintdefiiifunc. One can use the same formula in a new definition, but if one wants the expansion to execute in a parser independent way, one can transfer a function with scalar values like this:

      \xintdeffloatfunc foo(x) := float_(\xintexpr foo(x)\relax);

New with 1.4
The float_() wrapper is in order for the float variant to produce an already-rounded value, possibly speeding-up usage if used as input for other functions. Notice the final underscore in the name; it is not mandatory but float() here would mean adding a check for optional argument hence silly overhead, as this check can only be done at time of use (as \xintexpr...\relax may a priori produce an ople).

- And in the reverse direction one can do:

      \xintdeffunc bar(x) := \xintfloatexpr bar(float(x))\relax;

  With this the transplanted float-function will expand in \xintexpr as it would have in \xintfloatexpr, i.e. using float operations; this is different from declaring the function again with the same expression as used for the original, as it would have then been parsed with a mapping of infix operators to the macros doing the exact operations, not the floating point ones.

  The float() above is not mandatory but recommended. The macro associated to the user float function bar(x) may use many times its argument x and does not worry about rounding it, because its expectation is that it is already rounded; but in \xintexpr that value could very well be a fraction 19/13 and its float rounding will be done again by each float macro receiving it as argument; with a float() used as above this will have already been done once and the ulterior roundings are faster: they have nothing to do apart from realizing that they have nothing to do.... One can also use sfloat(), this would serve to nothing for the 19/13 case but would possibly for a short integer input involved in multiplications.

  Here it is not needed to use float_(), because it will be identified at time of definition that float() is used without optional argument.

A function once declared is a first class citizen, its expression is entirely parsed and converted into a big nested *f-expan*dable macro.
When used its action is via this defined macro. For example
\xintdeffunc
      e(z):=((((((((((z/10+1)z/9+1)z/8+1)z/7+1)z/6+1)z/5+1)z/4+1)z/3+1)z/2+1)z+1;

41

creates a macro whose meaning one can find in the log file, after `\xintverbosetrue`. Here it is (it has at 1.4 an extra external brace pair compared to what happened with earlier releases):

```
    Function e for \xintexpr parser associated to \XINT_expr_userfunc_e with me
aning macro:#1->{\xintAdd {\xintMul {\xintAdd {\xintDiv {\xintMul {\xintAdd {\x
intDiv {\xintMul {\xintAdd {\xintDiv {\xintMul {\xintAdd {\xintDiv {\xintMul {\
xintAdd {\xintDiv {\xintMul {\xintAdd {\xintDiv {\xintMul {\xintAdd {\xintDiv {
\xintMul {\xintAdd {\xintDiv {\xintMul {\xintAdd {\xintDiv {#1}{10}}{1}}{#1}}{9
}}{1}}{#1}}{8}}{1}}{#1}}{7}}{1}}{#1}}{6}}{1}}{#1}}{5}}{1}}{#1}}{4}}{1}}{#1}}{3}
}{1}}{#1}}{2}}{1}}{#1}}{1}}
```

Changed at 1.4! The above is not entirely true. At 1.4, `\xintdeffunc` is more powerful and digests more of the syntax but it may have to store it in such a way that usage will be done via a sub-expression: hence it is not the case that the original expression has been *entirely* parsed. See `\xintNewFunction` for related discussion.

The main difficulty of `\xintdeffunc` is with the pseudo-functions seq(), iter(), etc..., which admit the keywords omit, abort, break(). We have no alternative for them, if the iterated over values are not entirely numerical than to postpone expansion, but this means simply storing for later a possibly big sub-expression.

At 1.4 we did some obstinate work to make this working but:

- this means that the stored function body has not been entirely parsed, parsing will happen on the fly at each execution for small or large bits,

- there remains a main stumbling-block. If the variables used in the function declaration are used only in the iterated over values or the initial values, then the mechanism may work. If however they are used not only in those values iterated over but directly in the expression which the generators map to the iterated over values, then it will break certainly. Indeed at this stage the variables are simply names, and it is impossible to transfer the mechanism which converts these names into numerical arguments for delayed usage by the declared function. Except if one is ready to basically freeze the entire thing; which then is not any different at all than using `\xintNewFunction`.

Conclusion: if some `\xintdeffunc` break, check if it does not fit the above criterion before reporting... and recall `\xintNewFunction` is your friend. It has the big advantage of declaring a function for all parsers simultaneously!

A special note on subs(): it is and has always been hopeless in `\xintdeffunc` context. All it does (if it works at all) after being malaxed by `\xintdeffunc` is to copy over at the indicated places the *recipe* to compute something. Thus everywhere where that something is needed it will be evaluated from scratch again. Yes, this is disappointing. But... on the other hand the more general seq() does work, or pretends to work. Let me illustrate to make thinks clear. We start with this:

```
\xintverbosetrue
\xintdeffunc foo(x,y,z) = subs(S + S^2, S = x+y+z);
\xintdeffunc bar(x,y,z) = seq(S + S^2, S = x+y+z);
\xintexpr foo(100,10,1), bar(100,10,1)\relax
\xintverbosefalse
```

12432, 12432

It produces in the log:

```
Package xintexpr Info: (on line 10)
    Function foo for \xintexpr parser associated to \XINT_expr_userfunc_foo wit
h meaning macro:#1#2#3->{\xintAdd {\xintAdd {\xintAdd {#1}{#2}}{#3}}{\xintPow {
\xintAdd {\xintAdd {#1}{#2}}{#3}}{2}}}
Package xintexpr Info: (on line 11)
    Function bar for \xintexpr parser associated to \XINT_expr_userfunc_bar wit
h meaning macro:#1#2#3->\expanded \bgroup \expanded {\unexpanded {\XINT_expr_se
q:_b {\xintbareeval S + S^2\relax !S}}{\xintAdd {\xintAdd {#1}{#2}}{#3}}^}
```

Even without understanding all details one sees that in the first case the `\xintAdd {\xintAdd {#1}{#2}}{#3}}` appears twice, and in the second case only once. But in the second case we have a

42

yet to evaluate expression. So the second approach is not much different in its effect than using the more simple-minded \xintNewFunction. Besides one gets a feeling why the function arguments can not appear in the expression but only in the iterated over values, because there is no way to understand what x, y, z are supposed to mean without adding extra structure showing they map to #1, #2, #3.

   The above remarks apply to subsm() and subsn(). Even if they do work in \xintdeffunc context (warning, testing at 1.4 release has remained minimal), they will not bring added efficiency if the substituted values are to be used multiple times. They may still be useful to visually simplify the input of a big expression by expressing it in terms of smaller constituents.

   Another workaround if one wants genuine (not «macro»-) functions for some expression where the same thing is used multiple times is to define helper functions computing the intermediate data. One can see illustrations of this in the code source of xinttrig (or in the matrix multiplication example at the end of this chapter).

2.13.3. \xintdefiifunc

With \xintdeffunc the created function is known by the \xintexpr parser only. For usage in the \xintiiexpr parser, it is required to use \xintdefiifunc.

2.13.4. \xintdeffloatfunc

With \xintdeffunc the created function is known by the \xintexpr parser only. For usage in the \xintfloatexpr parser, it is required to use \xintdeffloatfunc.

   Note: the optional argument [Q] accepted by \xintfloatexpr does not work with \xintdeffloatfunc. It is still possible to wrap the expression in float(expression,Q), if it evaluates to a scalar.

2.13.5. Deprecated: \xintdefefunc, \xintdefiiefunc, \xintdeffloatefunc

Changed at 1.4! They are deprecated and currently only aliases to \xintdeffunc et al.. Please update your documents as they may be removed at any time.

2.13.6. \xintdefufunc, \xintdefiiufunc, \xintdeffloatufunc

New with 1.4 This allows to define so-called «Universal functions». This is terminology borrowed from NumPy.
   Here is an example:

```
\xintdefiivar Array = ndmap(lcm, 1..5; 1..10; 1..10);
Array = \xintthealign\xintiiexpr Array\relax
\xintdefiiufunc foo(x) = x^3;
\begin{figure}[htbp]
\caption{Output of a universal function acting on an array}\label{fig:ufunc}
\centeredline{$\vcenter{\xintthealign\xintiiexpr foo(Array)\relax}$}
\end{figure}
See \autopageref{fig:ufunc} for the output.
```

```
Array =
[[[  1,    2,    3,    4,    5,    6,    7,    8,    9,    10   ],
  [  2,    2,    6,    4,    10,   6,    14,   8,    18,   10   ],
  [  3,    6,    3,    12,   15,   6,    21,   24,   9,    30   ],
  [  4,    4,    12,   4,    20,   12,   28,   8,    36,   20   ],
  [  5,    10,   15,   20,   5,    30,   35,   40,   45,   10   ],
  [  6,    6,    6,    12,   30,   6,    42,   24,   18,   30   ],
  [  7,    14,   21,   28,   35,   42,   7,    56,   63,   70   ],
  [  8,    8,    24,   8,    40,   24,   56,   8,    72,   40   ],
  [  9,    18,   9,    36,   45,   18,   63,   72,   9,    90   ],
  [  10,   10,   30,   20,   10,   30,   70,   40,   90,   10   ]],
```

```
[[  2,    2,    6,    4,    10,   6,    14,   8,    18,    10   ],
 [  2,    2,    6,    4,    10,   6,    14,   8,    18,    10   ],
 [  6,    6,    6,    12,   30,   6,    42,   24,   18,    30   ],
 [  4,    4,    12,   4,    20,   12,   28,   8,    36,    20   ],
 [  10,   10,   30,   20,   10,   30,   70,   40,   90,    10   ],
 [  6,    6,    6,    12,   30,   6,    42,   24,   18,    30   ],
 [  14,   14,   42,   28,   70,   42,   14,   56,   126,   70   ],
 [  8,    8,    24,   8,    40,   24,   56,   8,    72,    40   ],
 [  18,   18,   18,   36,   90,   18,   126,  72,   18,    90   ],
 [  10,   10,   30,   20,   10,   30,   70,   40,   90,    10   ]],
[[  3,    6,    3,    12,   15,   6,    21,   24,   9,     30   ],
 [  6,    6,    6,    12,   30,   6,    42,   24,   18,    30   ],
 [  3,    6,    3,    12,   15,   6,    21,   24,   9,     30   ],
 [  12,   12,   12,   12,   60,   12,   84,   24,   36,    60   ],
 [  15,   30,   15,   60,   15,   30,   105,  120,  45,    30   ],
 [  6,    6,    6,    12,   30,   6,    42,   24,   18,    30   ],
 [  21,   42,   21,   84,   105,  42,   21,   168,  63,    210  ],
 [  24,   24,   24,   24,   120,  24,   168,  24,   72,    120  ],
 [  9,    18,   9,    36,   45,   18,   63,   72,   9,     90   ],
 [  30,   30,   30,   60,   30,   30,   210,  120,  90,    30   ]],
[[  4,    4,    12,   4,    20,   12,   28,   8,    36,    20   ],
 [  4,    4,    12,   4,    20,   12,   28,   8,    36,    20   ],
 [  12,   12,   12,   12,   60,   12,   84,   24,   36,    60   ],
 [  4,    4,    12,   4,    20,   12,   28,   8,    36,    20   ],
 [  20,   20,   60,   20,   20,   60,   140,  40,   180,   20   ],
 [  12,   12,   12,   12,   60,   12,   84,   24,   36,    60   ],
 [  28,   28,   84,   28,   140,  84,   28,   56,   252,   140  ],
 [  8,    8,    24,   8,    40,   24,   56,   8,    72,    40   ],
 [  36,   36,   36,   36,   180,  36,   252,  72,   36,    180  ],
 [  20,   20,   60,   20,   20,   60,   140,  40,   180,   20   ]],
[[  5,    10,   15,   20,   5,    30,   35,   40,   45,    10   ],
 [  10,   10,   30,   20,   10,   30,   70,   40,   90,    10   ],
 [  15,   30,   15,   60,   15,   30,   105,  120,  45,    30   ],
 [  20,   20,   60,   20,   20,   60,   140,  40,   180,   20   ],
 [  5,    10,   15,   20,   5,    30,   35,   40,   45,    10   ],
 [  30,   30,   30,   60,   30,   30,   210,  120,  90,    30   ],
 [  35,   70,   105,  140,  35,   210,  35,   280,  315,   70   ],
 [  40,   40,   120,  40,   40,   120,  280,  40,   360,   40   ],
 [  45,   90,   45,   180,  45,   90,   315,  360,  45,    90   ],
 [  10,   10,   30,   20,   10,   30,   70,   40,   90,    10   ]]]
```

See for the output.

The function can be applied to any nested strucure:

```
\xintiiexpr foo([1, [2, [3, [4, [5, 6, 7, 8, 9, 10]]]]])\relax
```
[1, [8, [27, [64, [125, 216, 343, 512, 729, 1000]]]]]

It must be defined as function acting on scalars, but its value type is not constrained.

```
\xintdefiivar Array = [1..10];
\xintdefiiufunc foo(x) = [1..x];
\xintthealign\xintiiexpr foo(Array)\relax
```
```
[[  1   ],
 [  1,   2    ],
 [  1,   2,   3    ],
 [  1,   2,   3,   4    ],
 [  1,   2,   3,   4,   5    ],
```

Figure 1: Output of a universal function acting on an array

```
[[[ 1,      8,       27,       64,        125,       216,       343,      512,        729,        1000      ],
  [ 8,      8,       216,      64,        1000,      216,       2744,     512,        5832,       1000      ],
  [ 27,     216,     27,       1728,      3375,      216,       9261,     13824,      729,        27000     ],
  [ 64,     64,      1728,     64,        8000,      1728,      21952,    512,        46656,      8000      ],
  [ 125,    1000,    3375,     8000,      125,       27000,     42875,    64000,      91125,      1000      ],
  [ 216,    216,     216,      1728,      27000,     216,       74088,    13824,      5832,       27000     ],
  [ 343,    2744,    9261,     21952,     42875,     74088,     343,      175616,     250047,     343000    ],
  [ 512,    512,     13824,    512,       64000,     13824,     175616,   512,        373248,     64000     ],
  [ 729,    5832,    729,      46656,     91125,     5832,      250047,   373248,     729,        729000    ],
  [ 1000,   1000,    27000,    8000,      1000,      27000,     343000,   64000,      729000,     1000      ]],
 [[ 8,      8,       216,      64,        1000,      216,       2744,     512,        5832,       1000      ],
  [ 8,      8,       216,      64,        1000,      216,       2744,     512,        5832,       1000      ],
  [ 216,    216,     216,      1728,      27000,     216,       74088,    13824,      5832,       27000     ],
  [ 64,     64,      1728,     64,        8000,      1728,      21952,    512,        46656,      8000      ],
  [ 1000,   1000,    27000,    8000,      1000,      27000,     343000,   64000,      729000,     1000      ],
  [ 216,    216,     216,      1728,      27000,     216,       74088,    13824,      5832,       27000     ],
  [ 2744,   2744,    74088,    21952,     343000,    74088,     2744,     175616,     2000376,    343000    ],
  [ 512,    512,     13824,    512,       64000,     13824,     175616,   512,        373248,     64000     ],
  [ 5832,   5832,    5832,     46656,     729000,    5832,      2000376,  373248,     5832,       729000    ],
  [ 1000,   1000,    27000,    8000,      1000,      27000,     343000,   64000,      729000,     1000      ]],
 [[ 27,     216,     27,       1728,      3375,      216,       9261,     13824,      729,        27000     ],
  [ 216,    216,     216,      1728,      27000,     216,       74088,    13824,      5832,       27000     ],
  [ 27,     216,     27,       1728,      3375,      216,       9261,     13824,      729,        27000     ],
  [ 1728,   1728,    1728,     1728,      216000,    1728,      592704,   13824,      46656,      216000    ],
  [ 3375,   27000,   3375,     216000,    3375,      27000,     1157625,  1728000,    91125,      27000     ],
  [ 216,    216,     216,      1728,      27000,     216,       74088,    13824,      5832,       27000     ],
  [ 9261,   74088,   9261,     592704,    1157625,   74088,     9261,     4741632,    250047,     9261000   ],
  [ 13824,  13824,   13824,    13824,     1728000,   13824,     4741632,  13824,      373248,     1728000   ],
  [ 729,    5832,    729,      46656,     91125,     5832,      250047,   373248,     729,        729000    ],
  [ 27000,  27000,   27000,    216000,    27000,     27000,     9261000,  1728000,    729000,     27000     ]],
 [[ 64,     64,      1728,     64,        8000,      1728,      21952,    512,        46656,      8000      ],
  [ 64,     64,      1728,     64,        8000,      1728,      21952,    512,        46656,      8000      ],
  [ 1728,   1728,    1728,     1728,      216000,    1728,      592704,   13824,      46656,      216000    ],
  [ 64,     64,      1728,     64,        8000,      1728,      21952,    512,        46656,      8000      ],
  [ 8000,   8000,    216000,   8000,      8000,      216000,    2744000,  64000,      5832000,    8000      ],
  [ 1728,   1728,    1728,     1728,      216000,    1728,      592704,   13824,      46656,      216000    ],
  [ 21952,  21952,   592704,   21952,     2744000,   592704,    21952,    175616,     16003008,   2744000   ],
  [ 512,    512,     13824,    512,       64000,     13824,     175616,   512,        373248,     64000     ],
  [ 46656,  46656,   46656,    46656,     5832000,   46656,     16003008, 373248,     46656,      5832000   ],
  [ 8000,   8000,    216000,   8000,      8000,      216000,    2744000,  64000,      5832000,    8000      ]],
 [[ 125,    1000,    3375,     8000,      125,       27000,     42875,    64000,      91125,      1000      ],
  [ 1000,   1000,    27000,    8000,      1000,      27000,     343000,   64000,      729000,     1000      ],
  [ 3375,   27000,   3375,     216000,    3375,      27000,     1157625,  1728000,    91125,      27000     ],
  [ 8000,   8000,    216000,   8000,      8000,      216000,    2744000,  64000,      5832000,    8000      ],
  [ 125,    1000,    3375,     8000,      125,       27000,     42875,    64000,      91125,      1000      ],
  [ 27000,  27000,   27000,    216000,    27000,     27000,     9261000,  1728000,    729000,     27000     ],
  [ 42875,  343000,  1157625,  2744000,   42875,     9261000,   42875,    21952000,   31255875,   343000    ],
  [ 64000,  64000,   1728000,  64000,     64000,     1728000,   21952000, 64000,      46656000,   64000     ],
  [ 91125,  729000,  91125,    5832000,   91125,     729000,    31255875, 46656000,   91125,      729000    ],
  [ 1000,   1000,    27000,    8000,      1000,      27000,     343000,   64000,      729000,     1000      ]]]
```

```
[ 1, 2, 3, 4, 5, 6   ],
[ 1, 2, 3, 4, 5, 6, 7   ],
[ 1, 2, 3, 4, 5, 6, 7, 8   ],
[ 1, 2, 3, 4, 5, 6, 7, 8, 9   ],
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]]
```
  It is even allowed to produce oples and act on oples:
```
  \xintdefiivar Ople = 1..10;
  \xintdefiiufunc bar(x) = x, x^2, x^3;
  \xintiiexpr bar(Ople)\relax
```
1, 1, 1, 2, 4, 8, 3, 9, 27, 4, 16, 64, 5, 25, 125, 6, 36, 216, 7, 49, 343, 8, 64, 512, 9, 81, 729, 10, 100, 1000

2.13.7. Using the same name for both a variable and a function

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

```
  \xintdefiiifunc f(x):=x^3;
  \xinttheiiexpr add(f(f),f=100..120)\relax\newline
  \xintdeffunc f(x,y):=x^2+y^2;
  \xinttheexpr mul(f(f(f,f),f(f,f)),f=1..10)\relax
  \xintunassigniiexprfunc{f}\xintunassignexprfunc{f}%
```
28205100
186188134867578885427848806400000000

2.13.8. \xintunassignexprfunc, \xintunassigniiexprfunc, \xintunassignfloatexprfunc

Function names can be unassigned via \xintunassignexprfunc{⟨*name*⟩}, \xintunassigniiexprfunc{⟨*name*⟩}, and \xintunassignfloatexprfunc{⟨*name*⟩}.
```
  \xintunassignexprfunc{e}
  \xintunassignexprfunc{f}
```
  Warning: no check is done to avoid undefining built-in functions...

2.13.9. \ifxintverbose conditional

With \xintverbosetrue the meanings of the functions (or rather their associated macros) will be written to the log. For example the Rump declaration above generates this in the log file:
```
    Function Rump for \xintexpr parser associated to \XINT_expr_userfunc_Rump w
  ith meaning macro:#1#2->{\xintAdd {\xintAdd {\xintAdd {\xintDiv {\xintMul {1335
  }{\xintPow {#2}{6}}}{4}}{\xintMul {\xintPow {#1}{2}}}{\xintSub {\xintSub {\xintS
  ub {\xintMul {11}{\xintMul {\xintPow {#1}{2}}{\xintPow {#2}{2}}}}{\xintPow {#2}
  {6}}}{\xintMul {121}{\xintPow {#2}{4}}}}{2}}}}{\xintDiv {\xintMul {11}{\xintPow
   {#2}{8}}}{2}}}}{\xintDiv {#1}{\xintMul {2}{#2}}}}
```

☞ {

Changed at 1.4!

> The meanings written out to the log for more complicated functions may sometimes use the same character at different locations but with different catcodes.
> It may thus be impossible to retokenize it (even after having removed the extra spaces from the added line breaks).
> This is in contrast with variable values which are always output in the log in the benign way, using digits, braces and some characters of catcode 12.

46

2.13.10. \ifxintglobaldefs conditional

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

2.14. Examples of user defined functions

2.14.1. Example with vectors and matrices

Changed at 1.4! This section has changed significantly at 1.4 due to the new extended data types manipulated by the syntax.

Suppose we want to manipulate 3-dimensional vectors, which will be represented as nut-ples of length 3. And let's add a bit of matrix algebra.

```
\xintdeffunc dprod(V, W) := V[0]*W[0] + V[1]*W[1] + V[2]*W[2];
\xintdeffunc cprod(V, W) := [V[1]*W[2] - V[2]*W[1],
                             V[2]*W[0] - V[0]*W[2],
                             V[0]*W[1] - V[1]*W[0]];
\xintdeffunc Det3(U, V, W) := dprod(cprod(U, V), W);
\xintdeffunc DetMat(M) = Det3(*M);
\xintdeffunc RowMat(U, V, W) := [U, V, W];
\xintdeffunc ColMat(U, V, W) := [[U[0], V[0], W[0]],
                                 [U[1], V[1], W[1]],
                                 [U[2], V[2], W[2]]];
\xintdeffunc MatMul(A, B) :=
 [[A[0,0]*B[0,0]+A[0,1]*B[1,0]+A[0,2]*B[2,0],
   A[0,0]*B[0,1]+A[0,1]*B[1,1]+A[0,2]*B[2,1],
   A[0,0]*B[0,2]+A[0,1]*B[1,2]+A[0,2]*B[2,2]],
  [A[1,0]*B[0,0]+A[1,1]*B[1,0]+A[1,2]*B[2,0],
   A[1,0]*B[0,1]+A[1,1]*B[1,1]+A[1,2]*B[2,1],
   A[1,0]*B[0,2]+A[1,1]*B[1,2]+A[1,2]*B[2,2]],
  [A[2,0]*B[0,0]+A[2,1]*B[1,0]+A[2,2]*B[2,0],
   A[2,0]*B[0,1]+A[2,1]*B[1,1]+A[2,2]*B[2,1],
   A[2,0]*B[0,2]+A[2,1]*B[1,2]+A[2,2]*B[2,2]]];

\xintdefvar vec1, vec2, vec3 := [1, 1, 1], [1, 1/2, 1/4], [1, 1/3, 1/9];
\xintdefvar mat1 = RowMat(vec1, vec2, vec3);
\xintdefvar mat2 = ColMat(vec1, vec2, vec3);
\xintdefvar mat12 = MatMul(mat1,mat2);
\xintdefvar mat21 = MatMul(mat2,mat1);
Some computations (|align| executes multiple times hence we pre-computed!):
\begin{align*}
   M_1      &= \vcenter{\xintthealign \xintexpr mat1\relax}&&\qquad
   M_2 . M_1 = \vcenter{\xintthealign \xintexpr mat21\relax}\\[3\jot]
   M_2      &= \vcenter{\xintthealign \xintexpr mat2\relax}&&\qquad
   M_1 . M_2 = \vcenter{\xintthealign \xintexpr mat12\relax}
\end{align*}
$$
   \det(M_1)     = \xinteval{DetMat(mat1)},\quad
   \det(M_1.M_2) = \xinteval{reduce(DetMat(mat12))},\quad
   \det(M_2.M_1) = \xinteval{reduce(DetMat(mat21))}
$$
```

Some computations (align executes multiple times hence we pre-computed!):

$$M_1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1/2 & 1/4 \\ 1 & 1/3 & 1/9 \end{bmatrix}, \qquad M_2.M_1 = \begin{bmatrix} 3 & 11/6 & 49/36 \\ 11/6 & 49/36 & 251/216 \\ 49/36 & 251/216 & 1393/1296 \end{bmatrix},$$

$$M_2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1/2 & 1/3 \\ 1 & 1/4 & 1/9 \end{bmatrix}, \qquad M_1.M_2 = \begin{bmatrix} 3 & 7/4 & 13/9 \\ 7/4 & 21/16 & 43/36 \\ 13/9 & 43/36 & 91/81 \end{bmatrix}$$

$$\det(M_1) = -1/18, \quad \det(M_1.M_2) = 1/324, \quad \det(M_2.M_1) = 1/324$$

For some hair-raising experience check the `\xintverbosetrue` output in the log... here is an alternative with two (three, counting dprod()) helper functions:

```
% annoying that Tr also starts Trace, but Spur is available
% well Sp also starts Spectrum. Big problems.
\xintdeffunc Tr(M) :=
     [[M[0,0], M[1,0], M[2,0]],
      [M[0,1], M[1,1], M[2,1]],
      [M[0,2], M[1,2], M[2,2]]];

\xintdeffunc MatMul_a(r1, r2, r3, c1, c2, c3) :=
     [[dprod(r1, c1), dprod(r1, c2), dprod(r1, c3)],
      [dprod(r2, c1), dprod(r2, c2), dprod(r2, c3)],
      [dprod(r3, c1), dprod(r3, c2), dprod(r3, c3)]];

\xintdeffunc MatMul(A, B) := MatMul_a(*A, *Tr(B));
```

And once we have the transpose and the scalar product of vectors, we can simply use ndmap() for a lean syntax (this would extend to arbitrary dimension):

```
\xintdeffunc MatMul(A, B) = ndmap(dprod, *A; *Tr(B));
\xintdefvar mat1212 = MatMul(mat12, mat12);
\begingroup
\def\xintexprPrintOne       #1{\xintFrac{#1}}% (no need for \protected with \xintFrac)
\def\xintexpralignbegin       {\begin{pmatrix}}%
\def\xintexpralignend         {\end{pmatrix}}%
\def\xintexpralignlinesep     {\noexpand\\[2\jot]}% needed to counteract an internal \expanded
\def\xintexpralignninnersep   {&}%
\let\xintexpralignleftbracket\empty  \let\xintexpralignleftsep\empty
\let\xintexpralignrightbracket\empty \let\xintexpralignrightsep\empty
$$ \xintthealign \xintexpr mat1\relax \cdot \xintthealign \xintexpr mat2\relax \cdot
   \xintthealign \xintexpr mat1\relax \cdot \xintthealign \xintexpr mat2\relax =
   \xintthealign \xintexpr mat12\relax ^2 = \xintthealign \xintexpr mat1212\relax$$
$$ \det(M_1\cdot M_2 \cdot M_1 \cdot M_2) = \xinteval{reduce(DetMat(mat1212))}$$
\endgroup
```

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & \frac{1}{2} & \frac{1}{4} \\ 1 & \frac{1}{3} & \frac{1}{9} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & \frac{1}{2} & \frac{1}{3} \\ 1 & \frac{1}{4} & \frac{1}{9} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & \frac{1}{2} & \frac{1}{4} \\ 1 & \frac{1}{3} & \frac{1}{9} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & \frac{1}{2} & \frac{1}{3} \\ 1 & \frac{1}{4} & \frac{1}{9} \end{pmatrix} = \begin{pmatrix} 3 & \frac{7}{4} & \frac{13}{9} \\ \frac{7}{4} & \frac{21}{16} & \frac{43}{36} \\ \frac{13}{9} & \frac{43}{36} & \frac{91}{81} \end{pmatrix}^2 = \begin{pmatrix} \frac{18337}{1296} & \frac{48067}{5184} & \frac{93853}{11664} \\ \frac{48067}{5184} & \frac{128809}{20736} & \frac{253687}{46656} \\ \frac{93853}{11664} & \frac{253687}{46656} & \frac{501289}{104976} \end{pmatrix}$$

$$\det(M_1 \cdot M_2 \cdot M_1 \cdot M_2) = \frac{1}{104976}$$

2.14.2. Example with the Rump test

Let's try out our Rump() function:

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

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

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

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

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

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

 8 7.0000000e29
 9 -1.00000000e28
10 5.000000000e27
11 -3.0000000000e26
12 4.00000000000e25
13 3.000000000000e24
14 3.0000000000000e23
15 -2.00000000000000e22
16 1.000000000000000e21
17 -5.0000000000000000e20
18 1.17260394005317863
19 1.000000000000000001e18
20 -9.9999999999999998827e16
21 1.0000000000000000011726e16
22 3.000000000000001172604e15
23 -9.999999999998827396060e13
24 -1.9999999999999988273960599e13
25 -1.99999999999988273960599947e12
26 1.17260394005317786318588349
27 -5.99999999988273960599468214e10
28 -9.99999998827396059946821368le8
29 2.0000000117260394005317863186e8
30 1.00000011726039400531786318588e7
31 -999998.82739605994682136814116510
32 200001.172603940053178631858834900
33 -9998.82739605994682136814116509548
34 -1998.82739605994682136814116509548e0
35 -198.82739605994682136814116509547982
36 21.17260394005317786318588349045201837
37 -0.82739605994682136814116509547981629200
38 -0.82739605994682136814116509547981629200
39 -0.827396059946821368141165095479816292000
40 -0.82739605994682136814116509547981629199990

2.14.3. Examples of recursive definitions

Recursive definitions *require* using the short-circuit branching operators. Prior to 1.4, to the contrary it was explained that one should use the if() or ifsgn() functions and that they would get converted into macros doing branching in a short-circuit manner. This was a bit counter-intuitive.

49

```
\xintdeffunc GCD(a,b):=(b)?{GCD(b,a/:b)}{a};
```
This of course is the Euclide algorithm: it will be here applied to variables which may be fractions. For example:
```
\xinttheexpr GCD(385/102, 605/238)\relax
```
55/714
There is already a built-in gcd() (which accepts arbitrarily many arguments):
```
\xinttheexpr gcd(385/102, 605/238)\relax
```
55/714
Our second example is modular exponentiation:
```
\xintdefiiifunc powmod_a(x, m, n) :=
      isone(m)?
          % m=1, return x modulo n
          {   x /: n  }
          % m > 1 test if odd or even and do recursive call
          {   odd(m)? {  x*sqr(powmod_a(x, m//2, n)) /: n }
                      {     sqr(powmod_a(x, m//2, n)) /: n }
            }
        ;
\xintdefiiifunc powmod(x, m, n) := (m)?{powmod_a(x, m, n)}{1};
```
I have made the definition here for the \xintiiexpr parser; we could do the same for the \xintexpr-parser (but its usage with big powers would quickly create big denominators, think powmod(1/2, 1000, 1) for example.)
```
\xinttheiiexpr seq(powmod(x, 1000, 128), x=9, 11, 13, 15, 17, 19, 21)\relax\par
```
65, 97, 33, 1, 1, 33, 97
The function assumes the exponent is non-negative (the Python pow behaved the same until 3.8 release), but zealous users will add the necessary code for negative exponents, after having defined another function for modular inverse!

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

2.15. Links to some (old) examples within this document

- The utilities provided by xinttools (section 14), some completely expandable, others not, are of independent interest. Their use is illustrated through various examples: among those, it is shown in subsection 16.8 how to implement in a completely expandable way the Quick Sort algorithm and also how to illustrate it graphically. Other examples include some dynamically constructed alignments with automatically computed prime number cells: one using a completely expandable prime test and \xintApplyUnbraced (subsection 16.2), another one with \xintFor* (subsection 16.6).

- One has also a computation of primes within an \edef (subsection 14.15), with the help of \xintiloop. Also with \xintiloop an automatically generated table of factorizations (subsection 16.5).

- The code for the title page fun with Fibonacci numbers is given in subsection 15.18 with \xintFor* joining the game.

- The computations of π and $\log 2$ (subsection 12.11) using xint and the computation of the convergents of e with the further help of the xintcfrac package are among further examples.

- Also included, an expandable implementation of the Brent-Salamin algorithm for evaluating π.

- The subsection 16.4 implements expandably the Miller-Rabin pseudo-primality test.

- The functionalities of xintexpr are illustrated with various other examples, in subsubsection 2.13.2, Functions with dummy variables, subsection 16.1 or Recursive definitions.

50

3. Macros of the **xinttrig** package

This package provides trigonometric functions for use with xintexpr. The sole macro is \xintreloadxinttrig.

This package was first included in release 1.3e (2019/04/05) of xintexpr. It is automatically loaded by xintexpr.

Acknowledgements: I finally decided to release some such functions under friendly pressure of Jürgen Gɪʟɢ and Thomas Sᴏ̈ʟʟ, let them both be thanked here.

3.1. \xintreloadxinttrig

The library is loaded automatically by xintexpr, thus with the default value of \xinttheDigits which is 16. But it can handle a precision of up to about 60 digits (make this 59 at most for the inverse functions). For this, execute for example \xintSetDigits*{48}.

☞ {

Since 1.3f, the starred variants \xintDigits*:=P; and \xintSetDigits*{⟨*num. expression*⟩} execute \xintreloadxinttrig to let the package re-configure itself.

The non-starred variants do not execute \xintreloadxinttrig (to avoid adding artificial overhead to existing documents).

}

Absence of guard digits (whether in the used hard-coded constants or in passing over values from one auxiliary function to the next) due to high level (user) interface used for the programming means that the produced values are definitely expected to be wrong in the last digit or last two digits. I should actually give some estimate of the actual maximal error in ulps unit, but I have not done the complete analysis for lack of time.

Final computation results should thus probably be printed via \xintfloateval{[-2]....} in order to strip off (with rounding) the last two digits, if one does not like seeing those non-meaningful figures in the last one or two positions (I don't say those last two figures are *systematically* off). For example, to achieve 16 digits of precision one should work with a precision of 18 digits (being careful to have issued \xintreloadxinttrig) and round results using \xintfloateval{[-2].₂ ...}.

Another approach is to use \xintieval{[D]...} for conversion to a fixed point format.

In future, lower level coding will probably replace the high-level interface, or at least the macros produced by the high-level interface will be hacked into to tell the float macros to work at a somewhat elevated precision.

3.2. Constants

They are the correct rounding to \xinttheDigits precision of the mathematically exact ones. Their values get incorporated into the trigonometrical functions at the time of their definitions during loading or reloading of the package. They are left free to use, or modified, or \xintunassignvar'd, as this will have no impact whatsoever on the functions.

twoPi what could that be?

threePiover2

Pi

Piover2

oneRadian this is one radian in degrees: $180/\pi$

oneDegree this is one degree in radian: $\pi/180$

invfact2 this is 1/2!

invfact3 this is 1/3!

...

invfact44 this is 1/44!

For a (very) slight optimization of usage, it is recommended to convert them to macro form, for example:

```
\edef\oneDegree{\xintfloatexpr oneDegree\relax}
\xintfloateval{sin(37\oneDegree)}\newline
\xintfloateval{sind(37)}\newline
```

0.6018150231520484
0.6018150231520484

By the way, the above value differs by 1ulp from correct rounding of exact one (which looks ...52048279917...), see subsection 3.4.

3.3. Functions

3.3.1. Direct trigonometry

With the variable in radians:

sin(x) sine

cos(x) cosine

tan(x) tangent

cot(x) cotangent

sec(x) secant

csc(x) cosecant

With the variable in degrees:

sind(x) sine

cosd(x) cosine

tand(x) tangent

cotd(x) cotangent

secd(x) secant

cscd(x) cosecant

Only available with the variable in radians:

tg(x) tangent

cotg(x) cotangent

sinc(x) cardinal sine sinc(x) = sin(x)/x

3.3.2. Inverse trigonometry

With the value in radians:

asin(x) arcsine

acos(x) arccosine

atan(x) arctangent

Arg(x, y) the main branch of the argument of the complex number x+iy, from $-\pi$ (excluded) to π (included). As the output is rounded -Pi is a possible return value.

pArg(x, y) the branch of the argument of the complex number x+iy with values going from 0 (included) to 2π (excluded). Inherent rounding makes twoPi a possible return value.

atan2(y, x) it is Arg(x, y). Note the reversal of the arguments, this seems to be the most frequently encountered convention across languages.

   With the value in degrees:

asind(x) arcsine

acosd(x) arccosine

atand(x) arctangent

Argd(x, y) the main branch of the argument of the complex number x+iy, from -180 (excluded) to 180 (included). Inherent rounding of output can cause -180 to be returned.

pArgd(x, y) the branch of the argument of the complex number x+iy with values going from 0 (included) to 360 (excluded). Inherent rounding of output can cause 360 to be returned.

atan2d(y, x) it is Argd(x, y). Note the reversal of the arguments, this seems to be the most frequently encountered convention across languages.

3.3.3. Conversion functions (optional definitions left to user decision)

Python provides functions degrees() and radians(). But as most of the xinttrig functions are already defined for the two units, I felt this was not really needed. It is a oneliner to add them:

```
\xintdeffloatfunc radians(x) := x * oneDegree;
\xintdeffloatfunc degrees(x) := x * oneRadian;
\xintdefefunc radians(x) := float_(x * oneDegree);
\xintdefefunc degrees(x) := float_(x * oneRadian);
```

   The \xintexpr-variants above do an exact multiplication. The float_() explicit rounding is a bit superfluous as anyhow trigonometrical functions do this initial rounding of their arguments.

   The conversion factors above are without guard digits. Probably better to work overall with an elevated precision and print final results at a lower precision.

3.4. Important implementation notes

- The package is almost entirely implemented using the high level user interface of xintexpr, using \xintdeffloatefunc (merged with \xintdeffloatfunc at 1.4) (and \xintdeffloatvar), the main two exceptions being for:

  1. the range reduction for the sind() and cosd() functions which required for optimized efficiency the coding at some more core level.

2. some changes at core level added at 1.3e to facilitate the transfer of the defined functions from the float parser to the exact parser.

   Also, to avoid problems if the package is reloaded at a time the user has used some letter variables as assigned variables, 1.3e added \xintensuredummy and \xintrestorevariable.

- It is not possible from this interface to (easily) let the computation proceed with a temporarily elevated precision (``guard digits''). Expect thus some errors in the last places; basically one should use the optional rounding argument of either \xintfloateval or \xintieval to reduce the number of digits of printed values by about two digits, if one hopes to get correct rounding (most of the time).

- Currently, xint is lacking some dedicated internal representation of floats which means that most operations re-parse the digit tokens of their arguments to count them... this does not contribute to efficiency (you can load the module under \xintverbosetrue regime and see how the nested macros look like and get an idea of how many times some rather silly re-counting of mantissa lengths will get done!)

- One should not overwrite some function names which are employed as auxiliaries: sin_aux, cos_aux, sin_, cos_, sind_, cosd_, asin_l...others...asin_a, asind_a, atan_a, atand_a, atan_b, atand_b. If you redefine any one of them, you break the whole thing.

- Floats with large exponents are integers and are multiple of 1000; hence modulo 360 all such ``angles'' are multiple of 40 degrees. Needless to say that considering usage of the sind() and cosd() functions with such large float numbers is meaningless.

- Regarding sin() and cos(), xinttrig converts their argument to degrees by multiplication by (pre-rounded) $180/\pi$, then does range reduction modulo 360 and finally goes back to radians in the appropriate octants to use usual Taylor series (roughly said). For large floats, the output value will thus be one of sind(40n), cosd(40n), n=0..8. If the unit in the last place of original variable was for example 1e9 the final result means nothing at all: the unit in the last place interval extends above possibly astronomical numbers of intervals of length 2π.

   This intrinsic problem is not a by-product of conversion problems to and from degrees, it is an in-built inadequacy of the concept of floating point numbers to provide meaning to evaluating trigonometrical functions. The argument should be treated as a uniformly distributed random variable modulo 2π, and the sine and cosine values should be random variables realizing the value distribution of these mathematical functions. Clearly this adds some (rather severe) implementation complications such as deciding how to make the transition to randomness. Too lazy for that.

   Opting for a random value also raises the question of how to deal with multiple such evaluations at the same argument in a single expression. I would argue again that as it is evil to consider meaningless quantities, it is not a problem if new compilations give different results, or even single compilation gives different results in various parts of the same formula, that's the whole point of randomness! As said already, I got too lazy to consider seriously implementing such a non-standard philosophy, despite its compelling soundness.

- Did I say the implementation was done at very high level (for the most part), hence has ample room for optimization? This is particularly the case for the handling of small inputs by functions such as sine or arcsine.

3.5. Some example evaluations

```
\xintDigits* := 50\relax % target 48 digits via 2 guard digits
$sind(17)\approx\xintfloateval{[-2] sind(17)}$\newline
```

```
$cosd(17)\approx\xintfloateval{[-2] cosd(17)}$\newline
$tand(17)\approx\xintfloateval{[-2] tand(17)}$\newline
$sind(43)\approx\xintfloateval{[-2] sind(43)}$\newline
$cosd(43)\approx\xintfloateval{[-2] cosd(43)}$\newline
$tand(43)\approx\xintfloateval{[-2] tand(43)}$\newline
$asind(0.3)\approx\xintfloateval{[-2] asind(0.3)}$\newline
$acosd(0.3)\approx\xintfloateval{[-2] acosd(0.3)}$\newline
$atand(3)\approx\xintfloateval{[-2] atand(3)}$\newline
$tan(atan(7))\approx\xintfloateval{[-2] tan(atan(7))}$\newline
$asind(sind(25))\approx\xintfloateval{[-2] asind(sind(25))}$\par\medskip
\noindent\xintDigits* := 26\relax % target 24 digits via 2 guard digits
$sind(17)\approx\xintfloateval{[-2] sind(17)}$\newline
$cosd(17)\approx\xintfloateval{[-2] cosd(17)}$\newline
$tand(17)\approx\xintfloateval{[-2] tand(17)}$\newline
$sind(43)\approx\xintfloateval{[-2] sind(43)}$\newline
$cosd(43)\approx\xintfloateval{[-2] cosd(43)}$\newline
$tand(43)\approx\xintfloateval{[-2] tand(43)}$\newline
$asind(0.3)\approx\xintfloateval{[-2] asind(0.3)}$\newline
$acosd(0.3)\approx\xintfloateval{[-2] acosd(0.3)}$\newline
$atand(3)\approx\xintfloateval{[-2] atand(3)}$\newline
$tan(atan(7))\approx\xintfloateval{[-2] tan(atan(7))}$\newline
$asind(sind(25))\approx\xintfloateval{[-2] asind(sind(25))}$\par
\xintDigits* := 16\relax
```

sind(17) ≈ 0.292371704722736728097468695377143252664687186183
cosd(17) ≈ 0.956304755963035481338650816618418962009410343991
tand(17) ≈ 0.305730681458660355734541958996550716146250221387
sind(43) ≈ 0.681998360062498500442225784711125580340433802762
cosd(43) ≈ 0.731353701619170483287543608275622403378396544763
tand(43) ≈ 0.932515086137661705612185627426186654353537299494
asind(0.3) ≈ 17.4576031237220922902460457924449418216636440147
acosd(0.3) ≈ 72.5423968762779077097539542075550581783363559853
atand(3) ≈ 71.5650511770779893515721937204532946712042142996
tan(atan(7)) ≈ 7.00000000000000000000000000000000000000000000000
asind(sind(25)) ≈ 25.00000000000000000000000000000000000000000000000

sind(17) ≈ 0.292371704722736728097469
cosd(17) ≈ 0.956304755963035481338651
tand(17) ≈ 0.305730681458660355734542
sind(43) ≈ 0.681998360062498500442226
cosd(43) ≈ 0.731353701619170483287544
tand(43) ≈ 0.932515086137661705612186
asind(0.3) ≈ 17.4576031237220922902460
acosd(0.3) ≈ 72.5423968762779077097540
atand(3) ≈ 71.5650511770779893515722
tan(atan(7)) ≈ 7.000000000000000000000000
asind(sind(25)) ≈ 25.00000000000000000000000

4. Macros of the xintlog package

This package provides logarithms, exponentials and fractional powers for use with xintexpr.

This package was first included in release 1.3e (2019/04/05) of xintexpr. It is automatically loaded by xintexpr.

Currently, the functions log10(), pow10(), log(), exp(), and pow() use at their core two fast expandable macros handling base 10 logarithms and powers for mantissas of 9 digit tokens. They are defined by package poormanlog which is automatically imported. The error is believed to be at most 2ulp (see its README). The package poormanlog has no dependencies and can be imported by any other TeX macro file.

Although the precision is thus limited to about 8 or 9 digits this is amply enough for plots.

4.1. \poormanloghack

\poormanloghack{}** use it to let the ** operator be remapped to the pow() function.

\poormanloghack{^} use it to let the ^ operator be remapped to the pow() function.

If used, they obey TeX scoping as usual.
```
\begingroup
\poormanloghack{**}\xintfloateval{[8]1.234**5.678}\newline
\poormanloghack{^}\xintfloateval{[8]1.234^5.678}\par
\endgroup
```
3.2997989
3.2997989

Notice that in \xintfloateval those (equivalent) operators already natively handle half-integer exponents. Once remapped to the pow() function they will become less precise than the original ones for half-integer and integer exponents.

4.2. Functions

All those functions achieve only about 8 or 9 digits of precision. Notice in particular that the digits beyond the ninth printed by log() have no significance (here we suppose 1<x<10), but I did not add the rounding overhead as it is expected anyhow that the final result will be appropriately rounded. Notice however that log10() should be seen as going from floating point to fixed point (in the sense of the number of fractional digits) and pow10() from fixed point to floating point.

log10(x) logarithm in base 10

pow10(x) fractional powers of 10

log(x) natural logarithm via log10(x)*2.3025850923 formula; only the first 8 or 9 digits of the output are significant...

exp(x) exponential function via pow10(x*0.434294481903) formula

pow(x, y) computes x^y via the formula pow10(y*log10(x))

```
\xintfloateval{[9] log(2), exp(1), pow(2,0.5)}
```
0.693147179, 2.71828183, 1.41421356 Notice that the last digit of log(2) is not the correctly rounded one... I did say 9 **or** 8 digits or precision... The documentation of poormanlog mentions an error of up to 2 units in the ninth digit when computing log10(x) for 1<x<10 and 10^x for 0<x<1.

Part II.
The macro layer for expandable computations: xintcore, xint, xintfrac,...

```
    WARNING !
    The documentation is getting old, and is in need of rewrites for many sections, particularly
for examples.
    We do try to keep updated the description of macros provided by the packages.
```

5. The xint bundle

5.1. Characteristics

```
    The main characteristics are:

  1. exact algebra on ``big numbers'', integers as well as fractions,
```

> 2. floating point variants with user-chosen precision,
>
> 3. the computational macros are compatible with expansion-only context,
>
> 4. the bundle comes with parsers (integer-only, or handling fractions, or doing floating point computations) of infix operations implementing beyond infix operations extra features such as dummy variables.
>
> Since 1.2 ``big numbers'' must have less than about 19950 digits: the maximal number of digits for addition is at 19968 digits, and it is 19959 for multiplication. The reasonable range of use of the package is with numbers of up to a few hundred digits.[17]
> TeX does not know off-hand how to print on the page such very long numbers, see subsection 1.4.

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

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

The bigintcalc package by Heiko Oberdiek provided expandable macros (using some of \numexpr possibilities, when available) on arbitrarily big integers, beyond the TeX bound. It does not provide an expression parser.[18] xint did it again using more of \numexpr for higher speed, and in a later evolution added handling of exact fractions, of scientific numbers, and an expression parser. Arbitrary precision floating points operations were added as a derivative, and not part of the initial design goal.

The concept of signed infinities, signed zeroes, NaN's, error traps...,[19] have not been implemented, only the notion of `scientific notation with a given number of significant figures'.[20]

The LaTeX3 project has implemented expandably floating-point computations with 16 significant figures (l3fp), including functions such as exp, log, sine and cosine.[21]

More directly related to the xint bundle there is the l3bigint package, also devoted to big integers and in development a.t.t.o.w (2015/10/09, no division yet). It is part of the experimental trunk of the LaTeX3 Project and provides an expression parser for expandable arithmetic with big integers. Its author Bruno Le Floch succeeded brilliantly into implementing expandably the Karatsuba multiplication algorithm and he achieves *sub-quadratic growth for the computation time*. This shows up very clearly with numbers having thousands of digits, up to the maximum which a.t.t.o.w is at 8192 digits.

The l3bigint multiplication from late 2015 is observed to be roughly 3x--4x faster than the one from \xintiiexpr in the range of 4000 to 5000 digits integers, and isn't far from being 9x faster at 8000 digits. On the other hand \xintiiexpr's multiplication is found to be on average roughly 2.5x faster than l3bigint's for numbers up to 100 digits and the two packages achieve about the

---

[17] For example multiplication of integers having from 50 to 100 digits takes roughly of the order of the millisecond on a 2012 desktop computer. I compared this to using Python3: using timeit module on a wrapper defined as return w*z with random integers of 100 digits, I observe on the same computer a computation time of roughly 4.10^{-7}s per call. And with return str(w* z) then this becomes more like 16.10^{-7}s per call. And with return str(int(W)*int(Z)) where W and Z are strings, this becomes about 26.10^{-7}s (I am deliberately ignoring Python's Decimal module here...) Anyway, my sentence from earlier version of this documentation: *this is, I guess, at least about 1000 times slower than what can be expected with any reasonable programming language,* is about right. I then added: *nevertheless as compilation of a typical LaTeX document already takes of the order of seconds and even dozens of seconds for long ones, this leaves room for reasonably many computations via xintexpr or via direct use of the macros of xint/xintfrac.* [18] One can currently use package bnumexpr to associate the bigintcalc macros with an expression parser. This may be unavailable in future if bnumexpr becomes more tightly associated with future evolutions or variants of xintcore. [19] The latter exist as work-in-progress for some time in the source code. [20] 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.) [21] at the time of writing (2014/10/28) the l3fp (exactly represented) floating point numbers have their exponents limited to ±9999.

same speed at 900 digits: but each such multiplication of numbers of 900 digits costs about one or two tenths of a second on a 2012 desktop computer, whereas the order of magnitude is rather the ms for numbers with 50--100 digits.[22]

Even with the superior l3bigint Karatsuba multiplication it takes about 3.5s on this 2012 desktop computer for a single multiplication of two 5000-digits numbers. Hence it is not possible to do routinely such computations in a document. I have long been thinking that without the expandability constraint much higher speeds could be achieved, but perhaps I have not given enough thought to sustain that optimistic stance.[23]

I remain of the opinion that if one really wants to do computations with *thousands* of digits, one should drop the expandability requirement. Indeed, as clearly demonstrated long ago by the pi computing file by D. Roegel one can program TeX to compute with many digits at a much higher speed than what xint achieves: but, direct access to memory storage in one form or another seems a necessity for this kind of speed and one has to renounce at the complete expandability.[24]

5.2. Floating point evaluations

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

$$\xintDigits:=P\relax$$

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

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

\xintfloatexpr[Q]...\relax also admits an optional argument [Q] but it has an altogether different meaning: the computations are always done with the prevailing \xinttheDigits precision and the optional argument Q is used for the final rounding. This makes sense only if Q<\xinttheDigits and is intended to clean up the result from dubious last digits (when Q<0 it indicates rather by how many digits one should reduce the mantissa lengths via a final rounding).

The IEEE 754[25] requirement of *correct rounding* for addition, subtraction, multiplication, division and square root is achieved (in arbitrary precision) by the macros of xintfrac hence also by the infix operators +, -, *, /.

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

Due to a typographical oversight, this documentation (up to 1.2j) adjoined ^ and ** to the above list of infix operators. But as is explained in subsection 9.86, what is guaranteed regarding integer powers is an error of at most 0.52ulp, not the correct rounding. Half-integer powers are computed as square roots of integer powers.

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

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

---

[22] I have tested this again on 2016/12/19, but the macros have not changed on the l3bigint side and barely on the xintcore side, hence I got again the same results... [23] The apnum package implements (non-expandably) arbitrary precision fixed point algebra and (v1.6) functions exp, log, sqrt, the trigonometrical direct and inverse functions. [24] The LuaTeX project possibly makes endeavours such as xint appear even more insane that they are, in truth: xint is able to handle fast enough computations involving numbers with less than one hundred digits and brings this to all engines. [25] The IEEE 754-1985 standard was for hardware implementations of binary floating-point arithmetic with a specific value for the precision (24 bits for single precision, 53 bits for double precision). The newer IEEE 754-2008 (https://en.wikipedia.org/wiki/IEEE_floating_point) normalizes five basic formats, three binaries and two decimals (16 and 34 decimal digits) and discusses extended formats with higher precision. These standards are only indirectly relevant to libraries like xint dealing with arbitrary precision.

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

The power function in the expression parsers accepts integer exponents and also half-integer exponents for float expressions.[26] A preliminary implementation of fractional powers is available see xintlog. Trigonometrical functions are available (xinttrig).

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

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

The more ambitious model would be for the computing macros to obey the intrinsic precision of their inputs, i.e. to compute the correct rounding to P digits of the exact mathematical result corresponding to inputs allowed to have their own higher precision.[27] This would be feasible by xintfrac which after all knows how to compute exactly, but I have for the time being decided that for reasons of efficiency, the chosen model is the one of rounding inputs to the target precision first.

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

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

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

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

5.3. Expansion matters

5.3.1. Full expansion of the first token

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

By convention in this manual *f*-expansion (``full expansion'' or ``full first expansion'') will be this TEX process of expanding repeatedly the first token seen. For those familiar with LATEX3

---

[26] Half-integer exponents work inside expressions, but not via the \xintFloatPower macro.   [27] The MPFR library http://www.mpfr.org/ implements this but it does not know fractions!

(which is not used by xint) this is what is called in its documentation full expansion (whereas expansion inside \edef would be described I think as ``exhaustive'' expansion).

Most of the package macros, and all those dealing with computations[28], are expandable in the strong sense that they expand to their final result via this *f-expan*sion. This will be signaled in their descriptions via a star in the margin.

★

These macros not only have this property of *f-expand*ability, they all begin by first applying *f-expan*sion to their arguments. Again from LaTeX3's conventions this will be signaled by a margin annotation next to the description of the arguments.

f

5.3.2. Summary of important expandability aspects

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

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

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

   To the contrary \xinttheiiexpr and others have no issues with things such as \xinttheiiexpr ⟩ \x+\x\y\relax.

2. using \if...\fi constructs *inside* the package macro arguments requires suitably mastering TeXniques (\expandafter's and/or swapping techniques) to ensure that the *f-expan*sion will indeed absorb the \else or closing \fi, else some error will arise in further processing. Therefore it is highly recommended to use the package provided conditionals such as \xintifEq, \xintifGt, \xintifSgn,... or, for LaTeX users and when dealing with short integers the etool-box[29] expandable conditionals (for small integers only) such as \ifnumequal, \ifnumgreater, ... Use of *non-expandable* things such as \ifthenelse is impossible inside the arguments of xint macros.

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

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

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

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

4. With the definition

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

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

   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}}}

---

[28] except \xintXTrunc.   [29] https://ctan.org/pkg/etoolbox

or use the *lowercase* form of \xintAdd:

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

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

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

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

6. In the expression parsers of xintexpr such as \xintexpr..\relax, \xintfloatexpr..\relax the contents are expanded completely from left to right until the ending \relax is found and swallowed, and spaces and even (to some extent) catcodes do not matter.

7. For all variants, prefixing with \xintthe allows to print the result or use it in other contexts. Shortcuts \xinttheexpr, \xintthefloatexpr, \xinttheiiexpr, ... are available.

5.4. Input formats for macros

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

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

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

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

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

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

    At 1.2o many macros from xintcore/xint which used. All these macros have now been removed at 1.3.

$\overset{\text{Frac}}{f}$ 4. the fraction input format applies to the arguments of xintfrac macros handling genuine fractions. It allows two types of inputs: general and restricted. The restricted type is parsed faster, but... is restricted.

    general: inputs of the shape A.BeC/D.EeF. Example:
    ```
    \noindent\xintRaw{+--0367.8920280e17/-++278.289287e-15}\newline
    ```

62

```
\xintRaw{+--+1253.2782e++--3/---0087.123e---5}\par
```
-3678920280/278289287[31]
-12532782/87123[7]

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

- everything is optional, absent numbers are treated as zero, here are some extreme cases:

  ```
  \xintRaw{}, \xintRaw{.}, \xintRaw{./1.e}, \xintRaw{-.e}, \xintRaw{e/-1}
  ```
  0/1[0], 0/1[0], 0/1[0], 0/1[0], 0/1[0]

- `AB` and `DE` may start with pluses and minuses, then leading zeroes, then digits.

- `C` and `F` will be given to `\numexpr` and can be anything recognized as such and not provoking arithmetic overflow (the lengths of `B` and `E` will also intervene to build the final exponent naturally which must obey the TₑX bound).

- the `/`, `.` (numerator and/or denominator) and `e` (numerator and/or denominator) are all optional components.

- each of `A`, `B`, `C`, `D`, `E` and `F` may arise from *f*-expansion of a macro.

- the whole thing may arise from *f*-expansion, however the `/`, `.`, and `e` should all come from this initial expansion. The `e` of scientific notation is mandatorily lowercased.

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

<div style="margin-left:2em">Frac
f</div>

Notice that `*`, `+` and `-` contrarily to the `/` (which is treated simply as a kind of delimiter) are not acceptable within arguments of this type (see subsection 5.6 for some exceptions to this.)

Generally speaking, there should be no spaces among the digits in the inputs (in arguments to the package macros). Although most would be harmless in most macros, there are some cases where spaces could break havoc.[30] So the best is to avoid them entirely.

This is entirely otherwise inside an `\xintexpr`-ession, where spaces are ignored (except when they occur inside arguments to some macros, thus escaping the `\xintexpr` parser). See the section 2.

There are also some slighly more obscure expansion types: in particular, the `\xintApplyInline` and `\xintFor*` macros from xinttools apply a special iterated *f*-expansion, which gobbles spaces, to the non-braced items (braced items are submitted to no expansion because the opening brace stops *f it) coming from their list argument; this is denoted by a special symbol in the margin. Some other macros such as `\xintSum` from xintfrac first do an *f*-expansion, then treat each found (braced or Frac
f→ * f not) item (skipping spaces between such items) via the general fraction input parsing, this is signaled as here in the margin where the signification of the `*` is thus a bit different from the previous case.

n, resp. *o* 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.

---

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

5.5. Output formats of macros

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

- arithmetic macros from `xintcore`/`xint` deliver integers in the strict format as described in the previous section.
- arithmetic macros from `xintfrac` produce on output the strict fraction format `A/B[N]`, which stands for `(A/B)×10^N`, where `A` and `B` are integers, `B` is positive, and `N` is a ``short'' integer. The output is not reduced to smallest terms. The `A` and `B` may end with zeroes (*i.e*, `N` does not represent all powers of ten). The denominator `B` is always strictly positive. There is no `+` sign. The `-` is always first if present (i.e. the denominator on output is always positive.) The output will be expressed as such a fraction even if the inputs are both integers and the mathematical result is an integer. The `B=1` is not removed.[31]
- macros with `Float` in their names produce on output scientific format with `P=\xinttheDigits` digits, a lowercase `e` and an exponent `N`. The first digit is not zero, it is preceded by an optional minus sign and is followed by a dot and `P-1` digits. Trailing zeroes are not trimmed. There is one exceptional case:
  - if the value is mathematically zero, it is output as `0.e0`, i.e. zeros after the decimal mark are removed and the exponent is always `0`.

Future versions of the package may modify this.

5.6. Count registers and variables

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

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

The macros allowing the extended format for long numbers or dealing with fractions will *to some extent* allow the direct use of count registers and even infix algebra inside their arguments: a count register `\mycountA` or `\count 255` is admissible as numerator or also as denominator, with no need to be prefixed by `\the` or `\number`. It is possible to have as argument an algebraic expression as would be acceptable by a `\numexpr...\relax`, under this condition: *each of the numerator and denominator is expressed with at most* nine *tokens*.[32] [33] Important: a slash for rounded division in a `\numexpr` should be written with braces `{/}` to not be confused with the `xintfrac` delimiter between numerator and denominator (braces will be removed internally and the slash will count for one token). Example: `\mycountA+\mycountB{/}17/1+\mycountA*\mycountB`, or `\count 0+\count 2{/}17/1+\count 0*\count 2`.

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

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

---

[31] refer to the documentation of `\xintPRaw` for an alternative.   [32] The `1.2k` and earlier versions manual claimed up to 8 tokens, but low-level TeX error arose if the `\numexpr ...\relax` occupied exactly 8 tokens *and* evaluated to zero. With `1.2l` and later, up to 9 tokens are always safe and one may even drop the ending `\relax`. But well, all these explanations are somewhat silly because prefixing by `\the` or `\number` is always working with arbitrarily many tokens.   [33] Attention! in the LaTeX context a `\value{countername}` will behave ok only if it is first in the input, if not it will not get expanded, and braces around the name will be removed and chaos will ensue inside a `\numexpr`. One should enclose the whole input in `\the\numexpr...\relax` in such cases.

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

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

5.7. Dimension registers and variables

⟨*dimen*⟩ variables can be converted into (short) integers suitable for the xint macros by prefixing them with `\number`. This transforms a dimension into an explicit short integer which is its value in terms of the sp unit (1/65536 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 macro defined by \newlength with `\number` will thus discard the plus and minus glue components and return the dimension component as described above, and usable in the xint bundle macros.

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

One may thus compute areas or volumes with no limitations, in units of 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.

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

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

TeX dimensions

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

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

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

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

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

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

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

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

1157 dd=\number\dimexpr 1157dd\relax/65536=1238.000000000000...pt
1357 dd=\number\dimexpr 1357dd\relax/65536=1452.001724243164...pt

We thus discover that TEX (or rather here, e-TEX, but one can check that this works the same in TEX82), uses 1238/1157 as a conversion factor (and necessarily intermediate computations simulate higher precision than a priori available with integers less than 2^{31} or rather 2^{30} for dimensions). Hence the 1452/1357 ratio is irrelevant, an artefact of the rounding (or rather, as we see, truncating) for one dd to be expressed as an integral number of sp's.

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

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

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

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

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

The difference of circa 5Å is arguably tiny!

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

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

and the centered convergents of this fraction are 1/1, 15/14, 61/57, 107/100, 1238/1157, 11249/10513, 23736/22183, 296081/276709, 615898/575601, 11382245/10637527, 22148592/20699453, 188570981/176233151, 543564351/508000000. We do recover the 1238/1157 therein!

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

5.8. \ifcase, \ifnum, ... constructs

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

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

1. ERROR
2. OK

3. OK

In order to use successfully `\if...\fi` constructions either as arguments to the xint bundle expandable macros, or when building up a completely expandable macro of one's own, one needs some TEXnical expertise (see also item 2 on page 61).

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

If these tests are to be applied to standard TEX short integers, it is more efficient to use (under LATEX) the equivalent conditional tests from the etoolbox[34] package.

5.9. No variable declarations are needed

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

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

```
\def\x{1729728} \def\y{352827927} \edef\z{\xintiiMul {\x}{\y}}
\meaning\z
```
macro:->610296344513856

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

```
\def\x{1729728} \def\y{352827927} \oodef\w {\xintiiMul\x\y} \fdef\z{\xintiiMul {\x}{\y}}
\meaning\w, \meaning\z
```
macro:->610296344513856, macro:->610296344513856

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

xintexpr does provide an interface to declare and assign values to identifiers which can then be used in expressions: subsection 2.12.

5.10. Possible syntax errors to avoid

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

- using - to prefix some macro: `-\xintiiSqr{35}/271`.[35]

- using one pair of braces too many `\xintIrr{{\xintiiPow {3}{13}}/243}` (the computation goes through with no error signaled, but the result is completely wrong).

- things like `\xintiiAdd { \x}{\y}` as the space will cause \x to be expanded later, most probably within a `\numexpr` thus provoking possibly an arithmetic overflow.

- using [] and decimal points at the same time `1.5/3.5[2]`, or with a sign in the denominator `3/-5[7]`. The scientific notation has no such restriction, the two inputs `1.5/-3.5e-2` and `-1.`
`5e2/3.5` are equivalent: `\xintRaw{1.5/-3.5e-2}=-15/35[2]`, `\xintRaw{-1.5e2/3.5}=-15/35[2]`.

- generally speaking, using in a context expecting an integer (possibly restricted to the TEX bound) a macro or expression which returns a fraction: `\xinttheexpr 4/2\relax` outputs `4/2`, not 2. Use `\xintNum {\xinttheexpr 4/2\relax}` or `\xinttheiexpr 4/2\relax` (which rounds the result to the nearest integer, here, the result is already an integer) or `\xinttheiiexpr 4/2`
`\relax`. Or, divide in your head 4 by 2 and insert the result directly in the TEX source.

---

[34] https://ctan.org/pkg/etoolbox   [35] to the contrary, this *is* allowed inside an `\xintexpr`-ession.

5.11. Error messages

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

Here is an example interactive run:

```
! Undefined control sequence.
<argument> \xint/
                DivisionByZero (hit <RET> thrice)
l.11 \xintiiDivision{123}{0}


?
! Undefined control sequence.
<argument> \xint/
                Division of 123 by 0
l.11 \xintiiDivision{123}{0}


?
! Undefined control sequence.
<argument> \xint/
                next: {0}{0}
l.11 \xintiiDivision{123}{0}


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

This is an experimental feature, which is in preparation for next major release.[36] [37]

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

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

as the \numexpr will swallow the \relax whose presence is mandatory for \xintexpr, errors will inevitably arise and may lead to very cryptic messages; but nothing unusual or especially traumatizing for the daring experienced TeX/LaTeX user, whose has seen zillions of un-helpful error messages already in her daily practice of TeX/LaTeX.[38]

5.12. Package namespace, catcodes

The bundle packages needs that the \space and \empty control sequences are pre-defined with the identical meanings as in Plain TeX (or LaTeX2e which has the same macros).

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

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

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

---

[36] The related macros checking or resetting error flags are implemented in embryonic form but no user interface is provided with 1.2 .21 release.   [37] The implementation is cloned from LaTeX3.   [38] not to mention the LaTeX error messages used by Emacs AUCTeX mode also for Plain TeX runs...

68

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

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

> For advanced TeX users. At loading time of the packages the catcode configuration may be arbitrary as long as it satisfies the following requirements: the percent is of category code comment character, the backslash is of category code escape character, digits have category code other and letters have category code letter. Nothing else is assumed.

5.13. Origins of the package

2013/03/28. Package bigintcalc by Heiko Oberdiek already provides expandable arithmetic operations on ``big integers'', exceeding the TeX limits (of 2^{31} – 1), so why another[39] one?

I got started on this in early March 2013, via a thread on the c.t.tex usenet group, where Ulrich Diez used the previously cited package together with a macro (\ReverseOrder) which I had contributed to another thread.[40] 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 precomputed digit arithmetic. I wondered if some speed could be gained by using \numexpr to do four digits at a time for elementary multiplications (as the maximal admissible number for \numexpr has ten digits).

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

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

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

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

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

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

---

[39] this section was written before the xintfrac package; the author is not aware of another package allowing expandable computations with arbitrarily big fractions.   [40] the \ReverseOrder could be avoided in that circumstance, but it does play a crucial rôle here.

6. Macros of the xintkernel package

The xintkernel package contains mainly the common code base for handling the load-order of the bundle packages, the management of catcodes at loading time, definition of common constants and macro utilities which are used throughout the code etc ... it is automatically loaded by all packages of the bundle.

It provides a few macros possibly useful in other contexts.

6.1. \odef, \oodef, \fdef

\oodef\controlsequence {<stuff>} does

```
\expandafter\expandafter\expandafter\def
\expandafter\expandafter\expandafter\controlsequence
\expandafter\expandafter\expandafter{<stuff>}
```

This works only for a single \controlsequence, with no parameter text, even without parameters. An alternative would be:

```
\def\oodef #1#{\def\oodefparametertext{#1}%
              \expandafter\expandafter\expandafter\expandafter
              \expandafter\expandafter\expandafter\def
              \expandafter\expandafter\expandafter\oodefparametertext
              \expandafter\expandafter\expandafter }
```

but it does not allow \global as prefix, and, besides, would have anyhow its use (almost) limited to parameter texts without macro parameter tokens (except if the expanded thing does not see them, or is designed to deal with them).

There is a similar macro \odef with only one expansion of the replacement text <stuff>, and \fdef which expands fully <stuff> using \romannumeral-`0.

They can be prefixed with \global. It appears than \fdef is generally a bit faster than \odef when expanding macros from the xint bundle, when the result has a few dozens of digits. \oodef needs thousands of digits it seems to become competitive.

6.2. \xintReverseOrder

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

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

xinttools provides a variant \xintRevWithBraces which keeps brace pairs in the output, and *f*-expands its input first.

For inputs consisting only digit tokens, see \xintReverseDigits from xint.

6.3. \xintLength

n ★ \xintLength{⟨*list*⟩} counts how many tokens (or braced items) there are (possibly none). It does no expansion of its argument, so to use it to count things in the replacement text of a macro \x one should do \expandafter\xintLength\expandafter{\x}. Blanks between items are not counted. See also \xintNthElt{0} (from xinttools) which first *f*-expands its argument and then applies the same code.

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

6.4. `\xintFirstItem`

n ⋆

New with
1.4

`\xintFirstItem{⟨list⟩}` returns the first item of its argument, one pair of braces removed. If the list has no items the output is empty.

It does no expansion. For this and the next similar ones, see sourcexint.pdf for comments on limitations.

6.5. `\xintLastItem`

Added at 1.2i.

n ⋆

`\xintLastItem{⟨list⟩}` returns the last item of its argument, one pair of braces removed. If the list has no items the output is empty.

It does no expansion, which should be obtained via suitable `\expandafter`'s. See also `\xint-NthElt{-1}` from xinttools which obtains the same result (but with another code) after having however *f-expan*ded its argument first.

6.6. `\xintFirstOne`

n ⋆

New with
1.4

`\xintFirstOne{⟨list⟩}` returns the first item as a braced item. I.e. if it was braced the braces are kept, else the braces are added. It looks like using `\xintFirstItem` within braces, but the difference is when the input was empty. Then the output is empty.

It does no expansion, which should be obtained via suitable `\expandafter`'s.

6.7. `\xintLastOne`

n ⋆

New with
1.4

`\xintLastOne{⟨list⟩}` returns the last item as a braced item. I.e. if it was braced the braces are kept, else the braces are added. It looks like using `\xintLastItem` within braces, but the difference is when the input was empty. Then the output is empty.

It does no expansion, which should be obtained via suitable `\expandafter`'s.

6.8. `\xintReplicate`, `\xintreplicate`

$\overset{\text{num}}{x}$ *n* ⋆

New with
1.4

`\romannumeral\xintreplicate{x}{⟨stuff⟩}` is simply copied over from LaTeX3's `\prg_replicate:nn` with some minor changes.[41]

And `\xintReplicate{x}` integrates the `\romannumeral` prefix.

It does not do any expansion of its second argument but inserts it in the upcoming token stream precisely x times. Using it with a negative x raises no error and does nothing.[42]

6.9. `\xintGobble`, `\xintgobble`

$\overset{\text{num}}{x}$ ⋆

New with
1.4

`\romannumeral\xintgobble{x}` is a Gobbling macro written in the spirit of LaTeX3's `\prg_replicate:nn` (which I cloned as `\xintreplicate`.) It gobbles x tokens upstream, with x allowed to be as large as 531440. Don't use it with x<0.

And `\xintGobble{x}` integrates the `\romannumeral`.

`\xintgobble` looks as if it must be related to `\xintTrim` from xinttools, but the latter uses different code (using directly `\xintgobble` is not possible because one must make sure not to gobble more than the number of available items; and counting available items first is an overhead which

---

[41] I started with the code from Joseph WRIGHT available on an online site.   [42] This behaviour may change in future.

71

\xintTrim avoids.) It is rather\xintKeep with a negative first argument which hands over to \xint-gobble (because in that case it is needed to count anyhow beforehand the number of items, hence \xintgobble can then be used safely.)

I wrote an \xintcount in the same spirit as \xintreplicate and \xintgobble. But it needs to be counting hundreds of tokens to be worth its salt compared to \xintLength.

6.10. (WIP) \xintUniformDeviate

^{num}
x ★ \xintUniformDeviate{x} is a wrapper of engine \pdfuniformdeviate (or \uniformdeviate).[43] The implementation is to be considered experimental for the time being.

The argument is expanded in \numexpr and the macro itself needs two expansion steps. It produces like the engine primitive an integer (digit tokens) with minimal value 0 and maximal one x-1 if x is positive, or minimal value x+1 and maximal value 0 if x is negative. For the discussion next, x is supposed positive as this avoids having to insert absolute values in formulas.

The underlying engine Random Number Generator works with an array of 55 28bits integers. To produce a « uniform » random integer in a given range 0..x-1 it produces next pseudo-random y (supposedly uniformly distributed, i.e. non-uniformity can be neglected) such that $0 \leq y < 2^{28}$ and the output is the rounding of $x * (y/2^{28})$, with upper bound x remapped to 0. This has following corollaries:

1. with x=2^{29} or x=2^{30} the engine primitive produces only even numbers,

2. with x=3*2^{26} the integers produced by the RNG when taken modulo three obey the proportion 1:1:2, not 1:1:1,

3. with x=3*2^{14} there is analogous although weaker non-uniformity of the random integers when taken modulo 3,

4. generally speaking pure powers of two should generate uniform random integers, but when the range is divisible by large powers of two, the non-uniformity may be amplified in surprising ways by modulo operations.

These observations are not to be construed as criticism of the engine primitive itself, which comes from MetaPost, as the code comments and more generally the whole of *The Art of Computer Programming, Vol. 2* stresses that it should rather be seen as producing random fractions (the unit fraction being 2^{28}). Using it as a generator for *integers* is a bit of an abuse.

The first goal of \xintUniformDeviate is to guarantee a better uniformity for the distribution of random integers in any given range x.

If the probability to obtain a given y in 0..x-1 is (1+e(y))/x, the ``relative non-uniformity'' for that value y is |e(y)|.

The engine primitive guarantees only $x/2^{28}$ relative non-uniformity, and \xintUniformDeviate (in its current implementation) improves this by a factor 2^{28}=268435456: the non-uniformity is guaranteed to be bounded by $x/2^{56}$.[44] With such a small non-uniformity, modulo phenomena as mentioned earlier are not observable in reasonable computing time.

```
%\xintdefiifunc mod3(x):= x 'mod' 3;
\xintNewIIExpr\ModThree[1]{#1 'mod' 3}

\pdfsetrandomseed 87654321
\xintdefiivar BadDigits:=qraw(%
   \romannumeral\xintreplicate{504}{{\ModThree{\pdfuniformdeviate "C000000}}}%
```

---

[43] The \uniformdeviate primitive has been added to XeTeX and will be available with TeXLive 2019 release. [44] These estimates assume that the engine RNG underlying stream of 28-bits integers can be considered uniform; it is known that the parity bits of these 28-bits integers have a period of 55(2^{55}-1) and that after that many draws the count of 1s has only an excess of 55 compared to the count of 0s, so the scale seems to be an intrinsic non-uniformity of 2^{-55} but it is not obvious if it applies to much shorter ranges. At any rate we assumed that the non-uniformity for x a power of two less than 2^{28} is negligible in comparison to 2^{-28}. Bigger powers of 2 produce only even integers because the output is rescaled by factor x/2^{28}!

```
);%

\pdfsetrandomseed 87654321
\xintdefiivar GoodDigits:=qraw(%
    \romannumeral\xintreplicate{504}{{\ModThree{\xintUniformDeviate{"C000000}}}}%
);%

These 504 digits generated from \string\pdfuniformdeviate:
\xinttheiiexpr BadDigits\relax\hfill\break
contain these respective amounts of 0, 1, and 2:
% (this is definitely not the fastest way to count, but it is fun - and expandable)
\xinttheiiexpr iter(0,0,0;(i=0)?[[@][0]+1,[@][1],[@][2]]
                            {(i=1)?[[@][0],[@][1]+1,[@][2]]
                                    {[@][0],[@][1],[@][2]+1}},
                    i=BadDigits)\relax\par

These 504 digits generated from \string\xintUniformDeviate:
\xinttheiiexpr GoodDigits\relax\hfill\break
contain these respective amounts of 0, 1, and 2:
\xinttheiiexpr iter(0,0,0;(i=0)?[[@][0]+1,[@][1],[@][2]]
                            {(i=1)?[[@][0],[@][1]+1,[@][2]]
                                    {[@][0],[@][1],[@][2]+1}},
                    i=GoodDigits)\relax\par
% % output to data file for double-check with python
% \newwrite\out
% \immediate\openout\out=\jobname.data
% \immediate\write\out{Lbad=[\xinttheiiexpr BadDigits\relax]}
% \immediate\write\out{Lgood=[\xinttheiiexpr GoodDigits\relax]}
% \immediate\closeout\out
```

These 504 digits generated from \pdfuniformdeviate: 1, 2, 1, 2, 1, 0, 0, 1, 2, 1, 1, 0, 1, 0, 0,
2, 2, 0, 0, 2, 2, 2, 1, 2, 2, 2, 2, 2, 0, 0, 2, 1, 1, 0, 0, 1, 1, 1, 2, 2, 2, 2, 0, 1, 1, 2, 1, 1, 1, 2,
1, 2, 2, 1, 0, 0, 2, 0, 0, 2, 0, 2, 1, 1, 2, 2, 0, 0, 2, 2, 2, 2, 2, 0, 2, 1, 2, 2, 1, 1, 0, 1, 2, 2, 0,
1, 0, 2, 2, 2, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 2, 2, 2, 1, 2, 2, 0, 1, 0, 2, 1, 2, 2, 1, 1, 2, 2, 1, 2, 2,
1, 2, 2, 1, 2, 2, 2, 1, 0, 2, 2, 2, 1, 1, 2, 1, 2, 2, 1, 2, 1, 0, 2, 2, 2, 2, 1, 2, 0, 2, 2, 2, 0, 0, 2,
0, 1, 0, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 2, 2, 1, 0, 2, 1, 2, 0, 1, 0, 1, 0, 2, 2, 0, 2, 2, 2, 2, 2, 0, 2,
2, 2, 2, 2, 1, 2, 2, 1, 1, 1, 1, 2, 0, 2, 1, 0, 1, 2, 1, 1, 1, 2, 1, 1, 2, 0, 0, 2, 1, 1, 2, 2, 0, 1, 2,
2, 1, 2, 0, 1, 1, 2, 2, 1, 2, 2, 1, 2, 1, 1, 2, 1, 2, 2, 1, 1, 0, 1, 1, 2, 2, 0, 1, 1, 2, 2, 2, 0, 1, 0,
2, 1, 1, 1, 0, 1, 0, 2, 1, 1, 2, 2, 1, 2, 0, 2, 2, 1, 0, 1, 2, 2, 0, 2, 0, 2, 2, 2, 2, 0, 2, 0, 2, 2, 2,
2, 2, 1, 1, 2, 1, 2, 1, 1, 0, 2, 2, 2, 1, 2, 0, 0, 2, 2, 0, 1, 0, 2, 2, 2, 2, 2, 1, 2, 0, 1, 0, 0, 2, 0,
2, 2, 1, 0, 1, 0, 1, 1, 2, 0, 1, 0, 0, 2, 1, 0, 2, 2, 0, 1, 2, 1, 1, 0, 0, 0, 0, 0, 0, 2, 2, 2, 0, 0, 2,
2, 2, 0, 2, 2, 1, 1, 2, 1, 1, 2, 0, 1, 2, 1, 2, 1, 1, 2, 2, 0, 2, 2, 2, 1, 2, 2, 1, 0, 1, 2, 2, 2, 2, 2,
0, 2, 2, 1, 0, 2, 0, 1, 2, 2, 2, 2, 1, 0, 1, 2, 0, 2, 2, 2, 2, 2, 2, 2, 0, 1, 0, 2, 1, 2, 1, 0, 2, 1, 1,
2, 2, 0, 1, 0, 0, 0, 2, 2, 1, 2, 2, 1, 1, 1, 0, 1, 2, 2, 2, 0, 2, 1, 0, 1, 2, 2, 1, 2, 0, 2, 0, 1, 0, 1,
0, 0, 2, 0, 2, 1, 2, 0, 1, 0, 0, 1, 1, 2, 0, 2, 0, 0, 2, 0, 2, 0, 0, 0, 0, 2, 1, 0, 2, 2, 0, 2, 1, 2
contain these respective amounts of 0, 1, and 2: 124, 147, 233
These 504 digits generated from \xintUniformDeviate: 2, 1, 0, 1, 0, 2, 1, 1, 2, 2, 2, 1, 2, 2, 0,
1, 0, 2, 2, 0, 1, 2, 0, 2, 0, 0, 2, 2, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 1, 0, 2, 1, 1, 0, 0, 2, 2, 1, 0, 0,
1, 1, 0, 2, 0, 2, 0, 0, 1, 0, 2, 0, 0, 1, 1, 0, 0, 1, 2, 0, 0, 2, 2, 0, 1, 1, 0, 2, 1, 2, 0, 1, 0, 0, 1,
2, 1, 1, 0, 1, 0, 2, 2, 1, 0, 1, 1, 2, 1, 0, 2, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0,
0, 1, 2, 2, 2, 1, 1, 1, 0, 2, 1, 0, 2, 2, 2, 2, 0, 0, 1, 2, 2, 2, 2, 2, 0, 2, 2, 1, 0, 1, 2, 0, 0, 2, 0,
0, 2, 1, 1, 0, 1, 1, 1, 1, 2, 1, 1, 1, 2, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 2, 0, 2, 1, 1, 0, 0, 2, 1, 1, 1,
0, 2, 2, 0, 1, 1, 1, 2, 0, 1, 0, 2, 1, 2, 1, 0, 2, 0, 1, 1, 2, 2, 0, 0, 2, 1, 0, 2, 0, 2, 2, 2, 1, 0, 0,
1, 2, 2, 2, 1, 1, 0, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 2, 2, 0, 0, 0, 1, 0, 0, 0, 1,
0, 2, 1, 0, 2, 2, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 2, 1, 1, 0, 0, 0, 0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 2, 1,
1, 0, 2, 0, 1, 1, 1, 1, 2, 1, 1, 2, 2, 1, 2, 2, 1, 1, 0, 1, 2, 2, 2, 0, 1, 1, 2, 0, 2, 1, 0, 0, 2, 1, 0,

1, 1, 2, 1, 1, 0, 1, 0, 2, 0, 1, 1, 1, 0, 2, 2, 0, 1, 2, 2, 0, 1, 0, 2, 2, 0, 1, 1, 2, 2, 0, 2, 1, 0, 1,
0, 1, 2, 0, 1, 1, 2, 1, 2, 2, 0, 2, 0, 0, 2, 0, 1, 2, 1, 2, 2, 0, 0, 0, 2, 2, 0, 2, 2, 1, 2, 2, 2, 0, 1,
0, 0, 0, 2, 2, 2, 0, 0, 2, 1, 0, 0, 1, 2, 2, 2, 2, 0, 2, 1, 0, 0, 2, 1, 0, 1, 2, 1, 1, 2, 1, 0, 0, 0, 2,
1, 0, 1, 0, 1, 1, 2, 2, 0, 2, 1, 1, 2, 0, 0, 1, 0, 2, 0, 1, 0, 2, 0, 1, 2, 1, 0, 2, 0, 1, 1, 2, 0, 2, 2,
2, 1, 1, 2, 2, 2, 1, 2, 2, 1, 2, 2, 0, 0, 0, 2, 0, 2, 2, 1, 2, 2, 1, 1, 2, 1, 1, 0, 0, 0, 0, 0, 0
contain these respective amounts of 0, 1, and 2: 161, 174, 169

There is a second peculiarity of the engine RNG: two seeds sharing the same low k bits generate sequences of 28-bits integers which are identical modulo 2^k! In particular after setting the seed, there are only 2 distinct sequences of parity bits for the integers generated by \pdfunifor
mdeviate (2 to the power 28)...

In order to mitigate, \xintUniformDeviate currently only uses the seven high bits from the underlying random stream, using multiple calls to \pdfuniformdeviate 128. From the Birthday Effect, after about 2^{11} seeds one will likely pick a new one sharing its 22 low bits with an earlier one.

1. but as the final random integer is obtained by additional operations involving the range x (currently a modulo operation), for odd ranges it is more difficult for bit correlations to be seen,

2. anyway as they are only 2^{28} seeds in total, after only 2^{14} seeds it is likely to encounter one already explored, and then random integers are identical, however complicated the RNG's raw output is malaxed, and whatever the target range x. And 2^{14} is only eight times as large as 2^{11}.

It would be nice if the engine provided some user interface for letting its RNG execute a given number of iterations without the overhead of replicated executions of \pdfuniformdeviate. This could help gain entropy and would reduce correlations across series from distinct seeds.

The description above summarizes parts of discussions held with Bruno Le Floch in May 2018 on occasion of his LaTeX3 contributions related to this.

TEXhackers note: currently the implementation of \xintUniformDeviate consumes exactly 5 calls to the engine primitive at each execution; the improved x/2^{56} non-uniformity could be obtained with only 2 calls, but paranoïa about the phenonemon of seeds with common bits has led me to accept the overhead of using the 7 high bits of 4 random 28bits integers, rather than one single 28bits integer, or two, or three.

Timings indicate that one \xintUniformDeviate has a time cost about 13 times the one for one call to the engine primitive (and not only 5, as the extra arithmetic expressions add overhead which is more costly than the primitive itself). Except if the code using the pseudo-random number is very short, this time penalty will prove in practice much less severe (and this is one important reason why we opted for obtaining 28bits via the 7 high bits of 4 successive pseudo random numbers from the engine primitive). For example let's raise 100 times a random integer to the tenth power: [45]

```
\pdfsetrandomseed 12345678
\pdfresettimer\romannumeral\xintreplicate
        {100}{\fdef\foo{\xintiiPow{\xintUniformDeviate{100000000}}{10}}}%
\the\dimexpr\pdfelapsedtime sp\relax\space (with \string\xintUniformDeviate)\newline
(last result: \foo)\newline
\pdfsetrandomseed 12345678
\pdfresettimer\romannumeral\xintreplicate
        {100}{\fdef\foo{\xintiiPow{\pdfuniformdeviate 100000000}{10}}}%
\the\dimexpr\pdfelapsedtime sp\relax\space (with \string\pdfuniformdeviate)\newline
(last result: \foo)\par
```

0.04396pt (with \xintUniformDeviate)
(last result: 3613350655737796214900713110162983015171362952221525157584613965918429029376)
0.04375pt (with \pdfuniformdeviate)
(last result: 6625445910227856739067071963066183205761325943682291308182364687926259765625)

---

[45] This is done on a 2.4GHz processor. Hmm... or on a 2.8GHz one, I should add some automatic recognition to the build process...

> **T_EXhackers note:** the macros `\xintRandomDigits` or `\xintiiRandRange`, and their variants, as well as the supporting macros for `random`() generate random decimal digits eight by eight as if using `\xintUniformDeviate{100000000}`, but via a direct optimized call made possibly by the range being a power of 10.

7. Macros of the xintcore package

Package `xintcore` is automatically loaded by `xint`.

xintcore provides for big integers the four basic arithmetic operations (addition, subtraction, multiplication, division), as well as powers and factorials.

In the descriptions of the macros `{N}` and `{M}` stand for (big) integers or macros *f-expanding* to such big integers in strict format as described in subsection 5.4.

All macros require strict integer format on input and produce strict integer format on output, except:

☞ {

- `\xintiNum` which converts to strict integer format an input in *extended* integer format, i.e. admitting multiple leading plus or minus signs, then possibly leading zeroes, then digits,
- and `\xintNum` which is an alias for the former, which gets redefined by `xintfrac` to accept more generally also decimal numbers or fractions as input and which truncates them to integers.

The `ii` in the names of the macros such as `\xintiiAdd` serves to stress that they accept only strict integers as input (this is signaled by the margin annotation *f*), or macros *f-expanding* to such strict format (big) integers and that they produce strict integers as output.

Other macros, such as `\xintDouble`, lack the `ii`, but this is only a legacy of the history of the package and they have the same requirements for input and format of output as the `ii`-macros.

The letter x (with margin annotation $\overset{num}{x}$) stands for an argument which will be handled embedded in `\numexpr..\relax`. It will thus be completely expanded and must give an integer obeying the TeX bounds. See also subsection 5.6. This is the case for the argument of `\xintiiFac` or the exponent argument of `\xintiiPow`.

The ⋆'s in the margin are there to remind of the complete expandability, even *f-expand*ability of the macros, as discussed in subsubsection 5.3.1.

7.1. \xintiNum

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

```
\xintiNum{+---++----+--000000000367941789479}
```
-367941789479

7.2. \xintDouble

f ⋆ `\xintDouble{N}` computes `2N`.

7.3. \xintHalf

f ⋆ `\xintHalf{N}` computes `N/2` truncated towards zero.

7.4. \xintInc

f ⋆   \xintInc{N} evaluates N+1.

7.5. \xintDec

f ⋆   \xintDec{N} evaluates N-1.

7.6. \xintDSL

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

7.7. \xintDSR

f ⋆   \xintDSR{N} is truncated decimal shift right, *i.e.* it is the truncation of N/10 towards zero.

7.8. \xintDSRr

f ⋆   \xintDSRr{N} is rounded decimal shift right, *i.e.* it is the rounding of N/10 away from zero. It is needed in xintcore for use by \xintiiDivRound.

7.9. \xintFDg

f ⋆   \xintFDg{N} outputs the first digit (most significant) of the number.

7.10. \xintLDg

f ⋆   \xintLDg{N} outputs the least significant digit. When the number is positive, this is the same as the remainder in the Euclidean division by ten.

7.11. \xintiiSgn

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

7.12. \xintiiOpp

f ⋆   \xintiiOpp{N} outputs the opposite -N of the number N.
   Important note: an input such as -\foo is not legal, generally speaking, as argument to the macros of the xint bundle (except, naturally in \xintexpr-essions). The reason is that the minus sign stops the *f*-expansion done during parsing of the inputs. One must use the syntax \xintiiOpp{\foo} if one wants to pass -\foo as argument to other macros.

7.13. \xintiiAbs

f ⋆   \xintiiAbs{N} outputs the absolute value of the number.

7.14. \xintiiAdd

ff ⋆   \xintiiAdd{N}{M} computes the sum of the two (big) integers.

7.15. \xintiiCmp

ff ⋆   \xintiiCmp{N}{M} produces 1 if N>M, 0 if N=M, and -1 if N<M.
   At 1.2l this macro was moved from package xint to xintcore.

77

7.16. \xintiiSub

ff ★ \xintiiSub{N}{M} computes the difference N-M.

7.17. \xintiiMul

ff ★ \xintiiMul{N}{M} computes the product of two (big) integers.

7.18. \xintiiSqr

f ★ \xintiiSqr{N} produces the square.

7.19. \xintiiPow

f $\overset{\text{num}}{x}$ ★ \xintiiPow{N}{x} computes N^x. For x=0, this is 1. For N=0 and x<0, or if |N|>1 and x<0, an error is raised. There will also be an error if x exceeds the maximal ε-TeX number 2147483647, but the real limit for exponents comes from either the computation time or the settings of some TeX memory parameters.

> Indeed, the maximal power of 2 which xint is able to compute explicitely is 2^(2^17)=2^131072 which has 39457 digits. This exceeds the maximal size on input for the xintcore multiplication, hence any 2^N with a higher N will fail. On the other hand 2^(2^16) has 19729 digits, thus it can be squared once to obtain 2^(2^17) or multiplied by anything smaller, thus all exponents up to and including 2^17 are allowed (because the power operation works by squaring things and making products).

7.20. \xintiiFac

$\overset{\text{num}}{x}$ ★ \xintiiFac{x} computes the factorial.

> The (theoretically) allowable range is $0 \leqslant x \leqslant 10000$.
> However the maximal possible computation depends on the values of some memory parameters of the tex executable: with the current default settings of TeXLive 2015, the maximal computable factorial (a.t.t.o.w. 2015/10/06) turns out to be 5971! which has 19956 digits.

The factorial function, or equivalently ! as post-fix operator is available in \xintiiexpr, \xintexpr:
  \printnumber{\xinttheiiexpr 200!\relax}\par
78865786736479050355523632139321850622951359776871732632947425332443594499634033429203042840 1⟩
198462390417721213891963883025764279024263710506192662495282993111346285727076331723739698892 ⟩
439224456214516642402540332918641312274282948532775242424075739032403212574055795686602260312 ⟩
904170324062351700858796178922222789623703897374720000000000000000000000000000000000000000000 ⟩
0000000
  See also \xintFloatFac from package xintfrac for the float variant, used in \xintfloatexpr.

7.21. \xintiiDivision

ff ★ \xintiiDivision{M}{N} produces {quotient}{remainder}, in the sense of (mathematical) Euclidean division: M = QN + R, $0 \leq R < |N|$. So the remainder is always non-negative and the formula M = QN + R always holds independently of the signs of N or M. Division by zero is an error (even if M vanishes) and returns {0}{0}.

7.22. \xintiiQuo

ff ⋆ \xintiiQuo{M}{N} computes the quotient from the Euclidean division.

7.23. \xintiiRem

ff ⋆ \xintiiRem{M}{N} computes the remainder from the Euclidean division.

7.24. \xintiiDivRound

ff ⋆ \xintiiDivRound{M}{N} returns the rounded value of the algebraic quotient M/N of two big integers. The rounding is ``away from zero.''

```
\xintiiDivRound {100}{3}, \xintiiDivRound {101}{3}
```

33, 34

7.25. \xintiiDivTrunc

ff ⋆ \xintiiDivTrunc{M}{N} computes trunc(M/N). For positive arguments M, N > 0 it is the same as the Euclidean quotient \xintiiQuo.

```
\xintiiQuo{1000}{57} (Euclidean), \xintiiDivTrunc{1000}{57} (truncated),
\xintiiDivRound{1000}{57} (rounded)\newline
\xintiiQuo{-1000}{57}, \xintiiDivTrunc{-1000}{57} (t), \xintiiDivRound{-1000}{57} (r)\newline
\xintiiQuo{1000}{-57}, \xintiiDivTrunc{1000}{-57} (t), \xintiiDivRound{1000}{-57} (r)\newline
\xintiiQuo{-1000}{-57}, \xintiiDivTrunc{-1000}{-57} (t), \xintiiDivRound{-1000}{-57} (r)\par
```

17 (Euclidean), 17 (truncated), 18 (rounded)
-18, -17 (t), -18 (r)
-17, -17 (t), -18 (r)
18, 17 (t), 18 (r)

7.26. \xintiiDivFloor

ff ⋆ \xintiiDivFloor{M}{N} computes floor(M/N). For positive divisor N > 0 and arbitrary dividend M it is the same as the Euclidean quotient \xintiiQuo.

```
\xintiiQuo{1000}{57} (Euclidean), \xintiiDivFloor{1000}{57} (floored)\newline
\xintiiQuo{-1000}{57}, \xintiiDivFloor{-1000}{57}\newline
\xintiiQuo{1000}{-57}, \xintiiDivFloor{1000}{-57}\newline
\xintiiQuo{-1000}{-57}, \xintiiDivFloor{-1000}{-57}\par
```

17 (Euclidean), 17 (floored)
-18, -18
-17, -18
18, 17

7.27. \xintiiMod

ff ⋆ \xintiiMod{M}{N} computes M − N ∗ floor(M/N). For positive divisor N > 0 and arbitrary dividend M it is the same as the Euclidean remainder \xintiiRem.

Formerly, this macro computed M − N ∗ trunc(M/N). The former meaning is retained as \xintiiMod-Trunc.

```
\xintiiRem {1000}{57} (Euclidean), \xintiiMod {1000}{57} (floored),
\xintiiModTrunc {1000}{57} (truncated)\newline
\xintiiRem {-1000}{57}, \xintiiMod {-1000}{57}, \xintiiModTrunc {-1000}{57}\newline
\xintiiRem {1000}{-57}, \xintiiMod {1000}{-57}, \xintiiModTrunc {1000}{-57}\newline
\xintiiRem {-1000}{-57}, \xintiiMod {-1000}{-57}, \xintiiModTrunc {-1000}{-57}\par
```

31 (Euclidean), 31 (floored), 31 (truncated)
26, 26, -31

79

```
31, -26, 31
26, -31, -31
```

7.28. \xintNum

f ★ \xintNum is originally an alias for \xintiNum. But with xintfrac loaded its meaning is modified to accept more general inputs. It then becomes an alias to \xintTTrunc which truncates the general input to an integer in strict format.

8. Macros of the xint package

This package loads automatically xintcore (and xintkernel) hence all macros described in section 7 are still available.

This is 1.4b of 2020/02/25.

Version 1.0 was released 2013/03/28. Since 1.1 2014/10/28 the core arithmetic macros have been moved to a separate package xintcore, which is automatically loaded by xint. Only the \xintiiSum, \xintiiPrd, \xintiiSquareRoot, \xintiiPFactorial, \xintiiBinomial genuinely add to the arithmetic macros from xintcore. (\xintiiFac which computes factorials is already in xintcore.)

With the exception of \xintLen, of the «Boolean logic macros» (see next paragraphs) all macros require inputs being integers in strict format, see subsection 5.4.[46] The ii in the macro names is here as a reminder of that fact. The output is an integer in strict format, or a pair of two braced such integers for \xintiiSquareRoot, with the exception of \xintiiE which may produce strings of zero's if its first argument is zero.

Macros \xintDecSplit and \xintReverseDigits are non-arithmetic and have their own specific rules.

For all macros described here for which it makes sense, package xintfrac defines a similar one without ii in its name. This will handle more general inputs: decimal, scientific numbers, fractions. The ii macros provided here by xint can be nested inside macros of xintfrac but the opposite does not apply, because the output format of the xintfrac macros, even for representing integers, is not understood by the ii macros. The «Boolean macros» \xintAND etc... are exceptions though, they work fine if served as inputs some xintfrac output, despite doing only *f*-expansion. Prior to 1.2o, these macros did apply the \xintNum or the more general xintfrac general parsing, but this overhead was deemed superfluous as it serves only to handle hand-written input and is not needed if the input is obtained as a nested chain of xintfrac macros for example.

Prior to release 1.2o, xint defined additional macros which applied \xintNum to their input arguments. All these macros were deprecated at 1.2o and have been removed at 1.3.

At 1.3d macros \xintiiGCD and \xintiiLCM from package xintgcd are also available from loading xint only. They are support macros for the (multi-arguments) functions gcd() and lcm() in \xintiiexpr.

See subsubsection 5.3.1 for the significance of the $\overset{\text{Num}}{f}$, *f* , $\overset{\text{num}}{x}$ and ⋆ margin annotations.

---

[46] of course for conditionals such as \xintiiifCmp this constraint applies only to the first two arguments.

8.1. **\xintiLen**

^{Num}
f ★   \xintiLen{N} returns the length of the number, after its parsing via \xintiNum. The count does not include the sign.

    \xintiLen{-12345678901234567890123456789}

29

Prior to 1.2o, the package defined only \xintLen, which is extended by xintfrac to fractions or decimal numbers, hence acquires a bit more overhead then.

8.2. **\xintReverseDigits**

3.60004pt, 8.39996pt, 12.0pt

f ★   \xintReverseDigits{N} will reverse the order of the digits of the number. \xintRev is the former denomination and is kept as an alias. Leading zeroes resulting from the operation are not removed. Contrarily to \xintReverseOrder this macro *f*-expands its argument; it is only usable with digit tokens. It does *not* apply \xintNum to its argument (so this must be done explicitly if the argument is an integer produced from some xintfrac macros). It does accept a leading minus sign which will be left upfront in the output.

    \oodef\x{\xintReverseDigits
       {9876543210987654321098765432109876543210}}\meaning\x\par
    \noindent\oodef\x{\xintReverseDigits {\xintReverseDigits
       {9876543210987654321098765432109876543210}}}\meaning\x\par

macro:->0123456789012345678901234567890123456789
macro:->9876543210987654321098765432109876543210

8.3. **\xintDecSplit**

^{num}
x f ★   \xintDecSplit{x}{N} cuts the N (a list of digits) into two pieces L and R: it outputs {L}{R} where the original N is the concatenation LR. These two pieces are decided according to x:

- for x>0, R coincides with the x least significant digits. If x equals or exceeds the length of N the first piece L will thus be *empty*,
- for x=0, R is empty, and L is all of N,
- for x<0, the first piece L consists of the |x| most significant digits and the second piece R gets the remaining ones. If x equals or exceeds the length of N the second piece R will thus be *empty*.

This macro provides public interface to some functionality which is primarily of internal interest. It operates only (after *f*-expansion) on ``strings'' of digits tokens: leading zeroes are allowed but a leading sign (even a minus sign) will provoke an error.

Breaking change with 1.2i: formerly N<0 was replaced by its absolute value. Now, a sign (positive or negative) will create an error.

8.4. \xintDecSplitL, \xintDecSplitR

^{num}*x* *f* ★ \xintDecSplitL{x}{N} returns the first piece (unbraced) from the \xintDecSplit output.
^{num}*x* *f* ★ \xintDecSplitR{x}{N} returns the second piece (unbraced) from the \xintDecSplit output.

8.5. \xintiiE

f^{num}*x* ★ \xintiiE{N}{x} serves to extend N with x zeroes. The parameter x must be non-negative. The same output would be obtained via \xintDSH{-x}{N}, except for N=0, as \xintDSH{-x}{N} multiplies N by 10^x hence produces 0 if N=0 whereas \xintiiE{0}{x} produces x+1 zeros.

    \xintiiE {0}{91}\par
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

8.6. \xintDSH

^{num}*x* *f* ★ \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 \xintDSR 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.

8.7. \xintDSHr, \xintDSx

^{num}*x* *f* ★ \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 \xintiiDivision),

- 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.

^{num}*x* *f* ★ \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.

8.8. \xintiiEq

f *f* ★ \xintiiEq{N}{M} returns 1 if N=M, 0 otherwise.

8.9. \xintiiNotEq

f *f* ★ \xintiiNotEq{N}{M} returns 0 if N=M, 1 otherwise.

8.10. \xintiiGeq

f *f* ★ \xintiiGeq{N}{M} returns 1 if the *absolute value* of the first number is at least equal to the absolute value of the second number. If |N|<|M| it returns 0.
   Important: the macro compares *absolute values*.

8.11. \xintiiGt

f *f* ★ \xintiiGt{N}{M} returns 1 if N>M, 0 otherwise.

8.12. \xintiiLt

ff ⋆ \xintiiLt{N}{M} returns 1 if N<M, 0 otherwise.

8.13. \xintiiGtorEq

ff ⋆ \xintiiGtorEq{N}{M} returns 1 if N⩾M, 0 otherwise. Extended by xintfrac to fractions.

8.14. \xintiiLtorEq

ff ⋆ \xintiiLtorEq{N}{M} returns 1 if N⩽M, 0 otherwise.

8.15. \xintiiIsZero

f ⋆ \xintiiIsZero{N} returns 1 if N=0, 0 otherwise.

8.16. \xintiiIsNotZero

f ⋆ \xintiiIsNotZero{N} returns 1 if N!=0, 0 otherwise.

8.17. \xintiiIsOne

f ⋆ \xintiiIsOne{N} returns 1 if N=1, 0 otherwise.

8.18. \xintiiOdd

f ⋆ \xintiiOdd{N} is 1 if the number is odd and 0 otherwise.

8.19. \xintiiEven

f ⋆ \xintiiEven{N} is 1 if the number is even and 0 otherwise.

8.20. \xintiiMON

f ⋆ \xintiiMON{N} computes (-1)^N.
   \xintiiMON {-280914019374101929}
-1

8.21. \xintiiMMON

f ⋆ \xintiiMMON{N} computes (-1)^{N-1}.
   \xintiiMMON {280914019374101929}
1

8.22. \xintiiifSgn

fnnn ⋆ \xintiiifSgn{⟨N⟩}{⟨A⟩}{⟨B⟩}{⟨C⟩} executes either the ⟨A⟩, ⟨B⟩ or ⟨C⟩ code, depending on its first argument being respectively negative, zero, or positive.

8.23. \xintiiifZero

fnn ⋆ \xintiiifZero{⟨N⟩}{⟨IsZero⟩}{⟨IsNotZero⟩} expandably checks if the first mandatory argument N (a number, possibly a fraction if xintfrac is loaded, or a macro expanding to one such) is zero or not. It then either executes the first or the second branch.
   Beware that both branches must be present.

8.24. \xintiiifNotZero

f n n ★ \xintiiifNotZero{⟨N⟩}{⟨IsNotZero⟩}{⟨IsZero⟩} expandably checks if the first mandatory argument N is not zero or is zero. It then either executes the first or the second branch.
  Beware that both branches must be present.

8.25. \xintiiifOne

f n n ★ \xintiiifOne{⟨N⟩}{⟨IsOne⟩}{⟨IsNotOne⟩} expandably checks if the first mandatory argument N is one or not one. It then either executes the first or the second branch. Beware that both branches must be present.

8.26. \xintiiifCmp

f f n n n ★ \xintiiifCmp{⟨A⟩}{⟨B⟩}{⟨A<B⟩}{⟨A=B⟩}{⟨A>B⟩} compares its first two arguments and chooses accordingly the correct branch.

8.27. \xintiiifEq

f f n n ★ \xintiiifEq{⟨A⟩}{⟨B⟩}{⟨A=B⟩}{⟨not(A=B)⟩} checks equality of its two first arguments and executes the corresponding branch.

8.28. \xintiiifGt

f f n n ★ \xintiiifGt{⟨A⟩}{⟨B⟩}{⟨A>B⟩}{⟨not(A>B)⟩} checks if A > B and executes the corresponding branch.

8.29. \xintiiifLt

f f n n ★ \xintiiifLt{⟨A⟩}{⟨B⟩}{⟨A<B⟩}{⟨not(A<B)⟩} checks if A < B and executes the corresponding branch.

8.30. \xintiiifOdd

f n n ★ \xintiiifOdd{⟨A⟩}{⟨A odd⟩}{⟨A even⟩} checks if A is and odd integer and executes the corresponding branch.

8.31. \xintiiSum

∗f ★ \xintiiSum{⟨braced things⟩} after expanding its argument expects to find a sequence of tokens (or braced material). Each is *f*-expanded, and the sum of all these numbers is returned.

```
\xintiiSum{{123}{-98763450}{\xintiiFac{7}}{\xintiiMul{3347}{591}}}\newline
\xintiiSum{1234567890}\newline
\xintiiSum{1234}\newline
\xintiiSum{}
```

-96780210
45
10
0

  A sum with only one term returns that number: \xintiiSum {{-1234}}=-1234. Attention that \xintiiSum {-1234} is not legal input and would make the TeX run fail.

8.32. \xintiiPrd

⋆f ⋆ \xintiiPrd{⟨*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.

```
\xintiiPrd{{-9876}{\xintiiFac{7}}{\xintiiMul{3347}{591}}}\newline
\xintiiPrd{123456789123456789}\newline
\xintiiPrd {1234}\newline
\xintiiPrd{}
```

-98458861798080
131681894400
24
1

Attention that \xintiiPrd {-1234} is not legal input and would make the TeX compilation fail.

```
$2^{200}3^{100}7^{100}=\printnumber
      {\xintiiPrd {{\xintiiPow {2}{200}}{\xintiiPow {3}{100}}{\xintiiPow {7}{100}}}}$
```

$2^{200}3^{100}7^{100} = 26787279316615775757566279517007548402324740266374015348974459614815426412965499$ ↩
$49000044400724076572713000016531207640654562118014357199401590334353924402821243896682224892$ ↩
7862988084382716133376

With xintexpr, the syntax is the natural one:

```
$2^{200}3^{100}7^{100}=\printnumber{\xinttheiiexpr 2^200 * 3^100 * 7^100\relax}$
```

$2^{200}3^{100}7^{100} = 26787279316615775757566279517007548402324740266374015348974459614815426412965499$ ↩
$49000044400724076572713000016531207640654562118014357199401590334353924402821243896682224892$ ↩
7862988084382716133376

8.33. \xintiiSquareRoot

f ⋆ \xintiiSquareRoot{N} returns two braced integers {M}{d} which satisfy d>0 and M^2-d=N with M the smallest (hence if N=k^2 is a perfect square then M=k+1, d=2k+1).

```
\xintAssign\xintiiSquareRoot {17000000000000000000000000000}\to\A\B
\xintiiSub{\xintiiSqr\A}\B=\A\string^2-\B
```

17000000000000000000000000000=4123105625618^2-2799177881924

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

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

8.34. \xintiiSqrt, \xintiiSqrtR

f ⋆
f ⋆ \xintiiSqrt{N} computes the largest integer whose square is at most equal to N. \xintiiSqrtR produces the rounded, not truncated, square root.

```
\begin{itemize}[nosep]
\item \xintiiSqrt  {3000000000000000000000000000000000000000}
\item \xintiiSqrtR {3000000000000000000000000000000000000000}
\item \xintiiSqrt  {\xintiiE {3}{100}}
\end{itemize}
```

- 1732050807568877293
- 1732050807568877294
- 17320508075688772935274463415058723669428052538138

8.35. \xintiiBinomial

num num
X X ⋆ \xintiiBinomial{x}{y} computes binomial coefficients.

If x<0 an out-of-range error is raised. Else, if y<0 or if x<y the macro evaluates to 0.

The allowable range is $0 \leqslant x \leqslant 99999999$. But this theoretical range includes binomial coefficients with more than the roughly 19950 digits that the arithmetics of xint can handle. In such cases, the computation will end up in a low-level TeX error after a long time.

It turns out that $\binom{65000}{32500}$ has 19565 digits and $\binom{64000}{32000}$ has 19264 digits. The latter can be evaluated (this takes a long long time) but presumably not the former (I didn't try). Reasonable feasible evaluations are with binomial coefficients not exceeding about one thousand digits.

The binomial function is available in the xintexpr parsers.

```
\xinttheiiexpr seq(binomial(100,i), i=47..53)\relax
```
84413487283064039501507937600, 93206558875049876949581681100, 98913082887808032681188722800, 100891344545564193334812497256, 98913082887808032681188722800, 93206558875049876949581681100, 84413487283064039501507937600

See \xintFloatBinomial from package xintfrac for the float variant, used in \xintfloatexpr.

In order to evaluate binomial coefficients $\binom{x}{y}$ with $x > 99999999$, or even $x \geqslant 2^{31}$, but y is not too large, one may use an ad hoc function definition such as:

```
\xintdeffunc mybigbinomial(x,y):=`*`(x-y+1..[1]..x)//y!;%
%                           without [1], x would have been limited to < 2^31
\printnumber{\xinttheexpr mybigbinomial(98765432109876543210,10)\relax}
```
24338098741940755592729533173058146177070669479669793038510211146784065843698581878582323710273605753727154823896333598784607399737267865769250677841005879712614223266522709755926675174871960261

To get this functionality in macro form, one can do:

```
\xintNewIIExpr\MyBigBinomial [2]{`*`(#1-#2+1..[1]..#1)//#2!}
\printnumber{\MyBigBinomial {98765432109876543210}{10}}
```
24338098741940755592729533173058146177070669479669793038510211146784065843698581878582323710273605753727154823896333598784607399737267865769250677841005879712614223266522709755926675174871960261

As we used \xintNewIIExpr, this macro will only accept strict integers. Had we used \xintNewExpr the \MyBigBinomial would have accepted general fractions or decimal numbers, and computed the product at the numerator without truncating them to integers; but the factorial at the denominator would truncate its argument.

8.36. \xintiiPFactorial

$\overset{num}{x}\overset{num}{x}$ ★ \xintiiPFactorial{a}{b} computes the partial factorial $(a+1)(a+2)...b$. For a=b the product is considered empty hence returns 1.

The allowed range is $-100000000 \leqslant a, b \leqslant 99999999$. The rule is to interpret the formula as the product of the j's such that $a < j \leqslant b$, hence in particular if $a \geqslant b$ the product is empty and the macro evaluates to 1.

Only for $0 \leqslant a \leqslant b$ is the behaviour to be considered stable. For $a > b$ or negative arguments, the definitive rules have not yet been fixed.

```
\xintiiPFactorial {100}{130}
```
692930218852038710122984228458228032875919700607893504000000000000

This theoretical range allows computations whose result values would have more than the roughly 19950 digits that the arithmetics of xint can handle. In such cases, the computation will end up in a low-level TeX error after a long time.

The pfactorial function is available in the xintexpr parsers.

```
\xinttheiiexpr pfactorial(100,130)\relax
```
692930218852038710122984228458228032875919700607893504000000000000

See \xintFloatPFactorial from package xintfrac for the float variant, used in \xintfloatexpr.

In case values are needed with $b > 99999999$, or even $b \geqslant 2^{31}$, but b - a is not too large, one may use an ad hoc function definition such as:

```
\xintdeffunc mybigpfac(a,b):=`*`(a+1..[1]..b);%
%                           without [1], b would have been limited to < 2^31
\printnumber{\xinttheexpr mybigpfac(98765432100,98765432120)\relax}
```

78000855017567528067298107313023778438653002029049647467208196028116499434050587656870489322 99630604482236853566403912561449912587404607844104078121472675461815442734098676283450069933 3229486005730169970340095665766440000

8.37. \xintiiMax

f f ⋆ \xintiiMax{N}{M} returns the largest of the two in the sense of the order structure on the relative integers (*i.e.* the right-most number if they are put on a line with positive numbers on the right): \xintiiMax {-5}{-6}=-5.

8.38. \xintiiMin

f f ⋆ \xintiiMin{N}{M} returns the smallest of the two in the sense of the order structure on the relative integers (*i.e.* the left-most number if they are put on a line with positive numbers on the right): \xintiiMin {-5}{-6}=-6.

8.39. \xintiiMaxof

f → *f* ⋆ \xintiiMaxof{{a}{b}{c}...} returns the maximum. The list argument may be a macro, it is *f*-expanded first.

8.40. \xintiiMinof

f → *f* ⋆ \xintiiMinof{{a}{b}{c}...} returns the minimum. The list argument may be a macro, it is *f*-expanded first.

8.41. \xintifTrueAelseB

f n n ⋆ \xintifTrueAelseB{⟨*f*⟩}{⟨*true branch*⟩}{⟨*false branch*⟩} is a synonym for \xintiiifNotZero. \xintiiifnotzero is lowercase companion macro.
    Note 1: as it does only *f*-expansion on its argument it fails with inputs such as --0. But with xintfrac loaded, it does work fine if nested with other xintfrac macros, because the output format of such macros is fine as input to \xintiiifNotZero. This remark applies to all other «Boolean logic» macros next.
    Note 2: prior to 1.2o this macro was using \xintifNotZero which applies \xintNum to its argument (or gets redefined by xintfrac to handle general decimal numbers or fractions). Hence it would have worked with input such as --0. But it was decided at 1.2o that the overhead was not worth it. The same remark applies to the other «Boolean logic» type macros next.

8.42. \xintifFalseAelseB

f n n ⋆ \xintifFalseAelseB{⟨*f*⟩}{⟨*false branch*⟩}{⟨*true branch*⟩} is a synonym for \xintiiifZero. \xintiiifzero is lowercase companion macro.

8.43. \xintNOT

f ⋆ \xintNOT is a synonym for \xintiiIsZero. \xintiiiszero serves as lowercase companion macro.

8.44. \xintAND

f f ⋆ \xintAND{f}{g} returns 1 if f!=0 and g!=0 and 0 otherwise.

8.45. `\xintOR`

f f ⋆   `\xintOR{f}{g}` returns 1 if f!=0 or g!=0 and 0 otherwise.

8.46. `\xintXOR`

f f ⋆   `\xintXOR{f}{g}` returns 1 if exactly one of f or g is true (i.e. non-zero), else 0.

8.47. `\xintANDof`

f → ∗*f* ⋆   `\xintANDof{{a}{b}{c}...}` returns 1 if all are true (i.e. non zero) and 0 otherwise. The list argument may be a macro, it (or rather its first token) is *f*-expanded first to deliver its items.

8.48. `\xintORof`

f → ∗*f* ⋆   `\xintORof{{a}{b}{c}...}` returns 1 if at least one is true (i.e. does not vanish), else it produces 0. The list argument may be a macro, it is *f*-expanded first.

8.49. `\xintXORof`

f → ∗*f* ⋆   `\xintXORof{{a}{b}{c}...}` returns 1 if an odd number of them are true (i.e. do not vanish), else it produces 0. The list argument may be a macro, it is *f*-expanded first.

8.50. `\xintiiGCD`

f f ⋆   `\xintiiGCD{N}{M}` computes the greatest common divisor. It is positive, except when both N and M vanish, in which case the macro returns zero.
   `\xintiiGCD{10000}{1113}`=1
   `\xintiiGCD{123456789012345}{9876543210321}`=3
   At 1.3d, this macro (which is used by the gcd() function in `\xintiiexpr`) was copied over to xint, thus removing a partial dependency of xintexpr on xintgcd.
   At 1.4 xintgcd requires xint and the latter is thus the one providing the macro.

8.51. `\xintiiLCM`

f f ⋆   `\xintiiLCM{N}{M}` computes the least common multiple. It is positive, except if one of N or M vanish, in which case the macro returns zero.
   `\xintiiLCM{10000}{1113}`=11130000
   `\xintiiLCM{123456789012345}{9876543210321}`=406442103762636081733470915
   At 1.3d, this macro (which is used by the lcm() function in `\xintiiexpr`) was copied over to xint, thus removing a partial dependency of xintexpr on xintgcd.
   At 1.4 xintgcd requires xint and the latter is thus the one providing the macro.

8.52. `\xintiiGCDof`

f → ∗*f* ⋆   `\xintiiGCDof{{a}{b}{c}...}` computes the greatest common divisor of the integers a, b, ... It is a support macro for the gcd() function of the `\xintiiexpr` parser.
   It replaces the `\xintGCDof` which was formerly provided by xintgcd and is now available via xintfrac in a version handling also fractions.

8.53. \xintiiLCMof

f → *f ★ \xintiiLCMof{{a}{b}{c}...} computes the least common multiple of the integers a, b, ... It is a support macro for the lcm() function of the \xintiiexpr parser.

It replaces the \xintLCMof which was formerly provided by xintgcd and is now available via xintfrac in a version handling also fractions.

8.54. \xintLen

Num
f ★ \xintLen is originally an alias for \xintiiLen. But with xintfrac loaded its meaning is modified to accept more general inputs.

8.55. (WIP) \xintRandomDigits

> All randomness related macros are Work-In-Progress: implementation and user interface may change. They work only if the TeX engine provides the \uniformdeviate or \pdfuniformdeviate primitive. See \xintUniformDeviate for additional information.

num
x ★ \xintRandomDigits{N} expands in two steps to N random decimal digits. The argument must be non-negative and is limited by TeX memory parameters. On TeXLive 2018 with input save stack size at 5000 the maximal allowed N is at most 19984 (tested within a \write to an auxiliary file, the macro context may cause a reduced maximum).

```
\pdfsetrandomseed 271828182
\xintRandomDigits{92}
```
6003378238914615120727799353934428057809087191963874539873557768643616576939495863937 6355806

TeXhackers note: the digits are produced eight by eight by the same method which would result from \xintUniformDeviate{100000000} but with less overhead.

8.56. (WIP) \xintXRandomDigits

num
x ☆ \xintXRandomDigits{N} expands under exhaustive expansion (\edef, \write, \csname ...) to N random decimal digits. The argument must be non-negative. For example:

```
\newwrite\out
\immediate\openout\out=\jobname-out.txt
\immediate\write\out{\xintXRandomDigits{4500000}}
\immediate\closeout\out
```
creates a 4500001 bytes file (it ends with a line feed character). Trying with 5000000 raises this error:

```
Runaway text?
5888759471685115827645141350702175553544798052404394077534513542232830\ETC.
! TeX capacity exceeded, sorry [main memory size=5000000].
<inserted text> 666515098

l.15 ...ate\write\out{\xintXRandomDigits{5000000}}

No pages of output.
Transcript written on temp.log.
```
This can be lifted by increasing the TeX memory settings (installation dependent).

TeXhackers note: the digits are produced eight by eight by the same method which would result from \xintUniformDeviate{100000000} but with less overhead.

8.57. (WIP) \xintiiRandRange

f ★ \xintiiRandRange{A} expands to a random (big) integer N such that 0<=N<A. It is a supporting macro for randrange(). As with Python's function of the same name, it is an error if A<=0.

```
\pdfsetrandomseed 271828314
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\newline
\xintiiRandRange{\xintNum{1e40}}\newline
\pdfsetrandomseed 271828314
\xinttheiiexpr randrange(num(1e40))\relax\newline % bare 1e40 not understood by \xintiiexpr
\pdfsetrandomseed 271828314
\xinttheexpr randrange(1e40)\relax
```

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1408107837990425263001878034077495278697
1408107837990425263001878034077495278697
1408107837990425263001878034077495278697

Of course, keeping in mind that the set of seeds is of cardinality 2^{28}, randomness is a bit illusory here say with A=10^N, N>8, if we proceed immediately after having set the seed. If we add some entropy in any way, then it is slightly more credible; but I think that for each seed the period is something like $2^{27}(2^{55}-1)55$,[47] so we expect at most about $2^{110}55$ ``points in time'', and this is already small compared to the 10^40 from example above. Thus already we are very far from being intrinsically able to generate all numbers with fourty digits as random numbers, and this makes the previous section about usage of \xintXRandomDigits to generate millions of digits a bit comical...

TₑXhackers note: the digits are produced eight by eight by the same method which would result from \xintUniformDeviate{100000000} but with less overhead.

8.58. (WIP) \xintiiRandRangeAtoB

ff ★ \xintiiRandRangeAtoB{A}{B} expands to a random (big) integer N such that A<=N<B. It is a supporting macro for randrange(). As with Python's function of the same name, it is an error if B<=A.

```
\pdfsetrandomseed 271828314
12345678911111111111111111111\newline
\xintiiRandRangeAtoB{12345678911111111111111111111}{12345678922222222222222222222}\newline
\pdfsetrandomseed 271828314
\def\test{%
\xinttheiiexpr randrange(12345678911111111111111111111,12345678922222222222222222222)\relax}%
\romannumeral\xintreplicate{10}{\test\newline}%
12345678922222222222222222222
```

12345678911111111111111111111
12345678916037426188606389808
12345678916037426188606389808
12345678916060337223949101536
12345678912190033095886250034
12345678917323740152668511995
12345678915424847208552293485
12345678921595726610650510660
12345678911673261982088192858
12345678911339325803675947159
12345678917791540296982027151
12345678913602899909728811895
12345678922222222222222222222

---

[47] Compare the result of exercise 3.2.2-30 in TAOCP, vol II.

TEXhackers note: the digits are produced eight by eight by the same method which would result from `\xintUniformDeviate{100000000}` but with less overhead.

9. Macros of the xintfrac package

First version of this package was in release 1.03 (2013/04/14) of the xint bundle.

At release 1.3 (2018/02/28) the behaviour of \xintAdd (and of \xintSub) was modified: when adding a/b and c/d they will use always the least common multiple of the denominators. This helps limit the build-up of denominators, but the author still hesitates if the fraction should be reduced to smallest terms. The current method allows (for example when multiplying two polynomials) to keep a well-predictable denominator among various terms, even though some may be reducible.

xintfrac loads automatically xintcore and xint and inherits their macro definitions. Only these two are redefined: \xintNum and \xintLen. As explained in subsection 5.4 and subsection 5.5 the interchange format for the xintfrac macros, i.e. A/B[N], is not understood by the ii-named macros of xintcore/xint which expect the so-called strict integer format. Hence, to use such an ii-macro with an output from an xintfrac macro, an extra \xintNum wrapper is required. But macros already defined by xintfrac cover most use cases hence this should be a rarely needed.

$\begin{smallmatrix}\text{Frac}\\ f\end{smallmatrix}$ In the macro descriptions, the variable f and the margin indicator stand for the xintfrac input format for integers, scientific numbers, and fractions as described in subsection 5.4.

$\begin{smallmatrix}\text{num}\\ x\end{smallmatrix}$ As in the xint.sty documentation, x stands for something which internally will be handled in a \numexpr. It may thus be an expression as understood by \numexpr but its evaluation and intermediate steps must obey the TeX bound.

The output format for most macros is the A/B[N] format but naturally the float macros use the scientific notation on output. And some macros are special, for example \xintTrunc produces decimal numbers, \xintIrr produces an A/B with no [N], \xintiTrunc and \xintiRound produce integers without trailing [N] either, etc...

1.3a belatedly adds documentation for some macros such as \xintDivFloor which had been defined long ago, but did not make it to the user manual for various reasons, one being that it is thought few users will use directly the xintfrac macros, the \xintexpr interface being more convenient. For complete documentation refer to sourcexint.pdf.

9.1. \xintNum

Frac
f ★ The original \xintNum from xint is made a synonym to \xintTTrunc (whose description is to be found farther in this section).

Attention that for example \xintNum{1e100000} expands to the needed 100001 digits...

The original \xintNum from xintcore which does not understand the fraction slash or the scientific notation is still available under the name \xintiNum.

9.2. \xintRaw

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

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

No simplification is done, not even of common zeroes between numerator and denominator:

    \xintRaw {178000/25600000}
178000/25600000[0]

9.3. \xintNumerator

Frac
f ★ The input data is parsed as if by \xintRaw into A/B[N] format and the macro outputs A if N<=0, or A extended by N zeroes if N>0.

    \xintNumerator {178000/25600000[17]}\newline
    \xintNumerator {312.289001/20198.27}\newline
    \xintNumerator {178000e-3/256e5}\newline
    \xintNumerator {178.000/25600000}
17800000000000000000000000
312289001

94

178000
178000

9.4. \xintDenominator

Frac
f ★ The input data is parsed as if by \xintRaw into `A/B[N]` format and the macro outputs `B` if `N>0`, or `B` extended by `|N|` zeroes if `N<=0`.

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

25600000
20198270000
25600000000
25600000000

9.5. \xintRawWithZeros

Frac
f ★ This macro parses the input and outputs `A/B`, with `A` as would be returned by \xintNumerator{f} and `B` as would be returned by \xintDenominator{f}.

```
\xintRawWithZeros{178000/25600000[17]}\newline
\xintRawWithZeros{312.289001/20198.27}\newline
\xintRawWithZeros{178000e-3/256e5}\newline
\xintRawWithZeros{178.000/25600000}\newline
\xintRawWithZeros{\the\numexpr 571*987\relax.123e-10/\the\numexpr-201+59\relax e-7}
```

17800000000000000000000/25600000
312289001/20198270000
178000/25600000000
178000/25600000000
-563577123/142000000

9.6. \xintREZ

Frac
f ★ The input is first parsed into `A/B[N]` as by \xintRaw, then trailing zeroes of `A` and `B` are suppressed and `N` is accordingly adjusted.

```
\xintREZ {178000/25600000[17]}
```

178/256[15]

This macro is used internally by various other constructs; its implementation was redone entirely at `1.3a`, and it got faster on long inputs.

9.7. \xintIrr

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

```
\xintIrr {178.256/256.1780}, \xintIrr {178000/25600000[17]}
```

6856/9853, 695312500000000/1

The current implementation does not cleverly first factor powers of 2 and 5, and \xintIrr {2/3} [100]} will execute the Euclidean division of $2 \cdot 10^{100}$ by 3, which is a bit stupid as it could have known that the 100 trailing zeros can not bring any divisibility by 3.

Starting with release `1.08`, \xintIrr does not remove the trailing `/1` when the output is an integer. This was deemed better for various (questionable?) reasons, anyway the output format is since *always* `A/B` with `B>0`, even in cases where it turns out that `B=1`. Use \xintPRaw on top of \xintIrr if it is needed to get rid of such a trailing `/1`.

9.8. \xintPIrr

Frac
f ★ This puts the fraction into irreducible form, *keeping as is the decimal part* [N] from raw internal
A/B[N] format. (P stands here for *Partial*)

    \xintPIrr {178.256/256.1780}, \xintPIrr {178000/25600000[17]}

3428/49265[1], 89/12800[17]

   Notice that the output always has the ending [N], which is exactly the opposite of \xintIrr's
behaviour. The interest of this macro is mainly in handling fractions which somehow acquired a big
[N] (perhaps from input in scientific notation) and for which the reduced fraction would have a
very large number of digits. This large number of digits can considerably slow-down computations
done afterwards.

   For example package polexpr uses \xintPIrr when differentiating a polynomial, or in setting up
a Sturm chain for localization of the real roots of a polynomial. This is relevant to polynomials
whose coefficients were input in decimal notation, as this automatically creates internally some
[N]. Keeping and combining those [N]'s during computations significantly increases their speed.

9.9. \xintJrr

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

    \xintJrr {178.256/256.178}

6856/9853

   This is (supposedly, not tested for ages) faster than \xintIrr for fractions having some big
common factor in the numerator and the denominator.

    \xintJrr {\xintiiPow\xintiiFac {15}}{3}/%
             \xintiiPrd{{\xintiiFac{10}}{\xintiiFac{30}}{\xintiiFac{5}}}}

1001/51705840

   But to notice the difference one would need computations with much bigger numbers than in this
example. As \xintIrr, \xintJrr does not remove the trailing /1 from a fraction reduced to an inte-
ger.

9.10. \xintPRaw

Frac
f ★ PRaw stands for ``pretty raw''. It does like \xintRaw apart from removing the [N] part if N=0 and
removing the B if B=1.

    \xintPRaw {123e10/321e10}, \xintPRaw {123e9/321e10}, \xintPRaw {\xintIrr{861/123}}

123/321, 123/321[-1], 7

9.11. \xintFracToSci, \xintFracToSciE

New with   \xintFracToSci is not really part of public interface. It is a macro used by \xintexpr and \xint-
1.4   iexpr for output.
   ☆   It is expandable but not *f*-expandable.

   It has specific rules regarding the input format: it expects it argument (after *f*-expanding it)
to *already* be either in raw xintfrac format A/B[N] (with optional denominator and [N] parts) or in
decimal format A.ddd...ddd. It does not accept scientific notation as input (or rather, the e in
input must be of catcode 12).

   Its output uses scientific notation (dropping unit demoninator or zero exponent -- but not (as
alas I have to tell the whole truth) in case input was itself in scientific notation), except for
decimal numbers. The latter pass through ``as is''.

    \xintFracToSci {\xintRaw{123e10/321e10}},
    \xintFracToSci {\xintRaw{123e9/321e10}},
    \xintFracToSci {\xintIrr{861/123}},
    \xintFracToSci {\xintTrunc{12}{1/3}}

123/321, 123e-1/321, 7, 0.333333333333

Not being *f*-expandable it can not be used as argument to the other package macros without being wrapped in \expanded{...}.

\xintFracToSciE says what to use for the scientific notation, its default definition is

    \def\xintFracToSciE{e}

9.12. \xintDecToString

This is a macro tailored for printing decimal numbers. It does not trim trailing zeros, use \xintDecToString{\xintREZ{<foo>}} for that.

    \xintDecToString {123456789e5}\newline
    \xintDecToString {123456789e-5}\newline
    \xintDecToString {12345e-10}\newline
    \xintDecToString {12345e-10/123}\newline % leaves the denominator as is
    \xintDecToString {1234567890000e-6}\newline % does not trim trailing zeros
    \xintDecToString {\xintREZ{1234567890000e-6}}\par % does trim trailing zeros

12345678900000

1234.56789

0.0000012345

0.0000012345/123

1234567.890000

1234567.89

Consider it an unstable macro, what it does exactly is yet to be decided. It is a backport from polexpr's \PolDecToString, which has now been made an alias to it.

9.13. \xintTrunc

\xintTrunc{x}{f} returns the start of the decimal expansion of the fraction f, truncated to:

- if x>0, x digits after the decimal mark,

- if x=0, an integer,

- if x<0, an integer multiple of 10^{-x} (in scientific notation).

The output is the sole digit token 0 if and only if the input was exactly zero; else it contains always either a decimal mark (even if x=0) or a scientific part and it conserves the sign of f (even if the truncated value represents the zero value).

Truncation is done towards zero.

    \xintFor* #1 in {\xintSeq[-1]{7}{-14}}:{\xintTrunc{#1}{-11e12/7}\newline}%
    \xintTrunc{10}{1e-11}\newline
    \xintTrunc{10}{1/65536}\par

-1571428571428.5714285

-1571428571428.571428

-1571428571428.57142

-1571428571428.5714

-1571428571428.571

-1571428571428.57

-1571428571428.5

-1571428571428.

-157142857142e1

-15714285714e2

-1571428571e3

-157142857e4

-15714285e5

-1571428e6

97

```
-157142e7
-15714e8
-1571e9
-157e10
-15e11
-1e12
-0e13
-0e14
0.0000000000
0.0000152587
```

> **Warning:** *it is not yet decided is the current behaviour is definitive.*
>
> Currently xintfrac has no notion of a positive zero or a negative zero. Hence transitivity of \xintTrunc is broken for the case where the first truncation gives on output 0.00...0 or -0.00...0: a second truncation to less digits will then output 0, whereas if it had been applied directly to the initial input it would have produced 0.00...0 or respectively -0.00...0 (with less zeros after decimal mark).
>
> If xintfrac distinguished zero, positive zero, and negative zero then it would be possible to maintain transitivity.
>
> The problem would also be fixed, even without distinguishing a negative zero on input, if \xintTrunc always produced 0.00...0 (with no sign) when the mathematical result is zero, discarding the information on original input being positive, zero, or negative.
>
> I have multiple times hesitated about what to do and must postpone again final decision.

9.14. \xintXTrunc

num Frac
 x   f ☆

\xintXTrunc{x}{f} is similar to \xintTrunc with the following important differences:

- it is completely expandable but not *f-expan*dable, as is indicated by the hollow star in the margin,
- hence it can not be used as argument to the other package macros, but as it *f-expan*ds its {f} argument, it accepts arguments expressed with other xintfrac macros,
- it requires x>0,
- contrarily to \xintTrunc the number of digits on output is not limited to about 19950 and may go well beyond 100000 (this is mainly useful for outputting a decimal expansion to a file),
- when the mathematical result is zero, it always prints it as 0.00...0 or -0.00...0 with x zeros after the decimal mark.

Warning: transitivity is broken too (see discussion of \xintTrunc), due to the sign in the last item. Hence *the definitive policy is yet to be fixed.*

Transitivity is here in the sense of using a first \edef and then a second one, because it is not possible to nest \xintXTrunc directly as argument to itself. Besides, although the number of digits on output isn't limited, nevertheless x should be less than about 19970 when the number of digits of the input (assuming it is expressed as a decimal number) is even bigger: \xintXTrunc{30000}{\Z} after \edef\Z{\xintXTrunc{60000}{1/66049} raises an error in contrast with a direct \xintXTrunc{30000}{1/66049}. But \xintXTrunc{30000}{123.456789} works, because here the number of digits originally present is smaller than what is asked for, thus the routine only has to add trailing zeros, and this has no limitation (apart from TeX main memory).

\xintXTrunc will expand fully in an \edef or a \write (\message, \wlog, ...) or in an \xint-expr-ession, or as list argument to \xintFor*.

Here is an example session where the user checks that the decimal expansion of $1/66049 = 1/257^2$ has the maximal period length $257 * 256 = 65792$ (this period length must be a divisor of $\phi(66049)$ and to check it is the maximal one it is enough to show that neither 32896 nor 256 are periods.)

```
$ rlwrap etex -jobname worksheet-66049
This is pdfTeX, Version 3.14159265-2.6-1.40.17 (TeX Live 2016) (preloaded format=etex)
 restricted \write18 enabled.
**xintfrac.sty
entering extended mode
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xintfrac.sty
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xint.sty
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xintcore.sty
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xintkernel.sty))))
*% we load xinttools for \xintKeep, etc... \xintXTrunc itself has no more

*% any dependency on xinttools.sty since 1.2i

*\input xinttools.sty
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xinttools.sty)
*\def\m#1;{\message{#1}}

*\m \the\numexpr 257*257\relax;
66049
*\m \the\numexpr 257*256\relax;
65792
*% Thus 1/66049 will have a period length dividing 65792.

*% Let us first check it is indeed periodical.

*\edef\Z{\xintXTrunc{66000}{1/66049}}

*% Let's display the first decimal digits.

*\m \xintXTrunc{208}{\Z};

0.0000151402746445820527184363124347075655952398976517434026253236233705279413 7
6856576178291874214598252812306015231116292449545034746930309315810988811337037
6538630410755651107511090251177156353616254598858423
*% let's now fetch the trailing digits

*\m \xintKeep{65792-66000}{\Z};% 208 trailing digits

0000151402746445820527184363124347075655952398976517434026253236233705279413768
5657617829187421459825281230601523111629244954503474693030931581098881133703765
38630410755651107511090251177156353616254598858423
*% yes they match! we now check that 65792/2 and 65792/257=256 aren't periods.

*\m \xintXTrunc{256}{\Z};

0.0000151402746445820527184363124347075655952398976517434026253236233705279413 7
6856576178291874214598252812306015231116292449545034746930309315810988811337037
6538630410755651107511090251177156353616254598858423291798513225029902042423049
55411891171705854744 2
*\m \xintXTrunc{256+256}{\Z};

0.0000151402746445820527184363124347075655952398976517434026253236233705279413 7
6856576178291874214598252812306015231116292449545034746930309315810988811337037
6538630410755651107511090251177156353616254598858423291798513225029902042423049
5541189117170585474420505987978621932201850141561567926842192917379521264515738
3154930430438008145467758785144362518736089872670290239064936637950612424109373
3440324607488379839210283274538600130206361943405653378552286938485064119063119
```

```
8049932625777831609865402958409665551333
*% now with 65792/2=32896. Problem: we can't do \xintXTrunc{32896+100}{\Z}

*% but only direct \xintXTrunc{32896+100}{1/66049}. Anyway we want to nest it

*% hence let's do it all with (slower) \xintKeep, \xintKeepUnbraced.

*\m \xintKeep {-100}{\xintKeepUnbraced{2+65792/2+100}{\Z}};

999984859725355417947281563687565292434404760102348256597374676376629472058 6231
4342382170812578540 17
*% This confirms 32896 isn't a period length.

*% To conclude let's write the 66000 digits to the log.

*\wlog{\Z}

*% We want always more digits:

*\wlog{\xintXTrunc{150000}{1/66049}}

*\bye
```
The acute observer will have noticed that there is something funny when one compares the first digits with those after the middle-period:
```
0000151402746445820527184363124347075655952398976517434026253236233705279413768...
999984859725355417947281563687565292434404760102348256597374676376629472058 6231...
```
Mathematical exercise: can you explain why the two indeed add to 9999...9999?
    You can try your hands at this simpler one:
```
1/49=\xintTrunc{42+5}{1/49}...\newline
\xintTrim{2}{\xintTrunc{21}{1/49}}\newline
\xintKeep{-21}{\xintTrunc{42}{1/49}}
```
1/49=0.020408163265306122448979591836734693877551020400...
020408163265306122448
979591836734693877551

This was again an example of the type 1/N with N the square of a prime. One can also find counter-examples within this class: 1/31^2 and 1/37^2 have an odd period length (465 and respectively 111) hence they can not exhibit the symmetry.

> Mathematical challenge: prove generally that if the period length of the decimal expansion of 1/p^r (with p a prime distinct from 2 and 5 and r a positive exponent) is even, then the previously observed symmetry about the two halves of the period adding to a string of nine's applies.

9.15. \xintTFrac

Frac
f ★ \xintTFrac{f} returns the fractional part, f=trunc(f)+frac(f). Thus if f<0, then –1<frac(f)<=0 and if f>0 one has 0<= frac(f)<1. The T stands for `Trunc', and there should exist also similar macros associated respectively with `Round', `Floor', and `Ceil', each type of rounding to an integer deserving arguably to be associated with a fractional ``modulo''. By sheer laziness, the package currently implements only the ``modulo'' associated with `Truncation'. Other types of modulo may be obtained more cumbersomely via a combination of the rounding with a subsequent subtraction from f.
    Notice that the result is filtered through \xintREZ, and will thus be of the form A/B[N], where neither A nor B has trailing zeros. But the output fraction is not reduced to smallest terms.

100

The function call in expressions (`\xintexpr`, `\xintfloatexpr`) is `frac`. Inside `\xintexpr..\relax`, the function `frac` is mapped to `\xintTFrac`. Inside `\xintfloatexpr..\relax`, `frac` first applies `\xintTFrac` to its argument (which may be an exact fraction with more digits than the floating point precision) and only in a second stage makes the conversion to a floating point number with the precision as set by `\xintDigits` (default is 16).

```
\xintTFrac {1235/97}, \xintTFrac {-1235/97}\newline
\xintTFrac {1235.973}, \xintTFrac {-1235.973}\newline
\xintTFrac {1.122435727e5}\par
```

71/97[0], -71/97[0]
973/1[-3], -973/1[-3]
5727/1[-4]

9.16. `\xintRound`

$\overset{\text{num}}{x}\ \overset{\text{Frac}}{f}$ ⋆ `\xintRound{x}{f}` returns the start of the decimal expansion of the fraction `f`, rounded to:

- if `x>0`, `x` digits after the decimal mark,

- if `x=0`, an integer,

New with
1.4a
- if `x<0`, an integer multiple of `10^{-x}` (in scientific notation).

The output is the sole digit token `0` if and only if the input was exactly zero; else it contains always either a decimal mark (even if `x=0`) or a scientific part and it conserves the sign of `f` (even if the rounded value represents the zero value).

```
\xintFor* #1 in {\xintSeq[-1]{7}{-14}}:{\xintRound{#1}{-11e12/7}\newline}%
\xintRound{10}{1e-11}\newline
\xintRound{10}{1/65536}\newline
```

-1571428571428.5714286
-1571428571428.571429
-1571428571428.57143
-1571428571428.5714
-1571428571428.571
-1571428571428.57
-1571428571428.6
-1571428571429.
-157142857143e1
-15714285714e2
-1571428571e3
-157142857e4
-15714286e5
-1571429e6
-157143e7
-15714e8
-1571e9
-157e10
-16e11
-2e12
-0e13
-0e14
0.0000000000
0.0000152588

Rounding is done with half-way numbers going towards infinity of the same sign.

9.17. `\xintFloor`

$\overset{\text{Frac}}{f}$ ⋆ `\xintFloor {f}` returns the largest relative integer `N` with `N ⩽ f`.

101

```
\xintFloor {-2.13}, \xintFloor {-2}, \xintFloor {2.13}
```
-3/1[0], -2/1[0], 2/1[0] Note the trailing [0], see \xintiFloor if it is not desired.

9.18. \xintCeil

Frac
f ★ \xintCeil {f} returns the smallest relative integer N with N > f.
```
\xintCeil {-2.13}, \xintCeil {-2}, \xintCeil {2.13}
```
-2/1[0], -2/1[0], 3/1[0]

9.19. \xintiTrunc

num Frac
x *f* ★ \xintiTrunc{x}{f} returns the integer equal to 10^x times what \xintTrunc{x}{f} would produce.
Attention that leading zeros are automatically removed: the output is in strict integer format.
```
\xintFor* #1 in {\xintSeq[-1]{7}{-14}}:{\xintiTrunc{#1}{-11e12/7}\newline}%
\xintiTrunc{10}{1e-11}\newline
\xintiTrunc{10}{1/65536}\par
```
-15714285714285714285
-1571428571428571428
-157142857142857142
-15714285714285714
-1571428571428571
-157142857142857
-15714285714285
-1571428571428
-157142857142
-15714285714
-1571428571
-157142857
-15714285
-1571428
-157142
-15714
-1571
-157
-15
-1
0
0
0
152587

9.20. \xintTTrunc

Frac
f ★ \xintTTrunc{f} truncates to an integer (truncation towards zero). This is the same as \xintiTrunc {0}{f} and also the same as \xintNum.

9.21. \xintiRound

num Frac
x *f* ★ \xintiRound{x}{f} returns the integer equal to 10^x times what \xintRound{x}{f} would return. The output has no leading zeroes, it is always in strict integer format.
```
\xintFor* #1 in {\xintSeq[-1]{7}{-14}}:{\xintiRound{#1}{-11e12/7}\newline}%
\xintiRound{10}{1e-11}\newline
\xintiRound{10}{1/65536}\par
```

-15714285714285714286
-1571428571428571429
-157142857142857143
-15714285714285714
-1571428571428571
-157142857142857
-15714285714286
-1571428571429
-157142857143
-15714285714
-1571428571
-157142857
-15714286
-1571429
-157143
-15714
-1571
-157
-16
-2
0
0
0
152588

9.22. \xintiFloor

Frac
f ★ \xintiFloor {f} does the same as \xintFloor but without the trailing /1[0].
  \xintiFloor {-2.13}, \xintiFloor {-2}, \xintiFloor {2.13}
-3, -2, 2

9.23. \xintiCeil

Frac
f ★ \xintiCeil {f} does the same as \xintCeil but its output is without the /1[0].
  \xintiCeil {-2.13}, \xintiCeil {-2}, \xintiCeil {2.13}
-2, -2, 3

9.24. \xintE

Frac num
f *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] Don't feed this example to \xintNum!

9.25. \xintCmp

Frac Frac
f *f* ★ This compares two fractions F and G and produces -1, 0, or 1 according to F<G, F=G, F>G.
  For choosing branches according to the result of comparing f and g, see \xintifCmp.

9.26. \xintEq

Frac Frac
f *f* ★ \xintEq{f}{g} returns 1 if f=g, 0 otherwise.

103

9.27. \xintNotEq

Frac Frac
f   *f*   ★   \xintNotEq{f}{g} returns 0 if f=g, 1 otherwise.

9.28. \xintGeq

Frac Frac
f   *f*   ★   This compares the *absolute values* of two fractions. \xintGeq{f}{g} outputs 1 if |f| ⩾ |g| and 0 if
not.
Important: the macro compares *absolute values*.

9.29. \xintGt

Frac Frac
f   *f*   ★   \xintGt{f}{g} returns 1 if f>g, 0 otherwise.

9.30. \xintLt

Frac Frac
f   *f*   ★   \xintLt{f}{g} returns 1 if f<g, 0 otherwise.

9.31. \xintGtorEq

Frac Frac
f   *f*   ★   \xintGtorEq{f}{g} returns 1 if f⩾g, 0 otherwise. Extended by xintfrac to fractions.

9.32. \xintLtorEq

Frac Frac
f   *f*   ★   \xintLtorEq{f}{g} returns 1 if f⩽g, 0 otherwise.

9.33. \xintIsZero

f ★   \xintIsZero{f} returns 1 if f=0, 0 otherwise.

9.34. \xintIsNotZero

f ★   \xintIsNotZero{f} returns 1 if f!=0, 0 otherwise.

9.35. \xintIsOne

f ★   \xintIsOne{f} returns 1 if f=1, 0 otherwise.

9.36. \xintOdd

f ★   \xintOdd{f} returns 1 if the integer obtained by truncation is odd, and 0 otherwise.

9.37. \xintEven

f ★   \xintEven{f} returns 1 if the integer obtained by truncation is even, and 0 otherwise.

9.38. \xintifSgn

Frac
f *n n n* ★   \xintifSgn{⟨f⟩}{⟨A⟩}{⟨B⟩}{⟨C⟩} executes either the ⟨A⟩, ⟨B⟩ or ⟨C⟩ code, depending on its first
argument being respectively negative, zero, or positive.

9.39. \xintifZero

Frac
f *n n* ★   \xintifZero{⟨f⟩}{⟨IsZero⟩}{⟨IsNotZero⟩} expandably checks if the first mandatory argument N (a
number, possibly a fraction if xintfrac is loaded, or a macro expanding to one such) is zero or
not. It then either executes the first or the second branch.
Beware that both branches must be present.

104

9.40. \xintifNotZero

Frac
f nn ★ \xintifNotZero{⟨N⟩}{⟨IsNotZero⟩}{⟨IsZero⟩} expandably checks if the first mandatory argument f is not zero or is zero. It then either executes the first or the second branch.
Beware that both branches must be present.

9.41. \xintifOne

Frac
f nn ★ \xintifOne{⟨N⟩}{⟨IsOne⟩}{⟨IsNotOne⟩} expandably checks if the first mandatory argument f is one or not one. It then either executes the first or the second branch. Beware that both branches must be present.

9.42. \xintifOdd

Frac
f nn ★ \xintifOdd{⟨N⟩}{⟨odd⟩}{⟨not odd⟩} expandably checks if the first mandatory argument f, after truncation to an integer, is odd or even. It then executes accordingly the first or the second branch. Beware that both branches must be present.

9.43. \xintifCmp

rac Frac
f f nnn ★ \xintifCmp{⟨f⟩}{⟨g⟩}{⟨if f<g⟩}{⟨if f=g⟩}{⟨if f>g⟩} compares its first two arguments and chooses accordingly the correct branch.

9.44. \xintifEq

Frac Frac
f f nn ★ \xintifEq{⟨f⟩}{⟨g⟩}{⟨YES⟩}{⟨NO⟩} checks equality of its two first arguments and executes accordingly the YES or the NO branch.

9.45. \xintifGt

Frac Frac
f f nn ★ \xintifGt{⟨f⟩}{⟨g⟩}{⟨YES⟩}{⟨NO⟩} checks if f > g and in that case executes the YES branch.

9.46. \xintifLt

Frac Frac
f f nn ★ \xintifLt{⟨f⟩}{⟨g⟩}{⟨YES⟩}{⟨NO⟩} checks if f < g and in that case executes the YES branch.

9.47. \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.

9.48. \xintSgn

Frac
f ★ The sign of a fraction.

9.49. \xintOpp

Frac
f ★ The opposite of a fraction. Note that \xintOpp {3} produces -3/1[0] whereas \xintiiOpp {3} produces -3.

9.50. \xintAbs

Frac
f ★ The absolute value. Note that \xintAbs {-2}=2/1[0] where \xintiiAbs {-2} outputs =2.

9.51. \xintAdd

Frac Frac
f *f* ★ Computes the addition of two fractions.

Since 1.3 always uses the least common multiple of the denominators.

9.52. \xintSub

Frac Frac
f *f* ★ Computes the difference of two fractions (\xintSub{F}{G} computes F-G).

Since 1.3 always uses the least common multiple of the denominators.

9.53. \xintMul

Frac Frac
f *f* ★ Computes the product of two fractions.

Output is not reduced to smallest terms.

9.54. \xintDiv

Frac Frac
f *f* ★ Computes the quotient of two fractions. (\xintDiv{F}{G} computes F/G).

Output is not reduced to smallest terms.

9.55. \xintDivFloor

Frac Frac
f *f* ★ Computes the quotient of two arguments then apply floor function to get an integer (in strict format). This macro was defined at 1.1 (but was left not documented until 1.3a...) and changed at 1.2p, formerly it appended /1[0] to output.

```
\xintDivFloor{-170/3}{23/2}
```

-5

9.56. \xintMod

Frac Frac
f *f* ★ Computes the remainder associated to the floored division \xintDivFloor. Prior to 1.2p the meaning was the one of \xintModTrunc. Was left undocumented until 1.3a.

```
\xintMod{-170/3}{23/2}
```

5/6[0]

Modified at 1.3 to use a l.c.m. for the denominator of the result.

9.57. \xintDivMod

Frac Frac
f *f* ★ Computes both the floored division and the remainder \xintDivFloor. New at 1.2p and documented at 1.3a.

```
\oodef\foo{\xintDivMod{-170/3}{23/2}}\meaning\foo
```

macro:->{-5}{5/6[0]}

9.58. \xintDivTrunc

Frac Frac
f *f* ★ Computes the quotient of two arguments then truncates to an integer (in strict format).

```
\xintDivTrunc{-170/3}{23/2}
```

-4

9.59. \xintModTrunc

Frac Frac
f *f* ★ Computes the remainder associated with the truncated division of two arguments. Prior to 1.2p it was named \xintMod, but the latter then got associated with floored division.

```
\xintModTrunc{-170/3}{23/2}
```

-64/6[0]

Modified at 1.3 to use a l.c.m. for the denominator of the result.

9.60. \xintDivRound

Frac Frac
f   *f* ★   Computes the quotient of the two arguments then rounds to an integer (in strict format).

```
\xintDivRound{-170/3}{23/2}
```
-5

9.61. \xintSqr

Frac
f ★   Computes the square of one fraction.

9.62. \xintPow

Frac Num
f   *f* ★   \xintPow{f}{x}: computes f^x with f a fraction and x possibly also, but x will first get truncated to a (positive or negative) integer.

The exponent x must obey the TeX-bound, but this limit is theoretical, as TeX's memory or expansion settings get saturated quite earlier: it is explained in the documentation of \xintiiPow that the maximal power of 2 computable by xint is 2^131072 which has 39457 digits. Actually, the pratical range is even smaller due to execution times.

The output will always be in the form A/B[n] (even if the exponent vanishes: \xintPow {2/3}{0} }=1/1[0]).

Within an \xintiiexpr..\relax the infix operator ^ is mapped to \xintiiPow; within an \xintexpr-ession it is mapped to \xintPow.

9.63. \xintFac

Num
f ★   This is a convenience variant of \xintiiFac which applies \xintNum to its argument. Notice however that the output will have a trailing [0] according to the xintfrac format for integers.

9.64. \xintBinomial

Num Num
f   *f* ★   This is a convenience variant of \xintiiBinomial which applies \xintNum to its arguments. Notice however that the output will have a trailing [0] according to the xintfrac format for integers.

9.65. \xintPFactorial

Num Num
f   *f* ★   This is a convenience variant of \xintiiPFactorial which applies \xintNum to its arguments. Notice however that the output will have a trailing [0] according to the xintfrac format for integers.

9.66. \xintMax

Frac Frac
f   *f* ★   The maximum of two fractions. Beware that \xintMax {2}{3} produces 3/1[0]. The original, for use with integers only with no need of normalization, is available as \xintiiMax: \xintiiMax {2}{3}
f f ★   =3.

```
\xintMax {2.5}{7.2}
```
72/1[-1]

9.67. \xintMin

Frac Frac
f   *f* ★   The minimum of two fractions. Beware that \xintMin {2}{3} produces 2/1[0]. The original, for use with integers only with no need of normalization, is available as \xintiiMin: \xintiiMin {2}{3}
f f ★   =2.

```
\xintMin {2.5}{7.2}
```
25/1[-1]

9.68. \xintMaxof

$f \to * \overset{\text{Frac}}{f}$ ★ The maximum of any number of fractions, each within braces, and the whole thing within braces.

    \xintMaxof {{1.23}{1.2299}{1.2301}} and \xintMaxof {{-1.23}{-1.2299}{-1.2301}}

12301/1[-4] and -12299/1[-4]

9.69. \xintMinof

$f \to * \overset{\text{Frac}}{f}$ ★ The minimum of any number of fractions, each within braces, and the whole thing within braces.

    \xintMinof {{1.23}{1.2299}{1.2301}} and \xintMinof {{-1.23}{-1.2299}{-1.2301}}

12299/1[-4] and -12301/1[-4]

9.70. \xintSum

$f \to * \overset{\text{Frac}}{f}$ ★ This computes the sum of fractions. The output will now always be in the form A/B[n]. The original, for big integers only (in strict format), is available as \xintiiSum.

    \xintSum {{1282/2196921}{-281710/291927}{4028/28612}}

-5037928302100692/6116678670072468[0]

  No simplification attempted.

9.71. \xintPrd

$f \to * \overset{\text{Frac}}{f}$ ★ TThis computes the product of fractions. The output will now always be in the form A/B[n]. The original, for big integers only (in strict format), is available as \xintiiPrd.

    \xintPrd {{1282/2196921}{-281710/291927}{4028/28612}}

-1454721142160/18350036010217404[0]

  No simplification attempted.

    $\xintIsOne {21921379213/21921379213}\neq\xintIsOne {1.00000000000000000000000000000001}$

$1 \neq 0$

9.72. \xintGCD

$\overset{\text{Frac}}{f} \ \overset{\text{Frac}}{f}$ ★ The greatest common divisor of its two arguments, which are possibly *fractions*.

  Prior to 1.4 a macro of the same name existed in xintgcd. But it truncated its two arguments to integers via \xintNum.

  See \xintiiGCD for the integer only variant.

9.73. \xintLCM

$\overset{\text{Frac}}{f} \ \overset{\text{Frac}}{f}$ ★ The least common multiple of its two arguments, which are possibly *fractions*.

  Prior to 1.4 a macro of the same name existed in xintgcd. But it truncated its two arguments to integers via \xintNum.

  See \xintiiLCM for the integer only variant.

9.74. \xintGCDof

$f \to * \overset{\text{Frac}}{f}$ ★ \xintGCDof{{a}{b}{c}...} computes the greatest common divisor of a, b, ... The arguments are allowed to be *fractions*: the macro produces the non-negative generator of the fractional ideal they generate. The list argument may be a macro as it is f-expanded first. If all arguments vanish, then also the output.

  Prior to 1.4 a macro of the same name existed in xintgcd. But it truncated all its arguments to integers via \xintNum and then proceeded with integer only computations.

  See \xintiiGCDof for the integer only variant (which is about 6X faster than this one for integer arguments).

108

9.75. \xintLCMof

^{Frac}
*f → * *f* ★ \xintLCMof{{a}{b}{c}...} computes the least common multiple of a, b, ... The arguments are allowed to be *fractions*: the macro produces the non-negative generator of the intersection of the corresponding fractional ideals. The list argument may be a macro, it is *f*-expanded first. If one of the item vanishes, then also the output.

Prior to 1.4 a macro of the same name existed in xintgcd. But it truncated all its arguments to integers via \xintNum.

See \xintiiLCMof for the integer only variant (which is about 9X faster than this one for integer arguments).

9.76. \xintDigits, \xinttheDigits

The syntax \xintDigits := D; assigns the value of D to the number of digits to be used by floating point operations (this uses internally a \mathchardef assignement, and D stands for (or expands to) a legal TeX number). The default is 16. The maximal value is 32767.

☞ {
    xintexpr adds the variant \xintDigits* which executes \xintreloadxinttrig.
}

★ The expandable macro \xinttheDigits serves to retrieve (internally it uses \number) the current value.
Spaces do not matter as long as they do not occur in-between digits:
```
\xintDigits := 24;\xinttheDigits, %
\xintDigits:=36  ;\xinttheDigits, %
\xintDigits:= 16 ;and \xinttheDigits.
```
24, 36, and 16.

Also \xintDigits = D; (i.e. without a colon) is accepted syntax.
An ending active semi-colon ; is *not* compatible: it can and will cause low-level TeX errors. But this has a trivial workaround: any non-expanding token can be used in place of the ending semi-colon. For example a full stop or a \relax token. This non-expanding ending token will get removed from the token stream.
The recommended syntax is thus now \xintDigits := D\relax (with or without the colon). This is the syntax in use in most examples from the documentation.
```
\xintDigits  = 24\def\xinttheDigits, % only for showing it works! don't do that!
\xintDigits := 36.\xinttheDigits,     % one can use a dot in place of semi-colon
\xintDigits  = 16\relax and \xinttheDigits.\par % with \relax, even better
```
24, 36, and 16.

9.77. \xintSetDigits

^{num}
_x To be used as \xintSetDigits{⟨*expression*⟩} where the expression will be fed to \numexpr. It is a shortcut for doing \xintDigits := \numexpr⟨*expression*⟩\relax \relax.
```
\xintSetDigits{1+2+3+4+5}The value is now \xinttheDigits.
\xintSetDigits{2*8}The value is now \xinttheDigits.\par
```
The value is now 15. The value is now 16.

    See also the xintexpr-added variant \xintSetDigits*.

9.78. \xintFloat

[^{num}_x] ^{Frac}*f* ★ The macro \xintFloat [P]{f} has an optional argument P which replaces the current value of \xinttheDigits. The fraction f is then printed in scientific notation with a rounding to P digits.

That is, on output: the first digit is from `1` to `9`, it is possibly prefixed by a minus sign and is followed by a dot and `P-1` digits, then a lower case `e` and an exponent `N`. The trailing zeroes are not trimmed.

> There is currently one exceptional case: the zero value, which gets output as `0.e0`. It is yet to be decided what the final policy will be.

Starting with `1.2k`, when the input is a fraction `AeN/BeM` the output always is the *correct rounding* to `P` digits. Formerly, this was guaranteed only when `A` and `B` had at most `P+2` digits, or when `B` was `1` and `A` was arbitrary, but in other cases it was only guaranteed that the difference between the original fraction and the rounding was at most `0.6` unit in the last place (of the output), hence the output could differ in the last digit (and earlier ones in case of chains of zeros or nines) from the correct rounding.

Also: for releases `1.2j` and earlier, in the special case when `A/B` ended up being rounded up to the next power of ten, the output was with a mantissa of the shape `10.0...0eN`. However, this worked only for `B=1` or when both `A` and `B` had at most `P+2` digits, because the detection of the rounding-up to next power of ten was done not on original `A/B` but on an approximation `A'/B'`, and it could happen that `A'/B'` was itself being rounded *down* to a power of ten which however was a rounding *up* of original `A/B`. With the `1.2j` refactoring which achieves correct rounding in all cases, it was decided not to add to the code the extra overhead of detecting with 100% fiability the rounding up to next power of ten (such overhead would necessitate alterations of the algorithm and as a result we would end up with a slightly less efficient one; it would make sense in a model where inputs have their intrinsic precisions which is obeyed by the implementation of the basic operations, but currently the design decision for the floating point macros is that when the target precision is `P` the inputs are rounded first to `P` digits before further processing.)

```
{\def\x{9999999999999994999999999999999/99999999999999999999999999999999}%
\xintFor #1 in {13, 14, 15, 16, 17, 18, 19, 47, 48, 49, 50, 79, 80, 81}
\do{#1: \xintFloat[#1]{\x}\xintifForLast{\par}{\newline}}}%
```
13: 1.000000000000e0
14: 1.0000000000000e0
15: 1.00000000000000e0
16: 9.999999999999999e-1
17: 9.9999999999999995e-1
18: 9.9999999999999950e-1
19: 9.99999999999999500e-1
47: 9.9999999999999995000000000000000000000000000000e-1
48: 9.9999999999999994999999999999999999999999999999e-1
49: 9.9999999999999994999999999999999999999999999995e-1
50: 9.9999999999999994999999999999999999999999999950e-1
79: 9.9999999999999994999999999999999999999999999995000000000000000000000000000000e-1
80: 9.9999999999999994999999999999999999999999999994999999999999999999999999999999e-1
81: 9.9999999999999994999999999999999999999999999994999999999999999999999999999995e-1

As an aside, which is illustrated by the above, rounding is not transitive in the number of kept digits.

```
{\def\x{13789378917328973917317/13890138013801398}%
\xintFor* #1 in {\xintSeq{4}{20}}
\do{#1: \xintFloat[#1]{\x}\newline}}%
\xintFloat{5/9999999999999999}\newline
\xintFloat[32]{5/9999999999999999}\newline
\xintFloat[48]{5/9999999999999999}\par
```
4: 9.927e6
5: 9.9275e6
6: 9.92746e6
7: 9.927460e6

```
 8: 9.9274600e6
 9: 9.92745997e6
10: 9.927459975e6
11: 9.9274599746e6
12: 9.92745997457e6
13: 9.927459974572e6
14: 9.9274599745717e6
15: 9.92745997457166e6
16: 9.927459974571665e6
17: 9.9274599745716647e6
18: 9.92745997457166465e6
19: 9.927459974571664655e6
20: 9.9274599745716646545e6
5.000000000000001e-16
5.00000000000000005000000000000001e-16
5.0000000000000000500000000000000005000000000000001e-16
```

9.79. \xintPFloat, \xintPFloatE

$\left[\begin{smallmatrix}\text{num}\\x\end{smallmatrix}\right]\ \overset{\text{Frac}}{f}\ \star$ \xintPFloat [P]{f} is like \xintFloat but ``pretty-prints'' the output. The macro applies one simple rule: x.yz...eN will drop scientific notation in favor of pure decimal notation if -5<=N<=5 (last time I checked Maple® proceeded this way).

Currently trailing zeros are not trimmed. And if the input vanishes the output will be 0. with a decimal mark. Final decision however on how zero value should be printed is yet to arrive...

\xintfloateval applies (via the default definition of \xintfloatexprPrintOne) this macro to each non-empty leaf of the output ople. This is done in an \expanded context, but the macro is already *f*-expandable.

New with 1.4b \xintPFloatE was added to allow customizing the symbol used on output for separating the significand from the exponent, if output uses scientific notation. The separator defaults to e, according to this definition:

```
\def\xintPFloatE{e}
```

See in this context \xintFracToSciE which brings the analogous customizability to \xinteval.

```
\begingroup\def\test #1{#1${}\to{}$\xintPFloat{#1}}%
\string\xintDigits\ at \xinttheDigits
\begin{itemize}[nosep]
\item \test {0}
\item \test {1.234e-7}
\item \test {1.234e-6}
\item \test {1.234e-5}
\item \test {1.234e-4}
\item \test {1.234e-3}
\item \test {1.234e-2}
\item \test {1.234e-1}
\end{itemize}
\def\xintPFloatE{E}% test custom separator. Should impact \xintfloateval as well
\begin{itemize}[nosep]
\item \test {1.234e0}
\item \test {1.234e1}
\item \test {1.234e2}
\item \test {1.234e3}
\item \test {1.234e4}
\item \test {1.234e5}
\item \test {1.234e6}
\item \test {1.234e7}
\end{itemize}
```

```
    \endgroup
\xintDigits at 16
```

- 0 → 0.
- 1.234e-7 → 1.234000000000000e-7
- 1.234e-6 → 1.234000000000000e-6
- 1.234e-5 → 0.00001234000000000000
- 1.234e-4 → 0.0001234000000000000
- 1.234e-3 → 0.001234000000000000
- 1.234e-2 → 0.01234000000000000
- 1.234e-1 → 0.1234000000000000
- 1.234e0 → 1.234000000000000
- 1.234e1 → 12.34000000000000
- 1.234e2 → 123.4000000000000
- 1.234e3 → 1234.000000000000
- 1.234e4 → 12340.00000000000
- 1.234e5 → 123400.0000000000
- 1.234e6 → 1.234000000000000E6
- 1.234e7 → 1.234000000000000E7

9.80. \xintFloatE

$\frac{\text{num}}{x}$ $\frac{\text{Frac num}}{f}$ $\star$ \xintFloatE [P]{f}{x} multiplies the input f by 10^x, and converts it to float format according to the optional first argument or current value of \xinttheDigits.

```
    \xintFloatE {1.23e37}{53}
```
1.230000000000000e90

There is since 1.4b an unfortunate proximity in name with \xintPFloatE despite the two things having absolutely nothing in common.

9.81. \xintFloatAdd

$\frac{\text{num}}{x}$ $\frac{\text{Frac Frac}}{f}$ $\star$ \xintFloatAdd [P]{f}{g} first replaces f and g with their float approximations f' and g' to P significant places or to the precision from \xintDigits. It then produces the sum f'+g', correctly rounded to nearest with the same number of significant places.

9.82. \xintFloatSub

$\frac{\text{num}}{x}$ $\frac{\text{Frac Frac}}{f}$ $\star$ \xintFloatSub [P]{f}{g} first replaces f and g with their float approximations f' and g' to P significant places or to the precision from \xintDigits. It then produces the difference f'-g' correctly rounded to nearest P-float.

9.83. \xintFloatMul

$\frac{\text{num}}{x}$ $\frac{\text{Frac Frac}}{f}$ $\star$ \xintFloatMul [P]{f}{g} first replaces f and g with their float approximations f' and g' to P (or \xinttheDigits) significant places. It then correctly rounds the product f'*g' to nearest P-float.

See subsection 5.2 for more.

> It is obviously much needed that the author improves its algorithms to avoid going through the exact 2P or 2P-1 digits before throwing to the waste-bin half of those digits !

9.84. \xintFloatDiv

\xintFloatDiv [P]{f}{g} first replaces f and g with their float approximations f' and g' to P (or \xinttheDigits) significant places. It then correctly rounds the fraction f'/g' to nearest P-float.

See subsection 5.2 for more.

Notice in the special situation with f and g integers that \xintFloatDiv [P]{f}{g} will *not necessarily* give the correct rounding of the exact fraction f/g. Indeed the macro arguments are each first individually rounded to P digits of precision. The correct syntax to get the correctly rounded integer fraction f/g is \xintFloat[P]{f/g}.

9.85. \xintFloatPow

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

The exponent x will be handed over to a \numexpr, hence count registers are accepted on input for this x. And the absolute value |x| must obey the TeX bound.

The argument f is first rounded to P significant places to give f'. The output Z is such that the exact f'^x differs from Z by an absolute error less than 0.52 ulp(Z).

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

9.86. \xintFloatPower

\xintFloatPower[P]{f}{g} computes a floating point value f^g where the exponent g is not constrained to be at most the TeX bound 2147483647. It may even be a fraction A/B but must simplify to a (possibly big) integer. The exponent of the *output* however *must* at any rate obey the TeX bound.

The argument f is first rounded to P significant places to give f'. The output Z is then such that the exact f'^g differs from Z by an absolute error less than 0.52 ulp(Z).

This is the macro which is used for the ^ (or **) infix operators in \xintthefloatexpr...\relax. In this context (but not directly with the macro,) half-integer exponents are allowed. This is handled via an integer power followed by a square-root extraction. The exponent is first rounded to nearest integer or half-integer so that the computation never raises errors (except naturally for negative exponent and zero f.) The 0.52 ulp(Z) bound applies with half-integer exponents too.

Notice that this is a bound on the distance from f'^g to Z, as f always gets rounded to P or \xinttheDigits digits. The distance from f^g to Z can be much worse if g is very large. Roughly, when g is negligible compared to 10^P, we get an extra difference of up to about 50g ulp(Z) which completely dwarfs the 0.52 ulp(Z). Thus, if f has strictly more than P digits, then the computation must be done with an elevated working precision P'. For example with g=1000 we should use P'=P+6 to achieve a total error at worst slightly bigger than 0.55 ulp(Z) after the final rounding from P' to P digits to get Z.

Examples:[48]

```
\np{\xintFloatPower [8]{3.1415}{3e9}}\newline% Notice that 3e9>2^31
\np{\xintFloatPower [48]{1.1547}{\xintiiPow {2}{35}}}\newline
```
$1.431,772,9 \times 10^{1,491,411,192}$
$2.785,837,382,571,371,438,495,789,880,733,698,213,205,183,990,48 \times 10^{2,146,424,193}$

$2^{35} = 34359738368$ exceeds TeX's bound, but what counts is the exponent of the result which, while dangerously close to 2^{31} is not quite there yet.

With expressions:

```
{\xintDigits:=48\relax \np{\xintthefloatexpr 1.1547^(2^35)\relax}}
```
$2.785,837,382,571,371,438,495,789,880,733,698,213,205,183,990,48 \times 10^{2,146,424,193}$

---

[48] \np is formatting macro from the http://ctan.org/pkg/numprint package.

There is a subtlety here that the `2^35` will be evaluated as a floating point number but fortunately it only has `11` digits, hence the final evaluation is done with a correct exponent. It would have been safer, and also more efficient to code the above rather as:

```
\xintthefloatexpr 1.1547^\xintiiexpr 2^35\relax\relax
```

Here is an example with `12^16` as exponent, which has 18 digits (=184884258895036416).

```
{\xintDigits:=12\relax \np{\xintthefloatexpr (1+1e-8)^\xintiiexpr 12^16\relax\relax}}\newline
\np{\xintthefloatexpr (1+1e-8)^\xintiiexpr 12^16\relax\relax}\newline
{\xintDigits:=27\relax \np{\xintthefloatexpr (1+1e-8)^(12^16)\relax}}\newline
{\xintDigits:=48\relax \np{\xintthefloatexpr (1+1e-8)^(12^16)\relax}}
```

$1.879,985,676,69 \times 10^{802,942,130}$

$1.879,985,676,694,948 \times 10^{802,942,130}$

$1.879,985,676,694,948,388,381,844,07 \times 10^{802,942,130}$

$1.879,985,676,694,948,388,381,844,074,802,295,996,746,413,609,97 \times 10^{802,942,130}$

There is an important difference between `\xintFloatPower[Q]{X}{Y}` and `\xintthefloatexpr[Q]` X^Y\relax: in the former case the computation is done with `Q` digits or precision,[49] whereas with `\xintthefloatexpr[Q]` the evaluation of the expression proceeds with `\xinttheDigits` digits of precision, and the final result is then rounded to `Q` digits: thus this makes real sense only if used with `Q<\xinttheDigits`.

9.87. \xintFloatSqrt

$\begin{bmatrix} \text{num} \\ x \end{bmatrix}$ $\overset{\text{Frac}}{f}$ ★ `\xintFloatSqrt[P]{f}` computes a floating point approximation of $\sqrt{f}$, either using the optional precision `P` or the value of `\xinttheDigits`.

☞ More precisely since `1.2f` the macro achieves so-called *correct rounding*: the produced value is the rounding to `P` significant places of the abstract exact value, *if the input has itself at most P digits* (and an arbitrary exponent).

```
\xintFloatSqrt [89]{10}\newline
\xintFloatSqrt [89]{100}\newline
\xintFloatSqrt [89]{123456789}\par
```

3.1622776601683793319988935444327185337195551393252168268575048527925944386392382213442481e0

1.0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000e1

1.1111111060555555544054166614335346924587840986013435107145857067525147147949636673657913 \hspace{0pt} 6e4

And now some tests to check that correct rounding applies correctly (sic):

```
The argument has 16 digits, hence escapes initial rounding:\newline
\xintFloatSqrt {5625000075000001}\newline
This one gets rounded hence same value is computed:\newline
\xintFloatSqrt {5625000075000001.4}\newline
but actual value is more like:\newline
\xintFloatSqrt [24]{5625000075000001.4}\newline
\xintFloatSqrt [32]{5625000075000001.4}\newline
The argument has 48 digits, hence escapes initial rounding:\newline
\xintFloatSqrt [48]{562500000000000000000000075000000000000000000001}\newline
\xintFloatSqrt [64]{562500000000000000000000075000000000000000000001}\newline
\xintFloatSqrt [80]{562500000000000000000000075000000000000000000001}\newline
```

The argument has 16 digits, hence escapes initial rounding:

7.500000050000000e7

This one gets rounded hence same value is computed:

7.500000050000000e7

but actual value is more like:

7.500000050000000076666666e7

7.5000000500000007666666615555556e7

The argument has 48 digits, hence escapes initial rounding:

7.50000000000000000000000050000000000000000000000e23

---

[49] if `X` and `Y` themselves stand for some floating point macros with arguments, their respective evaluations obey the precision `\xinttheDigits` or as set optionally in the macro calls themselves.

7.500000000000000000000050000000000000000000000050000000000000000e23
7.50000000000000000000005000000000000000000000004999999999999999999999966666667e23
 (we observe in passing illustrations that rounding to nearest is not transitive.)

9.88. \xintFloatFac

[$\overset{\text{num}}{x}$] $\overset{\text{Num}}{f}$ ⋆ \xintFloatFac[P]{f} returns the factorial with either \xinttheDigits or P digits of precision.
   The exact theoretical value differs from the calculated one Y by an absolute error strictly less than 0.6 ulp(Y).
   $1000!\approx{}$\xintFloatFac [30]{1000}
$1000! \approx 4.02387260077093773543702433923e2567$ The computation proceeds via doing explicitly the product, as the Stirling formula cannot be used for lack so far of exp/log.
   The maximal allowed argument is 99999999, but already 100000! currently takes, for 16 digits of precision, a few seconds on my laptop (it returns 2.824229407960348e456573).
   The factorial function is available in \xintfloatexpr:
   \xintthefloatexpr factorial(1000)\relax % same as 1000!
4.023872600770938e2567

9.89. \xintFloatBinomial

[$\overset{\text{num}}{x}$] $\overset{\text{Num}}{f}$ $\overset{\text{Num}}{f}$ ⋆ \xintFloatBinomial[P]{x}{y} computes binomial coefficients with either \xinttheDigits or P digits of precision.
   When x<0 an out-of-range error is raised. Else if y<0 or if x<y the macro evaluates to 0.e0. The exact theoretical value differs from the calculated one Y by an absolute error strictly less than 0.6 ulp(Y).
   ${3000\choose 1500}\approx{}$\xintFloatBinomial [24]{3000}{1500}
$\binom{3000}{1500} \approx 1.79196793754756005073269e901$
   The associated function in \xintfloatexpr is binomial():
   \xintthefloatexpr binomial(3000,1500)\relax
1.791967937547560e901
   The computation is based on the formula (x-y+1)...x/y! (here one arranges y<=x-y naturally).

9.90. \xintFloatPFactorial

[$\overset{\text{num}}{x}$] $\overset{\text{Num}}{f}$ $\overset{\text{Num}}{f}$ ⋆ \xintFloatPFactorial[P]{x}{y} computes the product (x+1)...y.
   The arguments must be integers (they are expanded inside \numexpr) and the allowed range is -100000000 ⩽ x,y ⩽ 99999999. If x ⩾ y the product is considered empty hence returns one (as a floating point value). See also \xintiiPFactorial.
   The exact theoretical value differs from the calculated one Y by an absolute error strictly less than 0.6 ulp(Y).
   The associated function in \xintfloatexpr is pfactorial():
   \xintthefloatexpr pfactorial(2500,5000)\relax
2.595989917947957e8914

9.91. \xintFrac

$\overset{\text{Frac}}{f}$ ⋆ This is a LaTeX only macro, to be used in math mode only. It will print a fraction, internally represented as something equivalent to A/B[n] as \frac {A}{B}10^n. The power of ten is omitted when n=0, the denominator is omitted when it has value one, the number being separated from the power of ten by a \cdot. $\xintFrac {178.000/25600000}$ gives $\frac{178000}{25600000}10^{-3}$, $\xintFrac {178.000/1}$ gives $178000 \cdot 10^{-3}$, $\xintFrac {3.5/5.7}$ gives $\frac{35}{57}$, and $\xintFrac {\xintNum {\xintiiFac{10}/\xintii$ Sqr{\xintiiFac {5}}}}$ gives 252. As shown by the examples, simplification of the input (apart

115

from removing the decimal points and moving the minus sign to the numerator) is not done automatically and must be the result of macros such as \xintIrr, \xintREZ, or \xintNum (for fractions being in fact integers.)

9.92. \xintSignedFrac

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

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

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

9.93. \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 $\xintFwOver {\xintNum {\xintiiFac{10}/\xintiiSqr{\xintiiFac {5}}}}$ gives 252.

9.94. \xintSignedFwOver

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

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

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

9.95. \xintLen

Frac
f ★ The original \xintLen macro is extended to accept a fraction on input: the length of A/B[n] is the length of A plus the length of B plus the absolute value of n and minus one (an integer input as N is internally represented in a form equivalent to N/1[0] so the minus one means that the extended \xintLen behaves the same as the original for integers).

```
\xintLen{201710/298219}=\xintLen{201710}+\xintLen{298219}-1\newline
\xintLen{1234/1}=\xintLen{1234}=\xintLen{1234[0]}=\xintiLen{1234}\newline
\xintLen{-1e3/5.425} (\xintRaw {-1e3/5.425})\par
```

11=6+6-1
4=4=4=4
10 (-1/5425[6])

The length is computed on the A/B[n] which would have been returned by \xintRaw, as illustrated by the last example above.

\xintLen is only for use with such (scientific) numbers or fractions. See also \xintNthElt from xinttools. See also \xintLength (which however does not expand its argument) from xintkernel for counting more general tokens (or rather braced items).

10. Macros of the xintbinhex package

This package provides expandable conversions of (big) integers to and from binary and hexadecimal.

   First version of this package was in the 1.08 (2013/06/07) release of xint. Its routines remained un-modified until their complete rewrite at release 1.2m (2017/07/31). The new macros are faster, using techniques from the 1.2 (2015/10/10) release of xintcore. But the inputs are now limited to a few thousand digits, whereas the 1.08 could handle (slowly...) tens of thousands of digits.

   Table 3 recapitulates the maximal allowed sizes (they got increased at 1.2n): for macro \xint-tFooToBar in the first column, the value in the second column is the maximal N such that \edef\X{\xintFooToBar{<N digits>}} does not raise an error with standard TeX memory parameters (input stack size=5000, expansion depth=10000, parameter stack size=10000). The tests were done with TL2017 and etex. Nested calls will allow slightly lesser values only. The third column gives the corresponding maximal size of output. The fourth column gives the TeX parameter cited in the error message when trying with N+1 digits.

	Max length of input	-> length of output	Limiting factor
\xintDecToHex	6014	4995	input stack size=5000
\xintDecToBin	6014	19979	input stack size=5000
\xintHexToDec	8298	9992	input stack size=5000
\xintBinToDec	19988	6017	input stack size=5000
\xintBinToHex	19988	4997	input stack size=5000
\xintHexToBin	4996	19984	input stack size=5000
\xintCHexToBin	4997	19988	input stack size=5000

Table 3: Maximal sizes of inputs (at 1.2n) for xintbinhex macros

   Roughly, base 10 numbers are limited to 6000 digits, hexadecimal numbers to (almost) 5000 digits, and binary numbers to (almost) 20000 digits. With the surprising exception of \xintHexToDec which allows almost 8300 hexadecimal digits on input.

   The argument is first *f*-expanded. It may optionally have a unique leading minus sign (a plus sign is not allowed), and leading zeroes.

   An input (possibly signed) with no leading zeroes is guaranteed to give an output without leading zero, with the sole, deliberate, exception of \xintCHexToBin: from N hexadecimal digits it produces 4N binary digits, hence possibly with up to three leading zeroes (if the input had none.)

   Inputs with leading zeroes usually produce outputs with an unspecified, case-dependent, number of leading zeroes (\xintBinToHex always uses the minimal number of hexadecimal digits needed to represent the binary digits, inclusive of leading zeroes if present.)

   The macros converting from binary or decimal are robust against non terminated inputs like \the\numexpr 2+3 or \the\mathcode`\-. The macro \xintHexToDec also but not \xintHexToBin and \xint-CHexToBin (anyway there are no primitive in (e)-TeX to my knowledge which will generate hexadecimal digits and may force expansion of next token).

   Hexadecimal digits A..F must be in uppercase. Category code for them on input may be *letter* or *other*. On output they are of category code *letter*, and in uppercase.

   Low-level unrecoverable errors will happen if for example a supposedly binary input contains other digits than 0 and 1. Inputs can not start with a 0b, 0x, #x, " or similar prefix: only digits/letters according to the binary, decimal, or hexadecimal notation.

   With this package loaded additionally to xintexpr, hexadecimal input is possible in expressions: simply by using the prefix ". Such hexadecimal numbers may have a fractional part. Lowercase hexadecimal letters are currently *not* recognized as such in expressions. Currently the p postfix notation from standard programming languages standing for an extra power of two multiplicand is not implemented.

10.1. \xintDecToHex

f ⋆ Converts from decimal to hexadecimal.

\xintDecToHex{271828182845904523536028747135266249775724709369995957496696762772407663035₂
5475945713821785251664274274663919320 03}

->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C6032936BF37DAC918₂
814C63

10.2. \xintDecToBin

f ⋆ Converts from decimal to binary.

\xintDecToBin{271828182845904523536028747135266249775724709369995957496696762772407663035₂
5475945713821785251664274274663919320 03}

->100011010100100111001011111000110011010010100100110101001011100000101000111110111110100001₂
010100000010111100100010100111000111110000010110001011111000100000110110001000111000100100011₂
011101011101111001010110101011101100000101101100111000110100100111001011110100011011011110001₂
110010001101100011000000011001010010011011010101111100110111110110101100100100011000100000010₂
100110001100011

10.3. \xintHexToDec

f ⋆ Converts from hexadecimal to decimal.

\xintHexToDec{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C603₂
2936BF37DAC918814C63}

->271828182845904523536028747135266249775724709369995957496696762772407663035354759457138217₂
85251664274274663919320 03

10.4. \xintBinToDec

f ⋆ Converts from binary to decimal.

\xintBinToDec{1000110101001001110010111110001100110100101001001101010010111000001010001111₂
1011111010000101010000001011110010001010011100011111000001011000101111100010000011011000100₂
1110001001000101110101110111110010101101010101110110000010111011001110001101001001110010111101₂
0001101101110011100100011011000110000000110010100100110110101011111100110111110110101100100100₂
0110001000001010011000110 0011}

->271828182845904523536028747135266249775724709369995957496696762772407663035354759457138217₂
85251664274274663919320 03

10.5. \xintBinToHex

f ⋆ Converts from binary to hexadecimal. The input is first zero-filled to 4N binary digits, hence the
output will have N hexadecimal digits (thus, if the input did not have a leading zero, the output
will not either).

\xintBinToHex{1000110101001001110010111110001100110100101001001101010010111000001010001111₂
1011111010000101010000001011110010001010011100011111000001011000101111100010000011011000100₂
1110001001000101110101110111110010101101010101110110000010111011001110001101001001110010111101₂
0001101101110011100100011011000110000000110010100100110110101011111100110111110110101100100100₂
0110001000001010011000110 0011}

118

->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C6032936BF37DAC918↩
814C63

10.6. \xintHexToBin

f ★ Converts from hexadecimal to binary. Up to three leading zeroes of the output are trimmed.
\xintHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C603↩
2936BF37DAC918814C63}
->1000110101001001110010111110001100110100101001001101010010111000010100011111011111010000100↩
0101000000101111001000101001110001111100000101100010111110001000001101100010001110000100100010↩
0111010111011110010101101010101110110000010111011001110001101001001110010111101000110110111001↩
1100100011011000110000000110010100100110110101111110011011111101101011001001000110001000000100↩
100110001100011

10.7. \xintCHexToBin

f ★ Converts from hexadecimal to binary. Same as \xintHexToBin, but an input with N hexadecimal digits
will give an output with exactly 4N binary digits, leading zeroes are not trimmed.
\xintCHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C60↩
32936BF37DAC918814C63}
->0001000110101001001110010111110001100110100101001001101010010111000010100011111011111010000↩
0010101000000101111001000101001110001111100000101100010111110001000001101100010001110001001010↩
0010111010111011110010101101010101110110000010111011001110001101001001110010111101000110110111↩
0011100100011011000110000000110010100100110110101111110011011111101101011001001000110001000000↩
10100110001100011
This can be combined with \xintBinToHex for round-trips preserving leading zeroes for 4N binary
digits numbers, whereas using \xintHexToBin gives reproducing round-trips only for 4N binary num-
bers numbers not starting with 0000.

This zero-fills to 4N digits the input, hence gives here a leading zero in output:
\xintBinToHex{0001111}\newline
Chaining, we end up with 4N-3 digits, as three binary zeroes are trimmed:
\xintHexToBin{\xintBinToHex{0001111}}\newline
But this will always reproduce the initial input zero-filled to length 4N:
\xintCHexToBin{\xintBinToHex{0001111}}\par
Another example (visible space characters manually inserted):\newline
$000000001111101001010001\xrightarrow{\text{\string\xintBinToHex}}
\xintBinToHex{000000001111101001010001}\xrightarrow{\text{\string\xintHexToBin\hphantom{X}}}
\text{\textvisiblespace\textvisiblespace\textvisiblespace}
\xintHexToBin{\xintBinToHex{000000001111101001010001}}$\newline
$000000001111101001010001\xrightarrow{\text{\string\xintBinToHex}}
\xintBinToHex{000000001111101001010001}\xrightarrow{\text{\string\xintCHexToBin}}
\xintCHexToBin{\xintBinToHex{000000001111101001010001}}$
\par

This zero-fills to 4N digits the input, hence gives here a leading zero in output: 0F
Chaining, we end up with 4N-3 digits, as three binary zeroes are trimmed: 01111
But this will always reproduce the initial input zero-filled to length 4N: 00001111
Another example (visible space characters manually inserted):

$$000000001111101001010001 \xrightarrow{\text{\xintBinToHex}} 00FA51 \xrightarrow{\text{\xintHexToBin}} \text{␣␣␣}000001111101001010001$$

$$000000001111101001010001 \xrightarrow{\text{\xintBinToHex}} 00FA51 \xrightarrow{\text{\xintCHexToBin}} 000000001111101001010001$$

11. Macros of the xintgcd package

This package was included in the original release 1.0 (2013/03/28) of the xint bundle.

At 1.3d macros \xintiiGCD and \xintiiLCM are copied over to xint, hence gcd() and lcm() functions in \xintiiexpr were available simply from loading only xintexpr, and the xintgcd dependency got removed.

☞ {
From 1.1 to 1.3f the package loaded only xintcore, not xint and neither xinttools.

But at 1.4 it loads automatically both xint and xinttools (the latter being a requirement since 1.09h of the \xintTypesetEuclideAlgorithm and \xintTypesetBezoutAlgorithm macros).

The macros \xintiiGCD and \xintiiLCM got relocated into xint. The macros \xintGCD, \xintLCM, \xintGCDof, and \xintLCMof are *removed*: xintfrac provides under these names more powerful macros handling general fractions and not only integers.
}

11.1. \xintBezout

Num Num
f f ★ \xintBezout{N}{M} returns three numbers U, V, D within braces where D is the (non-negative) GCD, and UN + VM = D.
```
\oodef\X{\xintBezout {10000}{1113}}\meaning\X\par
\xintAssign {\xintBezout {10000}{1113}}\to\U\V\D
U: \meaning\U, V: \meaning\V, D: \meaning\D\par
AU+BV: \xinttheiiexpr 10000*\U+1113*\V\relax\par
\noindent\oodef\X{\xintBezout {123456789012345}{9876543210321}}\meaning\X\par
\xintAssign \X\to\U\V\D
U: \meaning\U, V: \meaning\V, D: \meaning\D\par
AU+BV: \xinttheiiexpr 123456789012345*\U+9876543210321*\V\relax
```
macro:->{-131}{1177}{1}
U: macro:->-131, V: macro:->1177, D: macro:->1
AU+BV: 1
macro:->{256654313730}{-3208178892607}{3}
U: macro:->256654313730, V: macro:->-3208178892607, D: macro:->3
AU+BV: 3

11.2. \xintEuclideAlgorithm

Num Num
f f ★ \xintEuclideAlgorithm{N}{M} applies the Euclide algorithm and keeps a copy of all quotients and remainders.
```
\edef\X{\xintEuclideAlgorithm {10000}{1113}}\meaning\X
```
macro:->{5}{10000}{1}{1113}{8}{1096}{1}{17}{64}{8}{2}{1}{8}{0}
The first item is the number of steps, the second is N, the third is the GCD, the fourth is M then the first quotient and remainder, the second quotient and remainder, ... until the final quotient and last (zero) remainder.

11.3. \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.

```
\edef\X{\xintBezoutAlgorithm {10000}{1113}}\printnumber{\meaning\X}
```
macro:->{5}{10000}{0}{1}{1}{1113}{1}{0}{8}{1096}{8}{1}{1}{17}{9}{1}{64}{8}{584}{65}{2}{1}{11)
77}{131}{8}{0}{10000}{1113}

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

11.4. \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}
```
$$123456789012345 = 12 \times 9876543210321 + 4938270488493$$
$$9876543210321 = 2 \times 4938270488493 + 2233335$$
$$4938270488493 = 2211164 \times 2233335 + 536553$$
$$2233335 = 4 \times 536553 + 87123$$
$$536553 = 6 \times 87123 + 13815$$
$$87123 = 6 \times 13815 + 4233$$
$$13815 = 3 \times 4233 + 1116$$
$$4233 = 3 \times 1116 + 885$$
$$1116 = 1 \times 885 + 231$$
$$885 = 3 \times 231 + 192$$
$$231 = 1 \times 192 + 39$$
$$192 = 4 \times 39 + 36$$
$$39 = 1 \times 36 + 3$$
$$36 = 12 \times 3 + 0$$

11.5. \xintTypesetBezoutAlgorithm

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

```
\xintTypesetBezoutAlgorithm {10000}{1113}
```
$$10000 = 8 \times 1113 + 1096$$
$$8 = 8 \times 1 + 0$$
$$1 = 8 \times 0 + 1$$
$$1113 = 1 \times 1096 + 17$$
$$9 = 1 \times 8 + 1$$
$$1 = 1 \times 1 + 0$$
$$1096 = 64 \times 17 + 8$$
$$584 = 64 \times 9 + 8$$
$$65 = 64 \times 1 + 1$$
$$17 = 2 \times 8 + 1$$
$$1177 = 2 \times 584 + 9$$
$$131 = 2 \times 65 + 1$$
$$8 = 8 \times 1 + 0$$
$$10000 = 8 \times 1177 + 584$$
$$1113 = 8 \times 131 + 65$$
$$131 \times 10000 - 1177 \times 1113 = -1$$

121

12. Macros of the xintseries package

This package was first released with version 1.03 (2013/04/14) of the xint bundle.

The $\overset{\text{Frac}}{f}$ expansion type of various macro arguments is only a $\overset{\text{Num}}{f}$ if only xint but not xintfrac is loaded. The macro \xintiSeries is special and expects summing big integers obeying the strict format, even if xintfrac is loaded.

The arguments serving as indices are of the $\overset{\text{num}}{x}$ expansion type.

In some cases one or two of the macro arguments are only expanded at a later stage not immediately.

> Since 1.3, \xintAdd and \xintSub use systematically the least common multiple of the denominators. Some of the comments in this chapter refer to the earlier situation where often the denominators were simply multiplied together. *They have yet to be updated to reflect the new situation brought by the 1.3 release.* Some of these comments may now be off-synced from the actual computation results and thus may be wrong.

12.1. \xintSeries

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

```
\def\coeff #1{\xintiiMON{#1}/#1.5} % (-1)^n/(n+1/2)
\fdef\w {\xintSeries {0}{50}{\coeff}} % we want to re-use it
\fdef\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.
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12} = \xintFrac\z \]
```

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}} = \frac{17390933828737094043211279210162660 2278714}{11002746715939000302527991722603972 9050575}$$

The definition of \coeff as \xintiiMON{#1}/#1.5 is quite suboptimal. It allows #1 to be a big integer, but anyhow only small integers are accepted as initial and final indices (they are of the $\overset{\text{num}}{x}$ type). Second, when the xintfrac parser sees the #1.5 it will remove the dot hence create a denominator with one digit more. For example 1/3.5 turns internally into 10/35 whereas it would be more efficient to have 2/7. For info here is the non-reduced \w:

$$\frac{8695466914368547021605639605081330 1139357}{5501373355796950015126399586130198645252875}10^1$$

It would have been bigger still in releases earlier than 1.1: now, the xintfrac \xintAdd routine does not multiply blindly denominators anymore, it checks if one is a multiple of the other. However it does not practice systematic reduction to lowest terms.

A more efficient way to code \coeff is illustrated next.

```
\def\coeff #1{\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}%
```

```
% The [0] in \coeff is a tiny optimization: in its presence the \xintfracname parser
% sees something which is already in internal format.
\fdef\w {\xintSeries {0}{50}{\coeff}}
\[\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12}=\xintFrac\w\]
```

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

The reduced form `\z` as displayed above only differs from this one by a factor of 1.

```
\def\coeffleibnitz #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}
\cnta 1
\loop
% in this loop we recompute from scratch each partial sum!
% we can afford that, as \xintSeries is fast enough.
\noindent\hbox to 2em{\hfil\texttt{\the\cnta.} }%
         \xintTrunc {12}{\xintSeries {1}{\cnta}{\coeffleibnitz}}\dots
\endgraf
\ifnum\cnta < 30 \advance\cnta 1 \repeat
```

1. 1.000000000000...	11. 0.736544011544...	21. 0.716390450794...
2. 0.500000000000...	12. 0.653210678210...	22. 0.670935905339...
3. 0.833333333333...	13. 0.730133755133...	23. 0.714414166209...
4. 0.583333333333...	14. 0.658705183705...	24. 0.672747499542...
5. 0.783333333333...	15. 0.725371850371...	25. 0.712747499542...
6. 0.616666666666...	16. 0.662871850371...	26. 0.674285961081...
7. 0.759523809523...	17. 0.721695379783...	27. 0.711322998118...
8. 0.634523809523...	18. 0.666139824228...	28. 0.675608712404...
9. 0.745634920634...	19. 0.718771403175...	29. 0.710091471024...
10. 0.645634920634...	20. 0.668771403175...	30. 0.676758137691...

12.2. `\xintiSeries`

$\overset{\text{num}}{x}\,\overset{\text{num}}{x}\,f\,\star$ `\xintiSeries{A}{B}{\coeff}` computes $\sum_{n=A}^{n=B}$`\coeff{n}` where `\coeff{n}` must *f-expan*d to a (possibly long) integer in the strict format.

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

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

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

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

```
            \xintTrunc {40}{\xintiSeries {0}{50}{\coeff}[-40]}\]
    \def\exactcoeff #1%
      {\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}%
    \[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac12}
      = \xintTrunc {50}{\xintSeries {0}{50}{\exactcoeff}}\dots\]
```

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

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n + \frac{1}{2}} = 1.5805993064935250412367895069567264144806868680288367\dots$$

This shows indeed that our sum of truncated terms estimated wrongly the 39th and 40th digits of the exact result[50] and that the sum of rounded terms fared a bit better.

12.3. \xintRationalSeries

\xintRationalSeries{A}{B}{f}{\ratio} evaluates $\sum_{n=A}^{n=B} F(n)$, where F(n) is specified indirectly via the data of f=F(A) and the one-parameter macro \ratio which must be such that \macro{n} expands to F(n)/F(n-1). The name indicates that \xintRationalSeries was designed to be useful in the cases where F(n)/F(n-1) is a rational function of n but it may be anything expanding to a fraction. The macro \ratio must be an expandable-only compatible macro and expand to its value after iterated full expansion of its first item. A and B are fed to a \numexpr hence may be count registers or arithmetic expressions built with such; they must obey the TeX bound. The initial term f may be a macro \f, it will be expanded to its value representing F(A).

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

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

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

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

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

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

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

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

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

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

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

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

---

[50] 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.

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

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

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

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

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

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

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

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

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

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

```
\def\ratio #1{-1/#1[0]}% -1/n, comes from the series of exp(-1)
\cnta 0 % previously declared count
\begin{quote}
\loop
\fdef\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{}\par
\ifnum\cnta<20 \advance\cnta 1 \repeat
\end{quote}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

125

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

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

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

$$\sum_{n=0}^{19} \frac{(-1)^n}{n!} = 0.36787944117144232120\cdots = \frac{447750731559645106}{121645100408832000} = \frac{92079694567171}{250298560512000}$$

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

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 macro

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

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

```
\cnta 0
\def\ratioexp #1#2{\xintDiv{#1}{#2}}% #1/#2
\loop
\noindent
$\sum_{n=0}^{\the\cnta} (.57)^n/n! = \xintTrunc {50}
     {\xintRationalSeries {0}{\cnta}{1}{\ratioexp{.57}}}\dots$
     \vtop to 5pt {}\endgraf
\ifnum\cnta<50 \advance\cnta 10 \repeat
```

$\sum_{n=0}^{0}(.57)^n/n! = 1.00000000000000000000000000000000000000000000000000\ldots$

$\sum_{n=0}^{10}(.57)^n/n! = 1.76826705137947002480668058035714285714285714285714\ldots$

$\sum_{n=0}^{20}(.57)^n/n! = 1.76826705143373515162089324271187082272833005529082\ldots$

$\sum_{n=0}^{30}(.57)^n/n! = 1.76826705143373515162089339282382144915484884979430\ldots$

$\sum_{n=0}^{40}(.57)^n/n! = 1.76826705143373515162089339282382144915485219867776\ldots$

$\sum_{n=0}^{50}(.57)^n/n! = 1.76826705143373515162089339282382144915485219867776\ldots$

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
\begin{multicols}{2}
\loop \fdef\z {\xintRationalSeries
               {\cnta}
               {2*\cnta-1}
               {\xintiiPow {\the\cnta}{\cnta}/\xintiiFac{\cnta}}
               {\ratioexp{\the\cnta}}}%
\fdef\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
\end{multicols}
```

$\sum_{n=1}^{1} \frac{1^n}{n!} / \sum_{n=0}^{1} \frac{1^n}{n!} = 0.50000000\ldots$ $\sum_{n=11}^{21} \frac{11^n}{n!} / \sum_{n=0}^{21} \frac{11^n}{n!} = 0.53907332\ldots$

$\sum_{n=2}^{3} \frac{2^n}{n!} / \sum_{n=0}^{3} \frac{2^n}{n!} = 0.52631578\ldots$ $\sum_{n=12}^{23} \frac{12^n}{n!} / \sum_{n=0}^{23} \frac{12^n}{n!} = 0.53772178\ldots$

$\sum_{n=3}^{5} \frac{3^n}{n!} / \sum_{n=0}^{5} \frac{3^n}{n!} = 0.53804347\ldots$ $\sum_{n=13}^{25} \frac{13^n}{n!} / \sum_{n=0}^{25} \frac{13^n}{n!} = 0.53644744\ldots$

$\sum_{n=4}^{7} \frac{4^n}{n!} / \sum_{n=0}^{7} \frac{4^n}{n!} = 0.54317053\ldots$ $\sum_{n=14}^{27} \frac{14^n}{n!} / \sum_{n=0}^{27} \frac{14^n}{n!} = 0.53525726\ldots$

$\sum_{n=5}^{9} \frac{5^n}{n!} / \sum_{n=0}^{9} \frac{5^n}{n!} = 0.54502576\ldots$ $\sum_{n=15}^{29} \frac{15^n}{n!} / \sum_{n=0}^{29} \frac{15^n}{n!} = 0.53415135\ldots$

$\sum_{n=6}^{11} \frac{6^n}{n!} / \sum_{n=0}^{11} \frac{6^n}{n!} = 0.54518217\ldots$ $\sum_{n=16}^{31} \frac{16^n}{n!} / \sum_{n=0}^{31} \frac{16^n}{n!} = 0.53312615\ldots$

$\sum_{n=7}^{13} \frac{7^n}{n!} / \sum_{n=0}^{13} \frac{7^n}{n!} = 0.54445274\ldots$ $\sum_{n=17}^{33} \frac{17^n}{n!} / \sum_{n=0}^{33} \frac{17^n}{n!} = 0.53217628\ldots$

$\sum_{n=8}^{15} \frac{8^n}{n!} / \sum_{n=0}^{15} \frac{8^n}{n!} = 0.54327992\ldots$ $\sum_{n=18}^{35} \frac{18^n}{n!} / \sum_{n=0}^{35} \frac{18^n}{n!} = 0.53129566\ldots$

$\sum_{n=9}^{17} \frac{9^n}{n!} / \sum_{n=0}^{17} \frac{9^n}{n!} = 0.54191055\ldots$ $\sum_{n=19}^{37} \frac{19^n}{n!} / \sum_{n=0}^{37} \frac{19^n}{n!} = 0.53047810\ldots$

$\sum_{n=10}^{19} \frac{10^n}{n!} / \sum_{n=0}^{19} \frac{10^n}{n!} = 0.54048295\ldots$ $\sum_{n=20}^{39} \frac{20^n}{n!} / \sum_{n=0}^{39} \frac{20^n}{n!} = 0.52971771\ldots$

12.4. \xintRationalSeriesX

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

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

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

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

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

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

```
\def\firstterm #1{1[0]}% first term of the exponential series
% although it is the constant 1, here it must be defined as a
% one-parameter macro. Next comes the ratio function for exp:
\def\ratioexp  #1#2{\xintDiv {#1}{#2}}% x/n
% These are the (-1)^{n-1}/n of the log(1+h) series:
\def\coefflog #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}%
% Let L(h) be the first 10 terms of the log(1+h) series and
% let E(t) be the first 10 terms of the exp(t) series.
% The following computes E(L(a/10)) for a=1,...,12.
\begin{multicols}{3}\raggedcolumns
\cnta 0
\loop
\noindent\xintTrunc {18}{%
     \xintRationalSeriesX {0}{9}{\firstterm}{\ratioexp}
         {\xintPowerSeries{1}{10}{\coefflog}{\the\cnta[-1]}}}\dots
\endgraf
\ifnum\cnta < 12 \advance \cnta 1 \repeat
\end{multicols}
```

1.000000000000000000...   1.499954310225476533...   1.907197560339468199...

1.099999999999083906...   1.599659266069210466...   1.845117565491393752...

1.199999998111624029...   1.698137473697423757...   1.593831932293536053...

1.299999835744121464...   1.791898112718884531...

1.399996091955359088...   1.870485649686617459...

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

E(L(1[-1]))=1635914436931178893034310888060876341482507357910234976572613140141591073957392 0639913787199465741057336677116573252341295218688/14871949426659466386446745600000000000[-90] (length of numerator: 127)

E(L(12[-2]))=1665658335775723446764389561902687419132732099315718324756812577505935601836221 2319343960454005375422644487150283481664480833628821129984588724606679504116088223121980516622 9272737296607284122130748172615228417547299717122/14871949426659466386446745600000000000[-180] (length of numerator: 217)

E(L(123[-3]))=1670119920600555026998663239069002278266215966968669508145191626887938734862 7326998654607865880397901400311690337802593514890044881469893662763355806673815195853060316721 40612785673175692992742863679398303407413205084692383474722719804622771982161117197045873620 2576904911568721571272318238652705503373505331/14871949426659466386446745600000000000[-270] (length of numerator: 307)

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

E(L(1/7))=4822820386275088584804803229765516319371908349875212662294486368347892146353765 04219669541778764527949933/4219967838162271878244372527760312278633063806332105808131741656092 5005693672132881205616128819200000000000[0] (length of numerator: 105; length of denominator: 105)

E(L(1/71))=6190039670785350346406550995159476540272948182884398462882888922997238323197498 8597194001521824905943572083683283923737910672874993163246058732446704305028542916962821162872 5860387813549997353988721286046761040668975799982582643863990035761895246771100033570420271 671092203332984981842891074510835779826956944462566758343900417497150172256263898307611707752 7919998778523559418340083473473151235522560000000000[0] (length of numerator: 203; length of denominator: 203)

E(L(1/712))=3003564353778406020559670408411885925389093114199308387996560136260710297841742 4968192908849580413620381324217440556141531542682924131728705303727345332905581415389151732522 7569411232002636456694953665349180314390511046104875297961920582057259996416578066159049290482 9894646353314666223386924932999351781052209097669684895917631053617755075570396973643592153522 2246041089232853253973804191120212141242471588173404925471664008247098734098515193250428149422 240645967888744414705331478482078635497788470006171032646666387826770190191301139308374215312 810478060202596610291401752576000000000000[0] (length of numerator: 288; length of denominator: 288)

Thus decimal numbers such as 0.123 (equivalently 123[-3]) give less computing intensive tasks than fractions such as 1/712: in the case of decimal numbers the (raw) denominators originate in the coefficients of the series themselves, powers of ten of the input within brackets being treated separately. And even then the numerators will grow with the size of the input in a sort of linear way, the coefficient being given by the order of series: here 10 from the log and 9 from the exp, so 90. One more digit in the input means 90 more digits in the numerator of the output: obviously we can not go on composing such partial sums of series and hope that xint will joyfully do all at the speed of light!

Hence, truncating the output (or better, rounding) is the only way to go if one needs a general calculus of special functions. This is why the package xintseries provides, besides \xint-Series, \xintRationalSeries, or \xintPowerSeries which compute *exact* sums, \xintFxPtPowerSeries for fixed-point computations and a (tentative naive) \xintFloatPowerSeries.

12.5. \xintPowerSeries

num Frac Frac
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

128

is expanded only at the time `\coeff{n}` is needed) should be defined as a one-parameter expandable macro, its input will be an explicit number.

The `f` can be either a fraction directly input or a macro `\f` expanding to such a fraction. It is actually more efficient to encapsulate an explicit fraction `f` in such a macro, if it has big numerators and denominators (`big' means hundreds of digits) as it will then take less space in the processing until being (repeatedly) used.

This macro computes the *exact* result (one can use it also for polynomial evaluation), using a Horner scheme which helps avoiding a denominator build-up (this problem however, even if using a naive additive approach, is much less acute since release 1.1 and its new policy regarding `\xint-Add`).

```
\def\geom #1{1[0]} % the geometric series
\def\f {5/17[0]}
\[ \sum_{n=0}^{n=20} \Bigl(\frac 5{17}\Bigr)^n
 =\xintFrac{\xintIrr{\xintPowerSeries {0}{20}{\geom}{\f}}}
 =\xintFrac{\xinttheexpr (17^21-5^21)/12/17^20\relax}\]
```

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

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

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

$$\log 2 \approx \sum_{n=1}^{50} \frac{1}{n \cdot 2^n} = \frac{6046346975175226566357988455973921}{872303479657928392239462081783398}40$$

```
\setlength{\columnsep}{0pt}
\begin{multicols}{3}
\cnta 1 % previously declared count
\loop   % in this loop we recompute from scratch each partial sum!
% we can afford that, as \xintPowerSeries is fast enough.
\noindent\hbox to 2em{\hfil\texttt{\the\cnta.} }%
        \xintTrunc {12}
            {\xintPowerSeries {1}{\cnta}{\coefflog}{\f}}\dots
\endgraf
\ifnum \cnta < 30 \advance\cnta 1 \repeat
\end{multicols}
```

```
 1. 0.500000000000...       11. 0.693109245355...       21. 0.693147159757...
 2. 0.625000000000...       12. 0.693129590407...       22. 0.693147170594...
 3. 0.666666666666...       13. 0.693138980431...       23. 0.693147175777...
 4. 0.682291666666...       14. 0.693143340085...       24. 0.693147178261...
 5. 0.688541666666...       15. 0.693145374590...       25. 0.693147179453...
 6. 0.691145833333...       16. 0.693146328265...       26. 0.693147180026...
 7. 0.692261904761...       17. 0.693146777052...       27. 0.693147180302...
 8. 0.692750186011...       18. 0.693146988980...       28. 0.693147180435...
 9. 0.692967199900...       19. 0.693147089367...       29. 0.693147180499...
10. 0.693064856150...       20. 0.693147137051...       30. 0.693147180530...
```

```
\def\coeffarctg  #1{1/\the\numexpr\ifodd #1 -2*#1-1\else2*#1+1\fi\relax }%
```

129

```
% 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)
% Notice in passing this aspect of \numexpr:
%        ****  \numexpr -(1)\relax is ilegal !!! ****
\def\f {1/25[0]}% 1/5^2
\[\mathrm{Arctg}(\frac15)\approx \frac15\sum_{n=0}^{15} \frac{(-1)^n}{(2n+1)25^n}
= \xintFrac{\xintIrr {\xintDiv {\xintPowerSeries {0}{15}{\coeffarctg}{\f}}{5}}}\]
```

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

12.6. \xintPowerSeriesX

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

```
\def\ratioexp #1#2{\xintDiv {#1}{#2}}% x/n
% These are the (-1)^{n-1}/n of the log(1+h) series:
\def\coefflog #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}%
% Let L(h) be the first 10 terms of the log(1+h) series and
% let E(t) be the first 10 terms of the exp(t) series.
% The following computes L(E(a/10)-1) for a=1,..., 12.
\begin{multicols}{3}\raggedcolumns
\cnta 1
\loop
\noindent\xintTrunc {18}{%
   \xintPowerSeriesX {1}{10}{\coefflog}
  {\xintSub
      {\xintRationalSeries {0}{9}{1[0]}{\ratioexp{\the\cnta[-1]}}}
      {1}}}\dots
\endgraf
\ifnum\cnta < 12 \advance \cnta 1 \repeat
\end{multicols}
```

0.099999999998556159...
0.199999995263443554...
0.299999338075041781...
0.399974460740121112...

0.499511320760604148...
0.593980619762352217...
0.645144282733914916...
0.398118280111436442...

-1.597091692317639401...
-12.648937932093322763...
-66.259639046914679687...
-304.768437445462801227...

12.7. \xintFxPtPowerSeries

\xintFxPtPowerSeries{A}{B}{\coeff}{f}{D} computes $\sum_{n=A}^{n=B}\coeff{n}\cdot f^n$ with each term of the series truncated to `D` digits after the decimal point. As usual, `A` and `B` are completely expanded through their inclusion in a \numexpr expression. Regarding `D` it will be similarly be expanded each time it is used inside an \xintTrunc. The one-parameter macro \coeff is similarly expanded at the time it is used inside the computations. Idem for `f`. If `f` itself is some complicated macro it is thus better to use the variant \xintFxPtPowerSeriesX which expands it first and then uses the result of that expansion.

The current (1.04) implementation is: the first power f^A is computed exactly, then *truncated*. Then each successive power is obtained from the previous one by multiplication by the exact value

of `f`, and truncated. And `\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/\xintiiFac {#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
\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:

$$\xintPowerSeries\ \{0\}\{19\}\{\coeffexp\}\{\f\}\ = \frac{38682746160036397317757}{63777066403145711616000}$$

= 0.60653065971263342360379915212...

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

12.8. \xintFxPtPowerSeriesX

$\overset{\text{num num}}{\underset{\text{Frac Frac}}{}}\ \overset{X\ X}{\underset{f\ f}{}}\ \overset{\text{num}}{\underset{X}{}}\ \star$ \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
\begin{multicols}2
\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
\end{multicols}
  D(0/100): 0/1[0]                         D(28/100): 4/1[-5]
  D(7/100): 2/1[-5]                        D(35/100): 4/1[-5]
 D(14/100): 2/1[-5]                        D(42/100): 9/1[-5]
 D(21/100): 3/1[-5]                        D(49/100): 42/1[-5]
```
Let's say we evaluate functions on `[-1/2,+1/2]` with values more or less also in `[-1/2,+1/2]` and we want to keep 4 digits of precision. So, roughly we need at least 14 terms in series like the geometric or log series. Let's make this 15. Then it doesn't make sense to compute intermediate summands with more than 6 digits precision. So we compute with 6 digits precision but return only 4 digits (rounded) after the decimal point. This result with 4 post-decimal points precision is then used as input to the next evaluation.

```
\begin{multicols}2
\loop
\noindent \hbox to 2.5cm {\hss\texttt{D(\the\cnta/100): }}%
\dtt{\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
\end{multicols}
  D(0/100): 0                              D(28/100): -0.0001
  D(7/100): 0.0000                         D(35/100): -0.0001
 D(14/100): 0.0000                         D(42/100): -0.0000
 D(21/100): -0.0001                        D(49/100): -0.0001
```
Not bad... I have cheated a bit: the `four-digits precise' numeric evaluations were left unrounded in the final addition. However the inner rounding to four digits worked fine and made the next step faster than it would have been with longer inputs. The morale is that one should not use the raw results of \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 macro to do this final truncating, or better, rounding, at a given number `D'<D` of digits. Maybe for the next release.

12.9. \xintFloatPowerSeries

\xintFloatPowerSeries[P]{A}{B}{\coeff}{f} computes $\sum_{n=A}^{n=B}\coeff{n} \cdot f^n$ with a floating point precision given by the optional parameter `P` or by the current setting of \xintDigits.

In the current, preliminary, version, no attempt has been made to try to guarantee to the final result the precision `P`. Rather, `P` is used for all intermediate floating point evaluations. So rounding errors will make some of the last printed digits invalid. The operations done are first the evaluation of `f^A` using \xintFloatPow, then each successive power is obtained from this first one by multiplication by `f` using \xintFloatMul, then again with \xintFloatMul this is multiplied with \coeff{n}, and the sum is done adding one term at a time with \xintFloatAdd. To sum up, this is just the naive transformation of \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
```

12.10. \xintFloatPowerSeriesX

\xintFloatPowerSeriesX[P]{A}{B}{\coeff}{f} is like \xintFloatPowerSeries with the difference

Frac Frac
 f   *f* ★ 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/\xintiiFac {#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
```

12.11. 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 zeroes or nines in the exact decimal expansion (in the present case of log 2, strings of zeroes around the fortieth and the sixtieth decimals), this does not mean though that all digits printed were always exact. In the end one always end up having to compute at some higher level of desired precision to validate the earlier result.

Then we tried with \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 ≈ 0.69314718055994530941723212145817656807550013436025525412068000711...
     ≈ 0.69314718055994530941723212145817656807550013436025525412068000711...
     ≈ 0.69314718055994530941723212145817656807550013436025525412068000094933723...

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

```
\def\coeffarctg #1{\the\numexpr\ifodd#1 -1\else1\fi\relax/%
                                  \the\numexpr 2*#1+1\relax [0]}%
%\def\coeffarctg #1{\romannumeral0\xintmon{#1}/\the\numexpr 2*#1+1\relax }%
\def\xa {1/25[0]}%       1/5^2, the [0] for faster parsing
\def\xb {1/57121[0]}% 1/239^2, the [0] for faster parsing
\def\Machin #1{% #1 may be a count register, \Machin {\mycount} is allowed
    \romannumeral0\expandafter\MachinA \expandafter
     % number of terms for arctg(1/5):
    {\the\numexpr (#1+3)*5/7\expandafter}\expandafter
     % number of terms for arctg(1/239):
    {\the\numexpr (#1+3)*10/45\expandafter}\expandafter
     % do the computations with 3 additional digits:
    {\the\numexpr #1+3\expandafter}\expandafter
     % allow #1 to be a count register:
    {\the\numexpr #1\relax }}%
\def\MachinA #1#2#3#4%
{\xinttrunc {#4}
 {\xintSub
  {\xintMul {16/5}{\xintFxPtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}}
  {\xintMul{4/239}{\xintFxPtPowerSeries {0}{#2}{\coeffarctg}{\xb}{#3}}}%
 }}%
\begin{framed}
```

134

```
   \[ \pi = \Machin {60}\dots \]
\end{framed}
```

π = 3.14159265358979323846264338327950288419716939937510582097494...

Here is a variant \MachinBis, which evaluates the partial sums *exactly* using \xintPowerSeries, before their final truncation. No need for a ``+3'' then.

```
\def\MachinBis #1{% #1 may be a count register,
% the final result will be truncated to #1 digits post decimal point
    \romannumeral0\expandafter\MachinBisA \expandafter
     % number of terms for arctg(1/5):
    {\the\numexpr #1*5/7\expandafter}\expandafter
     % number of terms for arctg(1/239):
    {\the\numexpr #1*10/45\expandafter}\expandafter
      % allow #1 to be a count register:
    {\the\numexpr #1\relax }}%
\def\MachinBisA #1#2#3%
{\xinttrunc {#3} %
 {\xintSub
    {\xintMul {16/5}{\xintPowerSeries {0}{#1}{\coeffarctg}{\xa}}}
    {\xintMul{4/239}{\xintPowerSeries {0}{#2}{\coeffarctg}{\xb}}}}%
}}%
```
Let us use this variant for a loop showing the build-up of digits:
```
\begin{multicols}{2}
  \cnta 0 % previously declared \count register
  \loop \noindent
        \centeredline{\dtt{\MachinBis{\cnta}}}%
  \ifnum\cnta < 30
  \advance\cnta 1 \repeat
\end{multicols}
```

	3.141592653589793
3.	3.1415926535897932
3.1	3.14159265358979323
3.14	3.141592653589793238
3.141	3.1415926535897932384
3.1415	3.14159265358979323846
3.14159	3.141592653589793238462
3.141592	3.1415926535897932384626
3.1415926	3.14159265358979323846264
3.14159265	3.141592653589793238462643
3.141592653	3.1415926535897932384626433
3.1415926535	3.14159265358979323846264338
3.14159265358	3.141592653589793238462643383
3.141592653589	3.1415926535897932384626433832
3.1415926535897	3.14159265358979323846264338327
3.14159265358979	3.141592653589793238462643383279

You want more digits and have some time? compile this copy of the \Machin with etex (or pdftex):

```
% Compile with e-TeX extensions enabled (etex, pdftex, ...)
\input xintfrac.sty
\input xintseries.sty
% pi = 16 Arctg(1/5) - 4 Arctg(1/239) (John Machin's formula)
\def\coeffarctg #1{\the\numexpr\ifodd#1 -1\else1\fi\relax/%
                                \the\numexpr 2*#1+1\relax [0]}%
```

```
\def\xa {1/25[0]}%
\def\xb {1/57121[0]}%
\def\Machin #1{%
    \romannumeral0\expandafter\MachinA \expandafter
    {\the\numexpr (#1+3)*5/7\expandafter}\expandafter
    {\the\numexpr (#1+3)*10/45\expandafter}\expandafter
    {\the\numexpr #1+3\expandafter}\expandafter
    {\the\numexpr #1\relax }}%
\def\MachinA #1#2#3#4%
{\xinttrunc {#4}
 {\xintSub
  {\xintMul {16/5}{\xintFxPtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}}
  {\xintMul {4/239}{\xintFxPtPowerSeries {0}{#2}{\coeffarctg}{\xb}{#3}}}%
}}%
\pdfresettimer
\fdef\Z {\Machin {1000}}
\odef\W {\the\pdfelapsedtime}
\message{\Z}
\message{computed in \xintRound {2}{\W/65536} seconds.}
\bye
```

This will log the first 1000 digits of π after the decimal point. On my laptop (a 2012 model) this took about 5.05 seconds last time I tried.[51] [52]

As mentioned in the introduction, the file `pi.tex` by D. ROEGEL shows that orders of magnitude faster computations are possible within TeX, but recall our constraints of complete expandability and be merciful, please.

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

But, for the computation of a single constant, we are really interested in the exact decimal expansion, so we truncate and compute more terms until the earlier result gets validated. Finally if we do want the rounding we can always do it on a value computed with D+1 truncation.

---

[51] With 1.09i and earlier xint, this used to be 42 seconds; starting with 1.09j, and prior to 1.2, it was 16 seconds (this was probably due to a more efficient division with denominators at most 9999). The 1.2 xintcore achieves a further gain at 5.6 seconds.

[52] With `\xintDigits :=1001\relax`, the non-optimized implementation with the iter of xintexpr fame using the Brent-Salamin algorithm, took, last time I tried (1.2i), about 7 seconds on my laptop (the last two digits were wrong, which is ok as they serve as guard digits), and for obtaining about 500 digits, it was about 1.7s. This is not bad, taking into account that the syntax is almost free rolling speech, contrarily to the code above for the Machin formula computation; we would like to use the quadratically convergent Brent-Salamin algorithm for more digits, but with such computations with numbers of one thousand digits we are beyond the border of the reasonable range for xint. Innocent people not knowing what it means to compute with TeX, and with the extra constraint of expandability will wonder why this is at least thousands of times slower than with any other language (with a little Python program using the Decimal library, I timed the Brent-Salamin algorithm to 4.4ms for about 1000 digits and 1.14ms for 500 digits.) I will just say that for example digits are represented and manipulated via their ascii-code ! all computations must convert from ascii-code to cpu words; furthermore nothing can be stored away. And there is no memory storage with O(1) time access… if expandability is to be verified.

13. Macros of the xintcfrac package

First version of this package was included in release 1.04 (2013/04/25) of the xint bundle. It was kept almost unchanged until 1.09m of 2014/02/26 which brought some new macros: \xintFtoC, \xint-CtoF, \xintCtoCv, dealing with sequences of braced partial quotients rather than comma separated ones, \xintFGtoC which is to produce ``guaranteed'' coefficients of some real number known approximately, and \xintGGCFrac for displaying arbitrary material as a continued fraction; also, some changes to existing macros: \xintFtoCs and \xintCntoCs insert spaces after the commas, \xintCstoF and \xintCstoCv authorize spaces in the input also before the commas.

Note: \xintCstoF and \xintCstoCv create a partial dependency on xinttools (its \xintCSVtoList.)

13.1. Package overview

The package computes partial quotients and convergents of a fraction, or conversely start from co-efficients and obtain the corresponding fraction; three macros \xintCFrac, \xintGCFrac and \xint-GGCFrac are for typesetting (the first two assume that the coefficients are numeric quantities acceptable by the xintfrac \xintFrac macro, the last one will display arbitrary material), the others can be nested (if applicable) or see their outputs further processed by other macros from the xint bundle, particularly the macros of xinttools dealing with sequences of braced items or comma separated lists.

A *simple* continued fraction has coefficients [c0,c1,...,cN] (usually called partial quotients, but I dislike this entrenched terminology), where c0 is a positive or negative integer and the others are positive integers.

Typesetting is usually done via the amsmath macro \cfrac:
\[c_0 + \cfrac{1}{c_1+\cfrac1{c_2+\cfrac1{c_3+\cfrac1{\ddots}}}}\]

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

Here is a concrete example:
\[\xintFrac {208341/66317}=\xintCFrac {208341/66317}\]%

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

But it is the macro `\xintCFrac` which did all the work of *computing* the continued fraction *and* using `\cfrac` from amsmath to typeset it.

A *generalized* continued fraction has the same structure but the numerators are not restricted to be 1, and numbers used in the continued fraction may be arbitrary, also fractions, irrationals, complex, indeterminates.[53] The *centered* continued fraction is an example:

```
\[ \xintFrac {915286/188421}=\xintGCFrac {5+-1/7+1/39+-1/53+-1/13}
    =\xintCFrac {915286/188421}\]
```

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

The macro `\xintGCFrac`, contrarily to `\xintCFrac`, does not compute anything, it just typesets starting from a generalized continued fraction in inline format, which in this example was input literally. We also used \xintCFrac for comparison of the two types of continued fractions.

To let TeX compute the centered continued fraction of `f` there is `\xintFtoCC`:

```
\[\xintFrac {915286/188421}\to\xintFtoCC {915286/188421}\]
```

$$\frac{915286}{188421} \to 5 + -1/7 + 1/39 + -1/53 + -1/13$$

The package macros are expandable and may be nested (naturally \xintCFrac and \xintGCFrac must be at the top level, as they deal with typesetting).

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

$$5 - \cfrac{1}{7 + \cfrac{1}{39 - \cfrac{1}{53 - \cfrac{1}{13}}}}$$

The `inline' format expected on input by `\xintGCFrac` is

$$a_0 + b_0/a_1 + b_1/a_2 + b_2/a_3 + \cdots + b_{n-2}/a_{n-1} + b_{n-1}/a_n$$

Fractions among the coefficients are allowed but they must be enclosed within braces. Signed integers may be left without braces (but the + signs are mandatory). No spaces are allowed around the plus and fraction symbols. The coefficients may themselves be macros, as long as these macros are *f-expan*dable.

```
\[ \xintFrac{\xintGCtoF {1+-1/57+\xintPow {-3}{7}/\xintiiQuo {132}{25}}}
    = \xintGCFrac {1+-1/57+\xintPow {-3}{7}/\xintiiQuo {132}{25}}\]
```

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

To compute the actual fraction one has `\xintGCtoF`:

---

[53] xintcfrac may be used with indeterminates, for basic conversions from one inline format to another, but not for actual computations. See \xintGGCFrac.

138

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

$$\frac{1907}{1902}$$

For non-numeric input there is `\xintGGCFrac`.

```
\[\xintGGCFrac {a_0+b_0/a_1+b_1/a_2+b_2/\ddots+\ddots/a_{n-1}+b_{n-1}/a_n}\]
```

$$a_0 + \cfrac{b_0}{a_1 + \cfrac{b_1}{a_2 + \cfrac{b_2}{\cdots + \cfrac{\cdot \cdot}{a_{n-1} + \cfrac{b_{n-1}}{a_n}}}}}$$

For regular continued fractions, there is a simpler comma separated format:

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

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

The macro `\xintFtoCs` produces from a fraction `f` the comma separated list of its coefficients.

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

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

If one prefers other separators, one can use the two arguments macros `\xintFtoCx` whose first argument is the separator (which may consist of more than one token) which is to be used.

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

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

This allows under Plain TeX with `amstex` to obtain the same effect as with LaTeX+`\amsmath`+`\xintCFrac`:

```
$$\xintFwOver{2721/1001}=\xintFtoCx {+\cfrac1\\ }{2721/1001}\endcfrac$$
```

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

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

2721/1001=2+1/1+1/2+1/1+1/1+1/4+1/1+1/1+1/6+1/2 Let us compare in that case with the output of `\xintFtoCC`:

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

2721/1001=3+-1/4+-1/2+1/5+-1/2+1/7+-1/2 To obtain the coefficients as a sequence of braced numbers, there is `\xintFtoC` (this is a shortcut for `\xintFtoCx {}`). This list (sequence) may then be manipulated using the various macros of `xinttools` such as the non-expandable macro `\xintAssignArray` or the expandable `\xintApply` and `\xintListWithSep`.

Conversely to go from such a sequence of braced coefficients to the corresponding fraction there is `\xintCtoF`.

The `\printnumber` (subsection 1.4) macro which we use in this document to print long numbers can also be useful on long continued fractions.

```
\printnumber{\xintFtoCC {350370189063507202043510149/244241737886197404558180}}
```

143+1/2+1/5+-1/4+-1/4+-1/4+-1/3+1/2+1/2+1/6+-1/22+1/2+1/10+-1/5+-1/11+-1/3+1/4+-1/2+1/2+1/4+2
-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 -2
1's, the produced fraction is irreducible. And if we compute it again with the last sub-fraction omitted we get another irreducible fraction related to the bigger one by a Bézout identity. Doing this here we get:

139

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

and indeed:

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

The various fractions obtained from the truncation of a continued fraction to its initial terms are called the convergents. The macros of xintcfrac such as \xintFtoCv, \xintFtoCCv, and others which compute such convergents, return them as a list of braced items, with no separator (as does \xintFtoC for the partial quotients). Here is an example:

```
\[\xintFrac{915286/188421}\to
  \xintListWithSep{,}{\xintApply\xintFrac{\xintFtoCv{915286/188421}}}\]
```

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

```
\[\xintFrac{915286/188421}\to
  \xintListWithSep{,}{\xintApply\xintFrac{\xintFtoCCv{915286/188421}}}\]
```

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

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 a function given by a one-parameter macro. The produced values do not have to be integers.

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

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

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

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

140

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

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

There are also `\xintGCntoF` and `\xintGCntoGC` which allow the same for generalized fractions. An initial portion of a generalized continued fraction for π is obtained like this

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

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

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

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

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

When studying the continued fraction of some real number, there is always some doubt about how many terms are valid, when computed starting from some approximation. If $f \leqslant x \leqslant g$ and f, g both have the same first K partial quotients, then x also has the same first K quotients and convergents. The macro `\xintFGtoC` outputs as a sequence of braced items the common partial quotients of its two arguments. We can thus use it to produce a sure list of valid convergents of π for example, starting from some proven lower and upper bound:

```
$$\pi\to [\xintListWithSep{,}
          {\xintFGtoC {3.14159265358979323}{3.14159265358979324}}, \dots]$$
\noindent$\pi\to\xintListWithSep{,\allowbreak\;}
    {\xintApply{\xintFrac}
    {\xintCtoCv{\xintFGtoC {3.14159265358979323}{3.14159265358979324}}}}, \dots$
```

$$\pi \to [3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, \ldots]$$

141

$\pi \to 3, \frac{22}{7}, \frac{333}{106}, \frac{355}{113}, \frac{103993}{33102}, \frac{104348}{33215}, \frac{208341}{66317}, \frac{312689}{99532}, \frac{833719}{265381}, \frac{1146408}{364913}, \frac{4272943}{1360120}, \frac{5419351}{1725033}, \frac{80143857}{25510582},$
$\frac{165707065}{52746197}, \frac{245850922}{78256779}, \frac{411557987}{131002976}, \cdots$

13.2. \xintCFrac

Frac
f

\xintCFrac{f} is a math-mode only, LATEX with amsmath only, macro which first computes then displays with the help of \cfrac the simple continued fraction corresponding to the given fraction. 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 \xint-Frac macro from the xintfrac package. This macro is *f*-expandable in the sense that it prepares expandably the whole expression with the multiple \cfrac's, but it is not completely expandable naturally as \cfrac isn't.

13.3. \xintGCFrac

f

\xintGCFrac{a+b/c+d/e+f/g+h/...+x/y} uses similarly \cfrac to prepare the typesetting with the amsmath \cfrac (LATEX) of a generalized continued fraction given in inline format (or as macro which will *f*-expand to it). It admits the same optional argument as \xintCFrac. Plain TEX with amstex users, see \xintGCtoGCx.

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

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

This is mostly a typesetting macro, although it does provoke the expansion of the coefficients. See \xintGCtoF if you are impatient to see this specific fraction computed.

It admits an optional argument within square brackets which may be either [l], [c] or [r]. Default is [c] (numerators are centered).

Numerators and denominators are made arguments to the \xintFrac macro. This allows them to be themselves fractions or anything *f*-expandable giving numbers or fractions, but also means however that they can not be arbitrary material, they can not contain color changing macros for example. One of the reasons is that \xintGCFrac tries to determine the signs of the numerators and chooses accordingly to use + or -.

13.4. \xintGGCFrac

f

\xintGGCFrac{a+b/c+d/e+f/g+h/...+x/y} is a clone of \xintGCFrac, hence again LATEX specific with package amsmath. It does not assume the coefficients to be numbers as understood by xintfrac. The macro can be used for displaying arbitrary content as a continued fraction with \cfrac, using only plus signs though. Note though that it will first *f*-expand its argument, which may be thus be one of the xintcfrac macros producing a (general) continued fraction in inline format, see \xintFtoCx for an example. If this expansion is not wished, it is enough to start the argument with a space.

    \[\xintGGCFrac {1+q/1+q^2/1+q^3/1+q^4/1+q^5/\ddots}\]

$$1 + \cfrac{q}{1 + \cfrac{q^2}{1 + \cfrac{q^3}{1 + \cfrac{q^4}{1 + \cfrac{q^5}{\ddots}}}}}$$

13.5. \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

The following can be used byt Plain TeX+amstex users to obtain an output similar as the ones produced by \xintGCFrac and \xintGGCFrac:

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

13.6. \xintFtoC

Frac
f ★ \xintFtoC{f} computes the coefficients of the simple continued fraction of f and returns them as a list (sequence) of braced items.

    \fdef\test{\xintFtoC{-5262046/89233}}\texttt{\meaning\test}

macro:->{-59}{33}{27}{100}

13.7. \xintFtoCs

Frac
f ★ \xintFtoCs{f} returns the comma separated list of the coefficients of the simple continued fraction of f. Notice that starting with 1.09m a space follows each comma (mainly for usage in text mode, as in math mode spaces are produced in the typeset output by TeX itself).

    \[\xintSignedFrac{-5262046/89233} \to [\xintFtoCs{-5262046/89233}]\]

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

13.8. \xintFtoCx

Frac
n f ★ \xintFtoCx{sep}{f} returns the list of the coefficients of the simple continued fraction of f separated with the help of sep, which may be anything (and is kept unexpanded). For example, with Plain TeX and amstex,

    $$\xintFtoCx {+\cfrac1\\ }{-5262046/89233}\endcfrac$$

will display the continued fraction using \cfrac. Each coefficient is inside a brace pair { }, allowing a macro to end the separator and fetch it as argument, for example, again with Plain TeX and amstex:

    \def\highlight #1{\ifnum #1>200 \textcolor{red}{#1}\else #1\fi}
    $$\xintFtoCx {+\cfrac1\\ \highlight}{104348/33215}\endcfrac$$

Due to the different and extremely cumbersome syntax of \cfrac under LaTeX it proves a bit tortuous to obtain there the same effect. Actually, it is partly for this purpose that 1.09m added \xint-GGCFrac. We thus use \xintFtoCx with a suitable separator, and then the whole thing as argument to \xintGGCFrac:

    \def\highlight #1{\ifnum #1>200 \fcolorbox{blue}{white}{\boldmath\color{red}$#1$}%
        \else #1\fi}
    \[\xintGGCFrac {\xintFtoCx {+1/\highlight}{208341/66317}}\]

$$3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{\boxed{292} + \cfrac{1}{2}}}}}$$

13.9. \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'.

    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

13.10. \xintFGtoC

Frac Frac
f *f* ★   \xintFGtoC{f}{g} computes the common initial coefficients to two given fractions f and g. Notice that any real number f<x<g or f>x>g will then necessarily share with f and g these common initial coefficients for its regular continued fraction. The coefficients are output as a sequence of braced numbers. This list can then be manipulated via macros from xinttools, or other macros of xintcfrac.

    \fdef\test{\xintFGtoC{-5262046/89233}{-5314647/90125}}\texttt{\meaning\test}

macro:->{-59}{33}{27}

    \fdef\test{\xintFGtoC{3.141592653}{3.141592654}}\texttt{\meaning\test}

macro:->{3}{7}{15}{1}

    \fdef\test{\xintFGtoC{3.1415926535897932384}{3.1415926535897932385}}\meaning\test

macro:->{3}{7}{15}{1}{292}{1}{1}{1}{2}{1}{3}{1}{14}{2}{1}{1}{2}{2}{2}

    \xintRound {30}{\xintCstoF{\xintListWithSep{,}{\test}}}

3.141592653589793238386377506390

    \xintRound {30}{\xintCtoF{\test}}

3.141592653589793238386377506390

    \fdef\test{\xintFGtoC{1.41421356237309}{1.4142135623731}}\meaning\test

macro:->{1}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}

13.11. \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

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

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

13.12. \xintCstoF

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

☞ { *Usage of this macro requires the user to load* xinttools.
   Starting with release 1.09m spaces before commas are allowed and trimmed automatically (spaces after commas were already silently handled in earlier releases).

144

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

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

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

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

A generalized continued fraction may produce a reducible fraction (\xintCstoF tries its best not to accumulate in a silly way superfluous factors but will not do simplifications which would be obvious to a human, like simplification by 3 in the result above).

13.13. \xintCtoF

f ⋆ \xintCtoF{{a}{b}{c}...{z}} computes the fraction corresponding to the coefficients, which may be fractions or even macros.
```
  \xintCtoF {\xintApply {\xintiiPow 3}{\xintSeq {1}{5}}}
14946960/4805083
  \[ \xintFrac{14946960/4805083}=\xintCFrac {14946960/4805083}\]
```

$$\frac{14946960}{4805083} = 3 + \cfrac{1}{9 + \cfrac{1}{27 + \cfrac{1}{81 + \frac{1}{243}}}}$$

In the example above the power of 3 was already pre-computed via the expansion done by \xintApply, but if we try with \xintApply { \xintiiPow 3} where the space will stop this expansion, we can check that \xintCtoF will itself provoke the needed coefficient expansion.

13.14. \xintGCtoF

f ⋆ \xintGCtoF{a+b/c+d/e+f/g+......+v/w+x/y} computes the fraction defined by the inline generalized continued fraction. Coefficients may be fractions but must then be put within braces. They can be macros. The plus signs are mandatory.
```
  \[\xintGCFrac {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintiiFac {6}} =
  \xintFrac{\xintGCtoF {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintiiFac {6}}} =
  \xintFrac{\xintIrr{\xintGCtoF
              {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintiiFac {6}}}}\]
```

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

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

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

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.

13.15. \xintCstoCv

f ⋆ \xintCstoCv{a,b,c,d,...,z} returns the sequence of the corresponding convergents, each one within braces.

☞ *Usage of this macro requires the user to load* xinttools.
It is allowed to use fractions as coefficients (the computed convergents have then no reason to be the real convergents of the final fraction). When the coefficients are integers, the convergents are irreducible fractions, but otherwise it is not necessarily the case.

```
\xintListWithSep:{\xintCstoCv{1,2,3,4,5,6}}
```
1/1:3/2:10/7:43/30:225/157:1393/972
```
\xintListWithSep:{\xintCstoCv{1,1/2,1/3,1/4,1/5,1/6}}
```
1/1:3/1:9/7:45/19:225/159:1575/729
```
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintCstoCv {\xintPow
       {-.3}{-5},7.3/4.57,\xintCstoF{3/4,9,-1/3}}}}\]
```

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

13.16. \xintCtoCv

f ⋆ \xintCtoCv{{a}{b}{c}...{z}} returns the sequence of the corresponding convergents, each one within braces.

```
\fdef\test{\xintCtoCv {111111111111}}\texttt{\meaning\test}
```
macro:->{1/1}{2/1}{3/2}{5/3}{8/5}{13/8}{21/13}{34/21}{55/34}{89/55}{144/89}

13.17. \xintGCtoCv

f ⋆ \xintGCtoCv{a+b/c+d/e+f/g+......+v/w+x/y} returns the list of the corresponding convergents. The coefficients may be fractions, but must then be inside braces. Or they may be macros, too.
The convergents will in the general case be reducible. To put them into irreducible form, one needs one more step, for example it can be done with \xintApply\xintIrr.

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

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

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

13.18. \xintFtoCv

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

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

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

13.19. \xintFtoCCv

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

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

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

13.20. \xintCntoF

^{num} *x f* ★ \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]

   This example shows that the fraction is output with a trailing number in square brackets (representing a power of ten), this is for consistency with what do most macros of xintfrac, and does not have to be always this annoying [0] as the coefficients may for example be numbers in scientific notation. To avoid these trailing square brackets, for example if the coefficients are known to be integers, there is always the possibility to filter the output via \xintPRaw, or \xintIrr (the latter is overkill in the case of integer coefficients, as the fraction is guaranteed to be irreducible then).

13.21. \xintGCntoF

^{num} *x f f* ★ \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.

    \def\coeffA #1{\the\numexpr #1+4-3*((#1+2)/3)\relax }%
    \def\coeffB #1{\the\numexpr \ifodd #1 -\fi 1\relax }% (-1)^n
    \[\xintGCFrac{\xintGCntoGC {6}{\coeffA}{\coeffB}} =
                            \xintFrac{\xintGCntoF {6}{\coeffA}{\coeffB}}\]

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

There is also \xintGCntoGC to get the `inline format' continued fraction.

13.22. \xintCntoCs

num
x f ★ \xintCntoCs{N}{\macro} produces the comma separated list of the corresponding coefficients, from
n=0 to n=N. The N is given to a \numexpr.
   \xintCntoCs {5}{\macro}
1, 2, 5, 10, 17, 26
   \[\xintFrac{\xintCntoF{5}{\macro}}=\xintCFrac{\xintCntoF {5}{\macro}}\]

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

13.23. \xintCntoGC

num
x f ★ \xintCntoGC{N}{\macro} evaluates the c(j)=\macro{j} from j=0 to j=N and returns a continued frac-
tion 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}
   \fdef\x{\xintCntoGC {5}{\macro}}\meaning\x
   \[\xintGCFrac{\xintCntoGC {5}{\macro}}\]
macro:->{1/\the \numexpr 1+0*0\relax }+1/{-2/\the \numexpr 1+1*1\relax }+1/{3/\the \numexpr
1+2*2\relax }+1/{-4/\the \numexpr 1+3*3\relax }+1/{5/\the \numexpr 1+4*4\relax }+1/{-6/\the
\numexpr 1+5*5\relax }

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

13.24. \xintGCntoGC

num
x f f ★ \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. The coefficients are enclosed into pairs of braces, and may thus be fractions, the fraction
slash will not be confused in further processing by the continued fraction slashes.
   \def\an #1{\the\numexpr #1*#1*#1+1\relax}%
   \def\bn #1{\the\numexpr \ifodd#1 -\fi 1*(#1+1)\relax}%
   $\xintGCntoGC {5}{\an}{\bn}=\xintGCFrac {\xintGCntoGC {5}{\an}{\bn}} =
   \displaystyle\xintFrac {\xintGCntoF {5}{\an}{\bn}}$\par

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

148

13.25. `\xintCstoGC`

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

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

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

13.26. `\xintiCstoF`, `\xintiGCtoF`, `\xintiCstoCv`, `\xintiGCtoCv`

f ⋆ Essentially the same as the corresponding macros without the `i', but for integer-only input. Infinitesimally faster, mainly for internal use by the package.

13.27. `\xintGCtoGC`

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

```
\fdef\x {\xintGCtoGC {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/%
                     \xintiiFac {6}+\xintCstoF {2,-7,-5}/16}} \meaning\x
macro:->{1}+{3375/1[-3]}/{1/7}+{-3/5}/{720}+{67/36}/{16}
```

To be honest I have forgotten for which purpose I wrote this macro in the first place.

13.28. Euler's number e

Let us explore the convergents of Euler's number e. The volume of computation is kept minimal by the following steps:

- 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...

```
\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\dtt{\the\cnta.} }%
               $\xintTrunc {30}{\xintAdd {1[0]}{#1}}\dots=
```

149

```
                    \xintFrac{\xintAdd {1[0]}{#1}}$}%
    \xintListWithSep{\vtop to 6pt{}\vbox to 12pt{}\par}
        {\xintApply\mymacro{\xintiCstoCv{\xintCntoCs {35}{\cn}}}}
```

1. $2.0000000000000000000000000000000\cdots = 2$

2. $3.0000000000000000000000000000000\cdots = 3$

3. $2.6666666666666666666666666666666\cdots = \frac{8}{3}$

4. $2.7500000000000000000000000000000\cdots = \frac{11}{4}$

5. $2.7142857142857142857142857142857\cdots = \frac{19}{7}$

6. $2.7187500000000000000000000000000\cdots = \frac{87}{32}$

7. $2.7179487179487179487179487179487948717948\cdots = \frac{106}{39}$

8. $2.7183098591549295774647887323943\cdots = \frac{193}{71}$

9. $2.7182795698924731182795698924731\cdots = \frac{1264}{465}$

10. $2.7182835820895522388059701492537\cdots = \frac{1457}{536}$

11. $2.7182817182817182817182817182817\cdots = \frac{2721}{1001}$

12. $2.7182818352059925093632958801491\cdots = \frac{23225}{8544}$

13. $2.7182818229439497118910424305911\cdots = \frac{25946}{9545}$

14. $2.7182818287356957266847255237981\cdots = \frac{49171}{18089}$

15. $2.7182818284454013180350250741721\cdots = \frac{517656}{190435}$

16. $2.7182818284705837217778289309621\cdots = \frac{566827}{208524}$

17. $2.7182818284585634112778506062021\cdots = \frac{1084483}{398959}$

18. $2.7182818284590651140745295466481\cdots = \frac{13580623}{4996032}$

19. $2.7182818284590280132070655910261\cdots = \frac{14665106}{5394991}$

20. $2.7182818284590458514046210849491\cdots = \frac{28245729}{10391023}$

21. $2.7182818284590452135219837582211\cdots = \frac{410105312}{150869313}$

22. $2.7182818284590452546247950270921\cdots = \frac{438351041}{161260336}$

23. $2.7182818284590452347575606314791\cdots = \frac{848456353}{312129649}$

24. $2.7182818284590452353790133727721\cdots = \frac{14013652689}{5155334720}$

25. $2.7182818284590452353435355327871\cdots = \frac{14862109042}{5467464369}$

26. $2.7182818284590452353607532301881\cdots = \frac{28875761731}{10622799089}$

27. $2.7182818284590452353602745939411\cdots = \frac{534625820200}{196677847971}$

28. $2.7182818284590452353602991209111\cdots = \frac{563501581931}{207300647060}$

29. $2.7182818284590452353602871799001\cdots = \frac{1098127402131}{403978495031}$

30. $2.7182818284590452353602874786111\cdots = \frac{22526049624551}{8286870547680}$

31. $2.7182818284590452353602874647261\cdots = \frac{23624177026682}{8690849042711}$

32. $2.7182818284590452353602874715032\cdots = \frac{46150226651233}{16977719590391}$

33. $2.7182818284590452353602874713492\cdots = \frac{1038929163353808}{382200680031313}$

34. $2.7182818284590452353602874713552\cdots = \frac{1085079390005041}{399178399621704}$

35. $2.7182818284590452353602874713522\cdots = \frac{2124008553358849}{781379079653017}$

36. $2.7182818284590452353602874713522\cdots = \frac{52061284670617417}{19152276311294112}$

One can with no problem compute much bigger convergents. Let's get the 200th convergent. It turns out to have the same first 268 digits after the decimal point as e - 1. Higher convergents get more and more digits in proportion to their index: the 500th convergent already gets 799 digits correct! To allow speedy compilation of the source of this document when the need arises, I limit here to the 200th convergent.

```
\fdef\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 = 568964038871896267597523892315807875293889017667917446057232024547192296961118)
23017524386017499531081773136701241708609749634329382906

Denominator = 331123817669737619306256360816356753365468823729314438156205615463246659728581)
865461337692063148916019550614570592553376611426452172233

Expansion = 1.71828182845904523536028747135266249775724709369995957496696762772407663035352)
475945713821785251664274274663919320030599218174135966290435729003342952605956)
307381323286279434907632338298807531952510190115738341879307021540891499348841)
6750924476146066808226480016847741118...

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.

14. Macros of the xinttools package

These utilities used to be provided within the xint package; since 1.09g (2013/11/22) they have been moved to an independently usable package xinttools, which has none of the xint facilities regarding big numbers. Whenever relevant release 1.09h has made the macros \long so they accept \par tokens on input.

The completely expandable utilities (up to \xintiloop) are documented first, then the non expandable utilities.

section 16 gives additional (also dated) examples of use of macros of this package.

14.1. \xintRevWithBraces

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

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

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

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

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

14.2. \xintZapFirstSpaces, \xintZapLastSpaces, \xintZapSpaces, \xintZapSpacesB

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

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

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

It expands in two steps, and if the goal is to apply it to the expansion text of \x to define \y, then one can do: \odef\y{\romannumeral0\expandafter\xintzapfirstspaces\expandafter{\x}} (one can also define a wrapper macro to \xintZapFirstSpaces in order to expand once the argument first, but xinttools not being a programming layer, it provides no «Generate Variants» facilities).

Other use case: inside a macro which received a parameter #1, one can do \oodef\x{\xintZapFirstSpaces {#1}}, or, if #1, after leading spaces have been stripped can accept \edef expansion, one can do \edef\x{\xintZapFirstSpaces{#1}}.

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

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

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

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

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

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

14.3. \xintCSVtoList

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

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

n ★ 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[54] are removed , as well as the spaces at the start and the spaces at the end of the list.[55] The items may contain explicit `\par`'s or empty lines (converted by the TeX input parsing into `\par` tokens).

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

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

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

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

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

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

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

Else, we may have direct use:

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

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

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

14.4. `\xintNthElt`

`\xintNthElt{x}{⟨list⟩}` gets (expandably) the xth item of the ⟨*list*⟩. A braced item will lose one level of brace pairs. The token list is first *f-expan*ded.

---

[54] and multiple space tokens are not a problem; but those at the top level (not hidden inside braces) *must* be of character code 32. [55] 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.

Items are counted starting at one.
```
\xintNthElt {3}{{agh}\u{zzz}\v{Z}} is zzz
\xintNthElt {3}{{agh}\u{{zzz}}\v{Z}} is {zzz}
\xintNthElt {2}{{agh}\u{{zzz}}\v{Z}} is \u
\xintNthElt {37}{\xintiiFac {100}}=9 is the thirty-seventh digit of 100!.
\xintNthElt {10}{\xintFtoCv {566827/208524}}=1457/536
is the tenth convergent of 566827/208524 (uses xintcfrac package).
\xintNthElt {7}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=7
\xintNthElt {0}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=9
\xintNthElt {-3}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=7
```

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

If x<0, the macro returns the |x|th element from the end of the list. Thus for example x=-1 will fetch the last item of the list.
```
\xintNthElt {-5}{{{agh}}\u{zzz}\v{Z}} is {agh}
```

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

If x is strictly larger (in absolute value) than the length of the list then \xintNthElt produces empty contents.

14.5. \xintNthOnePy

num
x f ⋆ \xintNthOnePy{x}{⟨list⟩} gets (expandably) the xth item of the ⟨list⟩, adding a brace pair if there
New with wasn't one.
1.4
Attention, items are counted starting at zero. For negative index, behaves as \xintNthElt.

If the index is out of range, the empty output is returned. If the input list was empty (had no items) the empty output is returned.

14.6. \xintKeep

num
x f ⋆ \xintKeep{x}{⟨list⟩} expands the token list argument L and produces a new list, depending on the value of x:

- if x>0, the new list contains the first x items from L (counting starts at one.) *Each such item will be output within a brace pair.* Use \xintKeepUnbraced if this is not desired. This means that if the list item was braced to start with, there is no modification, but if it was a token without braces, then it acquires them.
- if x>=length(L), the new list is the old one with all its items now braced.
- if x=0 the empty list is returned.
- if x<0 the last |x| elements compose the output in the same order as in the initial list; as the macro proceeds by removing head items the kept items end up in output as they were in input: no added braces.
- if x<=-length(L) the output is identical with the input.

\xintKeepNoExpand does the same without first *f*-expanding its list argument.
```
\fdef\test {\xintKeep {17}{\xintKeep {-69}{\xintSeq {1}{100}}}}\meaning\test\par
\noindent\fdef\test {\xintKeep {7}{{1}{2}{3}{4}{5}{6}{7}{8}{9}}}\meaning\test\par
\noindent\fdef\test {\xintKeep {-7}{{1}{2}{3}{4}{5}{6}{7}{8}{9}}}\meaning\test\par
\noindent\fdef\test {\xintKeep {7}{123456789}}\meaning\test\par
\noindent\fdef\test {\xintKeep {-7}{123456789}}\meaning\test\par
```
```
macro:->{32}{33}{34}{35}{36}{37}{38}{39}{40}{41}{42}{43}{44}{45}{46}{47}{48}
macro:->{1}{2}{3}{4}{5}{6}{7}
macro:->{3}{4}{5}{6}{7}{8}{9}
macro:->{1}{2}{3}{4}{5}{6}{7}
```

155

```
macro:->3456789
```

14.7. \xintKeepUnbraced

Same as \xintKeep but no brace pairs are added around the kept items from the head of the list in the case x>0: each such item will lose one level of braces. Thus, to remove braces from all items of the list, one can use \xintKeepUnbraced with its first argument larger than the length of the list; the same is obtained from \xintListWithSep{}{⟨list⟩}. But the new list will then have generally many more items than the original ones, corresponding to the unbraced original items.

For x<0 the macro is no different from \xintKeep. Hence the name is a bit misleading because brace removal will happen only if x>0.

\xintKeepUnbracedNoExpand does the same without first *f-expan*ding its list argument.

```
\fdef\test {\xintKeepUnbraced {10}{\xintSeq {1}{100}}}\meaning\test\par
\noindent\fdef\test {\xintKeepUnbraced {7}{{1}{2}{3}{4}{5}{6}{7}{8}{9}}}\meaning\test\par
\noindent\fdef\test {\xintKeepUnbraced {-7}{{1}{2}{3}{4}{5}{6}{7}{8}{9}}}\meaning\test\par
\noindent\fdef\test {\xintKeepUnbraced {7}{123456789}}\meaning\test\par
\noindent\fdef\test {\xintKeepUnbraced {-7}{123456789}}\meaning\test\par
```

```
macro:->12345678910
macro:->1234567
macro:->{3}{4}{5}{6}{7}{8}{9}
macro:->1234567
macro:->3456789
```

14.8. \xintTrim

$\overset{\text{num}}{x}$ *f* ⋆ \xintTrim{x}{⟨list⟩} expands the list argument and gobbles its first x elements.
- if x>0, the first x items from L are gobbled. The remaining items are not modified.
- if x>=length(L), the returned list is empty.
- if x=0 the original list is returned (with no added braces.)
- if x<0 the last |x| items of the list are removed. *The head items end up braced in the output.* Use \xintTrimUnbraced if this is not desired.
- if x<=-length(L) the output is empty.

\xintTrimNoExpand does the same without first *f-expan*ding its list argument.

```
\fdef\test {\xintTrim {17}{\xintTrim {-69}{\xintSeq {1}{100}}}}\meaning\test\par
\noindent\fdef\test {\xintTrim {7}{{1}{2}{3}{4}{5}{6}{7}{8}{9}}}\meaning\test\par
\noindent\fdef\test {\xintTrim {-7}{{1}{2}{3}{4}{5}{6}{7}{8}{9}}}\meaning\test\par
\noindent\fdef\test {\xintTrim {7}{123456789}}\meaning\test\par
\noindent\fdef\test {\xintTrim {-7}{123456789}}\meaning\test\par
```

```
macro:->{18}{19}{20}{21}{22}{23}{24}{25}{26}{27}{28}{29}{30}{31}
macro:->{8}{9}
macro:->{1}{2}
macro:->89
macro:->{1}{2}
```

14.9. \xintTrimUnbraced

Same as \xintTrim but in case of a negative x (cutting items from the tail), the kept items from the head are not enclosed in brace pairs. They will lose one level of braces. The name is a bit misleading because when x>0 there is no brace-stripping done on the kept items, because the macro works simply by gobbling the head ones.

\xintTrimUnbracedNoExpand does the same without first *f-expan*ding its list argument.

```
\fdef\test {\xintTrimUnbraced {-90}{\xintSeq {1}{100}}}\meaning\test\par
\noindent\fdef\test {\xintTrimUnbraced {7}{{1}{2}{3}{4}{5}{6}{7}{8}{9}}}\meaning\test\par
\noindent\fdef\test {\xintTrimUnbraced {-7}{{1}{2}{3}{4}{5}{6}{7}{8}{9}}}\meaning\test\par
```

```
\noindent\fdef\test {\xintTrimUnbraced {7}{123456789}}\meaning\test\par
\noindent\fdef\test {\xintTrimUnbraced {-7}{123456789}}\meaning\test\par
```
macro:->12345678910
macro:->{8}{9}
macro:->12
macro:->89
macro:->12

14.10. \xintListWithSep

n f ⋆ \xintListWithSep{⟨*sep*⟩}{⟨*list*⟩} inserts the separator ⟨*sep*⟩ in-between all items of the given list of braced items (or individual tokens). The items are fetched as does TeX with undelimited macro arguments, thus they end up unbraced in output. If the ⟨*list*⟩ is only one (or multiple) space tokens, the output is empty.

The list argument ⟨*list*⟩ gets *f-expan*ded first (thus if it is a macro whose contents are braced items, the first opening brace stops the expansion, and it is as if the macro had been expanded once.) The separator ⟨*sep*⟩ is not pre-expanded, it ends up as is in the output (if the ⟨*list*⟩ contained at least two items.)

n n ⋆ The variant \xintListWithSepNoExpand does the same job without the initial expansion of the ⟨*list*⟩ argument.

```
\edef\foo{\xintListWithSep{, }{123456789{10}{11}{12}}}\meaning\foo\newline
\edef\foo{\xintListWithSep{:}{\xintiiFac{20}}}\meaning\foo\newline
\oodef\FOO{\xintListWithSepNoExpand{\FOO}{\bat\baz\biz\buz}}\meaning\FOO\newline
% a braced item or a space stops the f-expansion:
\oodef\foo{\xintListWithSep{\FOO}{{\bat}\baz\biz\buz}}\meaning\foo\newline
\oodef\foo{\xintListWithSep{\FOO}{ \bat\baz\biz\buz}}\meaning\foo\par
```
macro:->1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
macro:->2:4:3:2:9:0:2:0:0:8:1:7:6:6:4:0:0:0:0
macro:->\bat \FOO \baz \FOO \biz \FOO \buz
macro:->\bat \FOO \baz \FOO \biz \FOO \buz
macro:->\bat \FOO \baz \FOO \biz \FOO \buz

14.11. \xintApply

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

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

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

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

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

157

14.12. \xintApplyUnbraced

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

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

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

```
\def\macro #1{\expandafter\def\csname myself#1\endcsname {#1}}
\xintApplyUnbraced\macro{{elta}{eltb}{eltc}}
\begin{enumerate}[nosep,label=(\arabic{*})]
\item \meaning\myselfelta
\item \meaning\myselfeltb
\item \meaning\myselfeltc
\end{enumerate}
```

(1) macro:->elta
(2) macro:->eltb
(3) macro:->eltc

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

14.13. \xintSeq

[num/x] num num ⋆ \xintSeq[d]{x}{y} generates expandably {x}{x+d}... up to and possibly including {y} if d>0 or down to and including {y} if d<0. Naturally {y} is omitted if y-x is not a multiple of d. If d=0 the macro returns {x}. If y-x and d have opposite signs, the macro returns nothing. If the optional argument d is omitted it is taken to be the sign of y-x. Hence \xintSeq {1}{0} is not empty but {1}{0}. But \xintSeq [1]{1}{0} is empty.

  The arguments x and y are expanded inside a \numexpr so they may be count registers or a 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

☞ { When the macro is used without the optional argument d, it can only generate up to about 5000 numbers, the precise value depends upon some TeX memory parameter (input save stack).

  With the optional argument d the macro proceeds differently (but less efficiently) and does not stress the input save stack.

14.14. \xintloop, \xintbreakloop, \xintbreakloopanddo, \xintloopskiptonext

☆ \xintloop⟨*stuff*⟩\if<test>...\repeat is an expandable loop compatible with nesting. However to break out of the loop one almost always need some un-expandable step. The cousin \xintiloop is \xintloop with an embedded expandable mechanism allowing to exit from the loop. The iterated macros may contain \par tokens or empty lines.

  If a sub-loop is to be used all the material from the start of the main loop and up to the end of the entire subloop should be braced; these braces will be removed and do not create a group. The simplest to allow the nesting of one or more sub-loops is to brace everything between \xintloop and \repeat, being careful not to leave a space between the closing brace and \repeat.

  As this loop and \xintiloop will primarily be of interest to experienced TeX macro programmers, my description will assume that the user is knowledgeable enough. Some examples in this document will be perhaps more illustrative than my attemps at explanation of use.

  One can abort the loop with \xintbreakloop; this should not be used inside the final test, and one should expand the \fi from the corresponding test before. One has also \xintbreakloopanddo

158

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.[56]

```
\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 }%
```

---

[56] 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 some code online posted from November 11, 2013.

```
\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
\MatrixMultiplication\C\D\E \MatrixMultiplication\C\E\F
\begin{multicols}2
  \[\begin{pmatrix}
    \A11&\A12&\A13&\A14\\
    \A21&\A22&\A23&\A24\\
    \A31&\A32&\A33&\A34
  \end{pmatrix}
  \times
  \begin{pmatrix}
    \B11&\B12&\B13\\
    \B21&\B22&\B23\\
    \B31&\B32&\B33\\
    \B41&\B42&\B43
  \end{pmatrix}
  =
  \begin{pmatrix}
    \C11&\C12&\C13\\
    \C21&\C22&\C23\\
    \C31&\C32&\C33
  \end{pmatrix}\]
  \[\begin{pmatrix}
    \C11&\C12&\C13\\
    \C21&\C22&\C23\\
    \C31&\C32&\C33
  \end{pmatrix}^2 = \begin{pmatrix}
    \D11&\D12&\D13\\
    \D21&\D22&\D23\\
    \D31&\D32&\D33
  \end{pmatrix}\]
  \[\begin{pmatrix}
    \C11&\C12&\C13\\
    \C21&\C22&\C23\\
    \C31&\C32&\C33
  \end{pmatrix}^3 = \begin{pmatrix}
    \E11&\E12&\E13\\
    \E21&\E22&\E23\\
    \E31&\E32&\E33
  \end{pmatrix}\]
  \[\begin{pmatrix}
    \C11&\C12&\C13\\
    \C21&\C22&\C23\\
    \C31&\C32&\C33
  \end{pmatrix}^4 = \begin{pmatrix}
    \F11&\F12&\F13\\
    \F21&\F22&\F23\\
    \F31&\F32&\F33
  \end{pmatrix}\]
```

```
\end{multicols}
```

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

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

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

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

14.15. \xintiloop, \xintiloopindex, \xintouteriloopindex, \xintbreakiloop, \xintbreakiloopanddo, \xintiloopskiptonext, \xintiloopskipandredo

☆ \xintiloop[start+delta]⟨*stuff*⟩\if<test> ... \repeat is a completely expandable nestable loop. complete expandability depends naturally on the actual iterated contents, and complete expansion will not be achievable under a sole *f-expan*sion, as is indicated by the hollow star in the margin; thus the loop can be used inside an \edef but not inside arguments to the package macros. It can be used inside an \xintexpr..\relax. The [start+delta] is mandatory, not optional.

This loop benefits via \xintiloopindex to (a limited access to) the integer index of the iteration. The starting value start (which may be a \count) and increment delta (*id.*) are mandatory arguments. A space after the closing square bracket is not significant, it will be ignored. Spaces inside the square brackets will also be ignored as the two arguments are first given to a \numexpr ...\relax. Empty lines and explicit \par tokens are accepted.

As with \xintloop, this tool will mostly be of interest to advanced users. For nesting, one puts inside braces all the material from the start (immediately after [start+delta]) and up to and inclusive of the inner loop, these braces will be removed and do not create a loop. In case of nesting, \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 \xintbreakil oopanddo 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 \xintiloopi ndex 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, `\xintiloopskip-andredo` 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:

```
\begingroup
\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\endgroup
```
macro:->10007, 10009, 10037, 10039, 10061, 10067, 10069, 10079, 10091, 10093, 10099, 10103, 10111, 10133, 10139, 10141, 10151, 10159, 10163, 10169, 10177, 10181, 10193, 10211, 10223, 10243, 10247, 10253, 10259, 10267, 10271, 10273, 10289, 10301, 10303, 10313, 10321, 10331, 10333, 10337, 10343, 10357, 10369, 10391, 10399, 10427, 10429, 10433, 10453, 10457, 10459, 10463, 10477, 10487, 10499, 10501, 10513, 10529, 10531, 10559, 10567, 10589, 10597, 10601, 10607, 10613, 10627, 10631, 10639, 10651, 10657, 10663, 10667, 10687, 10691, 10709, 10711, 10723, 10729, 10733, 10739, 10753, 10771, 10781, 10789, 10799, 10831, 10837, 10847, 10853, 10859, 10861, 10867, 10883, 10889, 10891, 10903, 10909, 10937, 10939, 10949, 10957, 10973, 10979, 10987, 10993, and we should have taken some steps to not have a trailing comma, but the point was to show that one can do that in an `\edef`! See also subsection 16.3 which extracts from this code its way of testing primality.

Let us create an alignment where each row will contain all divisors of its first entry. Here is the output, thus obtained without any count register:

```
\begin{multicols}2
\tabskip1ex \normalcolor
\halign{&\hfil#\hfil\cr
    \xintiloop [1+1]
    {\expandafter\bfseries\xintiloopindex &
     \xintiloop [1+1]
     \ifnum\xintouteriloopindex=\numexpr
           (\xintouteriloopindex/\xintiloopindex)*\xintiloopindex\relax
     \xintiloopindex&\fi
     \ifnum\xintiloopindex<\xintouteriloopindex\space % CRUCIAL \space HERE
     \repeat \cr }%
    \ifnum\xintiloopindex<30
    \repeat
}
\end{multicols}
```

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

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:

```
\tabskip1ex
\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 }
```

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

14.16. `\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}.
```
0, 3, 4, 8, 9, 14, 23, 25, 31, 36, 39. The first argument `\macro` does not have to be an expandable macro.

   `\xintApplyInline` submits its second, token list parameter to an *f-expansion*. Then, each *unbraced* item will also be *f-expanded*. This provides an easy way to insert one list inside another. *Braced* items are not expanded. Spaces in-between items are gobbled (as well as those at the start or the end of the list), but not the spaces *inside* the braced items.

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

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

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

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

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

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

```
\begin{figure*}[ht!]
  \centering\phantomsection\label{float}
  \def\Row #1{#1:\xintApplyInline {\Item {#1}}{0123456789}\\ }%
  \def\Item #1#2{&\xintiiPow {#1}{#2}}%
  \centeredline {\begin{tabular}{ccccccccccc} &0&1&2&3&4&5&6&7&8&9\\ \hline
      \xintApplyInline \Row {0123456789}
  \end{tabular}}
\end{figure*}
```

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

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

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

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

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

164

But see `\xintFor`.

14.17. `\xintFor`, `\xintFor*`

on `\xintFor` is a new kind of for loop.[57] Rather than using macros for encapsulating list items, its behaviour is like a macro with parameters: `#1`, `#2`, . . . , `#9` are used to represent the items for up to nine levels of nested loops. Here is an example:

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

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.

TEXnical notes:

- The `#1` is replaced in the iterated-over text exactly as in general TEX macros or LATEX commands. This spares the user quite a few `\expandafter`'s or other tricks needed with loops which have the values encapsulated in macros, like LATEX's `\@for` and `\@tfor`.

- `\xintFor` (and `\xintFor*`) isn't purely expandable: one can not use it inside an `\edef`. But it may be used, as will be shown in examples, in some contexts such as LATEX's `tabular` which are usually hostile to non-expandable loops.

- `\xintFor` (and `\xintFor*`) does some assignments prior to executing each iteration of the replacement text, but it acts purely expandably after the last iteration, hence if for example the replacement text ends with a `\\`, the loop can be used insided a tabular and be followed by a `\hline` without creating the dreaded ``Misplaced `\noalign`'' error.

- As stated in previous item the first iteration follows some non-expandable internal dealings. This means for example that in LATEX, one can not inject a `\multicolumn` in the first iteration. Sometimes one way work around this by injecting father `&\multicolumn` or `\\ \multicolumn`.

- It does not create groups.

- It makes no global assignments.

- The iterated replacement text may close a group which was opened even before the start of the loop (typical example being with `&` in alignments).

```
\begin{tabular}{rccccc}
    \hline
    \xintFor #1 in {A, B, C} \do {%
      #1:\xintFor #2 in {a, b, c, d, e} \do {&($ #2 \to #1 $)}\\ }%
    \hline
\end{tabular}
```

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

---

[57] first introduced with xint 1.09c of 2013/10/09.

> - There is no facility provided which would give access to a count of the number of itera-
>   tions as it is technically not easy to do so it in a way working with nested loops while
>   maintaining the ``expandable after done'' property; something in the spirit of \xint-
>   iloopindex is possible but this approach would bring its own limitations and complica-
>   tions. Hence the user is invited to update her own count or LaTeX counter or macro at each
>   iteration, if needed.
>
> - A \macro whose definition uses internally an \xintFor loop may be used inside another
>   \xintFor loop even if the two loops both use the same macro parameter. The loop definition
>   inside \macro must use ## as is the general rule for definitions done inside macros.
>
> - \xintFor is for comma separated values and \xintFor* for lists of braced items; their
>   respective expansion policies differ. They are described later.

Regarding \xintFor:
- the spaces between the various declarative elements are all optional,
- in the list of comma separated values, spaces around the commas or at the start and end are
  ignored,
- if an item must contain itself its own commas, then it should be braced, and the braces will
  be removed before feeding the iterated-over text,
- the list may be a macro, it is expanded only once,
- items are not pre-expanded. The first item should be braced or start with a space if the list
  is explicit and the item should not be pre-expanded,
- empty items give empty #1's in the replacement text, they are not skipped,
- an empty list executes once the replacement text with an empty parameter value,
- the list, if not a macro, must be braced.

fn Regarding \xintFor*:
- it handles lists of braced items (or naked tokens),
- it *f*-expands the list,
- and more generally it *f*-expands each naked token encountered before assigning the #1 values
  (gobbling spaces in the process); this makes it easy to simulate concatenation of multiple
  lists\x, \y: if \x expands to {1}{2}{3} and \y expands to {4}{5}{6} then {\x\y} as argument to
  \xintFor* has the same effect as {{1}{2}{3}{4}{5}{6}}.
      For a further illustration see the use of \xintFor* at the end of subsection 15.18.
- spaces at the start, end, or in-between items are gobbled (but naturally not the spaces inside
  braced items),
- except if the list argument is a macro (with no parameters), it must be braced. ,
- an empty list leads to an empty result.

The macro \xintSeq which generates arithmetic sequences is to be used with \xintFor* as its
output consists of successive braced numbers (given as digit tokens).

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

stuff with -7
stuff with -5
stuff with -3
stuff with -1
stuff with 1

When nesting \xintFor* loops, using \xintSeq in the inner loops is inefficient, as the arith-
metic 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 .. }
```

166

This is a general remark applying for any nesting of loops, one should avoid recreating the inner lists of arguments at each iteration of the outer loop.

When the loop is defined inside a macro for later execution the # characters must be doubled.[58] For example:

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

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

The iterated macros as well as the list items are allowed to contain explicit \par tokens.

14.18. \xintifForFirst, \xintifForLast

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

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

> Pay attention to these implementation features:
> - *if an inner \xintFor loop is positioned before the \xintifForFirst or \xintifForLast of the outer loop it will contaminate their settings. This applies also naturally if the inner loop arises from the expansion of some macro located before the outer conditionals.*
>   *One fix is to make sure that the outer conditionals are expanded before the inner loop is executed, e.g. this will be the case if the inner loop is located inside one of the branches of the conditional.*
>   *Another approach is to enclose, if feasible, the inner loop in a group of its own.*
> - *if the replacement text closes a group (e.g. from a & inside an alignment), the conditionals will lose their ascribed meanings and end up possibly undefined, depending whether there is some outer loop whose execution started before the opening of the group.*
>   *The fix is to arrange things so that the conditionals are expanded before TeX encounters the closing-group token.*

14.19. \xintBreakFor, \xintBreakForAndDo

One may immediately terminate an \xintFor or \xintFor* loop with \xintBreakFor.

> As it acts by clearing up all the rest of the replacement text when encountered, it will not work from inside some \if...\fi without suitable \expandafter or swapping technique.
> Also it can't be used from inside braces as from there it can't see the end of the replacement text.

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

---

[58] 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.

14.20. \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!*)[59], 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 macros 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.

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).

```
\begingroup\small
\noindent\parbox{\dimexpr\linewidth-3em}{\color[named]{OrangeRed}%
\xintFor #1 in {\xintrationals [10/21+1/21]} \do
{#1=\xintifInt {#1}
    {\textcolor{blue}{\xintTrunc{10}{#1}}}
    {\xintTrunc{10}{#1}}% display in blue if an integer
    \xintifGt {#1}{1.123}{\xintBreakFor}{, }%
}}
\endgroup\smallskip
```

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

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

---

[59] 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.

in scientific notation with exponents in the hundreds, as they will get converted into as many zeroes.

```
\noindent\parbox{\dimexpr.7\linewidth}{\raggedright
\xintFor #1 in {\xintrationals [0.000+0.125]} \do
{\edef\tmp{\xintTrunc{3}{#1}}%
 \xintifInt {#1}
    {\textcolor{blue}{\tmp}}
    {\tmp}%
    \xintifGt {#1}{2}{\xintBreakFor}{, }%
  }}\smallskip
```

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

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

14.21. \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.

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

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

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

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

\xintForpair must be followed by either #1#2, #2#3, #3#4, ..., or #8#9 with #1 usable as an alias for #1#2, #2 as alias for #2#3, etc ... and similarly for \xintForthree (using #1#2#3 or simply #1, #2#3#4 or simply #2, ...) and \xintForfour (with #1#2#3#4 etc...).

Nesting works as long as the macro parameters are distinct among #1, #2, ..., #9. A macro which expands to an \xintFor or a \xintFor(pair,three,four) can be used in another one with no constraint about using distinct macro parameters.

\par tokens are accepted in both the comma separated list and the replacement text.

14.22. \xintAssign

\xintAssign⟨*braced things*⟩\to⟨*as many cs as they are things*⟩ defines (without checking if something gets overwritten) the control sequences on the right of \to to expand to the successive tokens or braced items located to the left of \to. \xintAssign is not an expandable macro.

*f-expan*sion is first applied to the material in front of \xintAssign which is fetched as one argument if it is braced. Then the expansion of this argument is examined and successive items are assigned to the macros following \to. There must be exactly as many macros as items. No check is done. The macro assignments are done with removal of one level of brace pairs from each item.

After the initial *f-expan*sion, each assigned (brace-stripped) item will be expanded according to the setting of the optional parameter.

For example `\xintAssign [e]...` means that all assignments are done using `\edef`. With `[f]` the assignments will be made using `\fdef`. The default is simply to make the definitions with `\def`, corresponding to an empty optional paramter `[]`. Possibilities for the optional parameter are: `[]` , `[g]`, `[e]`, `[x]`, `[o]`, `[go]`, `[oo]`, `[goo]`, `[f]`, `[gf]`. For example `[oo]` means a double expansion.

```
\xintAssign \xintiiDivision{1000000000000}{133333333}\to\Q\R
\meaning\Q\newline
\meaning\R\newline
\xintAssign {{\xintiiDivision{1000000000000}{133333333}}}\to\X
\meaning\X\newline
\xintAssign [oo]{{\xintiiDivision{1000000000000}{133333333}}}\to\X
\meaning\X\newline
\xintAssign \xintiiPow{7}{13}\to\SevenToThePowerThirteen
\meaning\SevenToThePowerThirteen\par
```
macro:->7500
macro:->2500
macro:->\xintiiDivision {1000000000000}{133333333}
macro:->{7500}{2500}
macro:->96889010407

Two special cases:
- if after this initial expansion no brace is found immediately after \xintAssign, it is assumed that there is only one control sequence following `\to`, and this control sequence is then de- fined via `\def` (or what is set-up by the optional parameter) to expand to the material between \xintAssign and `\to`.
- if the material between \xintAssign and `\to` is enclosed in two brace pairs, the first brace pair is removed, then the *f*-expansion is immediately stopped by the inner brace pair, hence \xintAssign now finds a unique item and thus defines only a single macro to be this item, which is now stripped of the second pair of braces.

Note: prior to release 1.09j, `\xintAssign` did an `\edef` by default for each item assignment but it now does `\def` corresponding to no or empty optional parameter.

It is allowed for the successive braced items to be separated by spaces. They are removed during the assignments. But if a single macro is defined (which happens if the argument after *f*-expansion does not start with a brace), naturally the scooped up material has all intervening spaces, as it is considered a single item. But an upfront initial space will have been absorbed by *f*-expansion.

```
\def\X{ {a}  {b} {c}   {d} }\def\Y { u {a}  {b} {c}    {d} }
\xintAssign\X\to\A\B\C\D
\xintAssign\Y\to\Z
\meaning\A, \meaning\B, \meaning\C, \meaning\D+++\newline
\meaning\Z+++\par
```
macro:->a, macro:->b, macro:->c, macro:->d+++
macro:->u {a} {b} {c} {d} +++

As usual successive space characters in input make for a single TeX space token.

14.23. \xintAssignArray

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

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

\xintAssignArray admits an optional parameter, for example \xintAssignArray [e] means that the definitions of the macros will be made with \edef. The empty optional parameter (default) means that definitions are done with \def. Other possibilities: [], [o], [oo], [f]. Contrarily to \xintAssign one can not use the g here to make the definitions global. For this, one should rather do \xintAssignArray within a group starting with \globaldefs 1.

14.24. \xintDigitsOf

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

    \xintDigitsOf\xintiiPow {7}{500}\to\digits

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

14.25. \xintRelaxArray

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

15. Macros of the xintexpr package

The xintexpr package was first released with version 1.07 (2013/05/25) of the xint bundle. It was substantially enhanced with release 1.1 from 2014/10/28.

The 1.4 release from 2020/01/31 maintains the same general architecture but needed adapting all the code base for the switch from \csname to \expanded techniques. On this occasion the mechanism for defining functions was substantially strengthened. The parser core mechanisms were improved too.

The package loads automatically xintfrac and xinttools.

This section should be trimmed to contain only information not already covered in section 2.

15.1. 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.

Changed at 1.4!   An \xintexpr...\relax *must* end in a \relax (which will be absorbed). Contrarily to a \numexpr expression, it is printable as is without a prefix \the or \number (don't use them with \xintexpr this will raise an error).

But one can use \xintthe prefix if one does need the explicit digits and other characters as in the final typesetted result.

As an alternative and equivalent syntax to

  \xintexpr round(<expression>, D)\relax

there is

  \xintiexpr [D] <expression> \relax

The parameter D must be zero or positive.[60] Perhaps some future version will give a meaning to using a negative D.[61]

- the expression may contain arbitrarily many levels of nested parenthesized sub-expressions,

- the expression may contain explicitely or from a macro expansion a sub-expression \xintexpr. ..\relax, which itself may contain a sub-expressions etc...

- to let sub-contents evaluate as a sub-unit it should thus be either

---

[60] D=0 corresponds to using round(<expression>) not round(<expression>,0) which would leave a trailing dot. Same for trunc. There is also function float for floating point rounding to \xinttheDigits or the given number of significant digits as second argument.   [61] Thanks to KT for this suggestion. Sorry for the delay in implementing it... matter of formatting the output and corresponding choice of user interface are still in need of some additional thinking.

1. parenthesized,

2. or a sub-expression `\xintexpr...\relax`.

- to use an expression as argument to macros from `xintfrac`, or more generally to macros which expand their arguments, one must use the `\xinttheexpr...\relax` or `\xintthe\xintexpr...\relax` forms.

- one should not use `\xintthe\xintexpr...\relax` as a sub-constituent of another expression but only the `\xintexpr...\relax` form which is more efficient in this context.

- each `xintexpr`ession, whether prefixed or not with `\xintthe`, is completely expandable and obtains its result in two expansion steps.

The information now following is possibly in need of updates.

- An expression is built the standard way with opening and closing parentheses, infix operators, and (big) numbers, with possibly a fractional part, and/or scientific notation (except for `\xintiiexpr` which only admits big integers). All variants work with comma separated expressions. On output each comma will be followed by a space. A decimal number must have digits either before or after the decimal mark.

- As everything gets expanded, the characters ., +, -, *, /, ^, !, &, |, ?, :, <, >, =, (,), ",], [, @ and the comma , should not (if used in the expression) be active. For example, the French language in `Babel` system, for pdfLaTeX, activates !, ?, ; and :. Turn off the activity before expressions using such characters.
  Alternatively the macro `\xintexprSafeCatcodes` resets all characters potentially needed by `\xintexpr` to their standard catcodes and `\xintexprRestoreCatcodes` restores the former status.

- Count registers and `\numexpr`-essions are accepted (LaTeX's counters can be inserted using `\value`) natively without `\the` or `\number` as prefix. Also dimen registers and control sequences, skip registers and control sequences (LaTeX's lengths), `\dimexpr`-essions, `\glueexpr`-essions are automatically unpacked using `\number`, discarding the stretch and shrink components and giving the dimension value in `sp` units (1/65536th of a TeX point). Furthermore, tacit multiplication is implied, when the (count or dimen or glue) register or variable, or the (`\numexpr` or `\dimexpr` or `\glueexpr`) expression is immediately prefixed by a (decimal) number. See subsection 2.11 for the complete rules of tacit multiplication.

☞ {
- With a macro `\x` defined like this:
  ```
  \def\x {\xintexpr \a + \b \relax} or \edef\x {\xintexpr \a+\b\relax}
  ```
  one may then do `\xintthe\x`, either for printing the result on the page or to use it in some other macros expanding their arguments. The `\edef` does the computation immediately but keeps it in a protected form. Naturally, the `\edef` is only possible if `\a` and `\b` are already defined. With both approaches the `\x` can be inserted in other expressions, as for example (assuming naturally as we use an `\edef` that in the `yet-to-be computed' case the `\a` and `\b` now have some suitable meaning):
  ```
  \edef\y {\xintexpr \x^3\relax}
  ```

- There is also `\xintboolexpr ... \relax` and `\xinttheboolexpr ... \relax`.

- See also `\xintifboolexpr` (subsection 15.14) and the `bool`() and `togl`() functions in section 2. Here is an example. Well in fact the example ended up using only `\xintboolexpr` so it was modified to use `\xintifboolexpr`.
  ```
  \xintdeffunc A(p,q,r) =  p && (q || r) ;
  \xintdeffunc B(p,q,r) =  p || (q && r) ;
  \xintdeffunc C(p,q,r) =  xor(p, q, r)  ;

  \centeredline{\normalcolor
  \begin{tabular}{ccrclcl}
  ```

```
        \xintFor* #1 in  {{False}{True}} \do {%
        \xintFor* #2 in  {{False}{True}} \do {%
        \xintFor* #3 in  {{False}{True}} \do {%
    #1 &AND &(#2 &OR  &#3)&is&\textcolor[named]{OrangeRed}
                        {\xintifboolexpr{A(#1,#2,#3)}{true}{false}}\\
    #1 &OR  &(#2 &AND &#3)&is&\textcolor[named]{OrangeRed}
                        {\xintifboolexpr{B(#1,#2,#3)}{yes}{no}}\\
    #1 &XOR & #2 &XOR &#3 &is&\textcolor[named]{OrangeRed}
                        {\xintifboolexpr{C(#1,#2,#3)}{oui}{non}}\\
}}}
\end{tabular}%
}
```

False	AND	(False	OR	False)	is	false
False	OR	(False	AND	False)	is	no
False	XOR	False	XOR	False	is	non
False	AND	(False	OR	True)	is	false
False	OR	(False	AND	True)	is	no
False	XOR	False	XOR	True	is	oui
False	AND	(True	OR	False)	is	false
False	OR	(True	AND	False)	is	no
False	XOR	True	XOR	False	is	oui
False	AND	(True	OR	True)	is	false
False	OR	(True	AND	True)	is	yes
False	XOR	True	XOR	True	is	non
True	AND	(False	OR	False)	is	false
True	OR	(False	AND	False)	is	yes
True	XOR	False	XOR	False	is	oui
True	AND	(False	OR	True)	is	true
True	OR	(False	AND	True)	is	yes
True	XOR	False	XOR	True	is	non
True	AND	(True	OR	False)	is	true
True	OR	(True	AND	False)	is	yes
True	XOR	True	XOR	False	is	non
True	AND	(True	OR	True)	is	true
True	OR	(True	AND	True)	is	yes
True	XOR	True	XOR	True	is	oui

- See also `\xintifsgnexpr`.
- There is `\xintfloatexpr ... \relax` where the algebra is done in floating point approximation (also for each intermediate result). Use the syntax `\xintDigits:=N\relax` to set the precision. Default: 16 digits.
  `\xintthefloatexpr 2^100000\relax`: 9.990020930143845e30102

The square-root operation can be used in `\xintexpr`, it is computed as a float with the precision set by `\xintDigits` or by the optional second argument:

```
\xinttheexpr sqrt(2,60)\relax\newline
Here the [60] is to avoid truncation to |\xinttheDigits| of precision on output.\newline
\printnumber{\xintthefloatexpr [60] sqrt(2,60)\relax}
```
14142135623730950488016887242096980785696718753769480731766e-59
Here the [60] is to avoid truncation to \xinttheDigits of precision on output.
1.4142135623730950488016887242096980785696718753769480731766

Floats are quickly indispensable when using the power function, as exact results will easily have hundreds, even thousands of digits.

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

Only integer and (in `\xintfloatexpr...\relax`) half-integer exponents are allowed.

174

- if one uses *macros* within `\xintexpr..\relax` one should obviously take into account that the parser will *not* see the macro arguments, hence one cannot use the syntax there, except if the arguments are themselves wrapped as `\xinttheexpr...\relax` and assuming the macro `f-expan`ds these arguments.

15.2. `\numexpr` or `\dimexpr` expressions, count and dimension registers and variables

Count registers, count control sequences, dimen registers, dimen control sequences (like `\parind`⟩`ent`), skips and skip control sequences, `\numexpr`, `\dimexpr`, `\glueexpr`, `\fontdimen` can be inserted directly, they will be unpacked using `\number` which gives the internal value in terms of scaled points for the dimensional variables: $1\,\mathrm{pt} = 65536\,\mathrm{sp}$ (stretch and shrink components are thus discarded).

Tacit multiplication (see subsection 2.11) is implied, when a number or decimal number prefixes such a register or control sequence. LaTeX lengths are skip control sequences and LaTeX counters should be inserted using `\value`.

Release 1.2 of the `\xintexpr` parser also recognizes and prefixes with `\number` the `\ht`, `\dp`, and `\wd` TeX primitives as well as the `\fontcharht`, `\fontcharwd`, `\fontchardp` and `\fontcharic` ε-TeX primitives.

In the case of numbered registers like `\count255` or `\dimen0` (or `\ht0`), the resulting digits will be re-parsed, so for example `\count255 0` is like 100 if `\the\count255` would give 10. The same happens with inputs such as `\fontdimen6\font`. And `\numexpr 35+52\relax` will be exactly as if 87 as been encountered by the parser, thus more digits may follow: `\numexpr 35+52\relax 000` is like 87000. If a new `\numexpr` follows, it is treated as what would happen when `\xintexpr` scans a number and finds a non-digit: it does a tacit multiplication.

```
\xinttheexpr \numexpr 351+877\relax\numexpr 1000-125\relax\relax{} is the same
as \xinttheexpr 1228*875\relax.
```
1074500 is the same as 1074500.

Control sequences however (such as `\parindent`) are picked up as a whole by `\xintexpr`, and the numbers they define cannot be extended extra digits, a syntax error is raised if the parser finds digits rather than a legal operation after such a control sequence.

A token list variable must be prefixed by `\the`, it will not be unpacked automatically (the parser will actually try `\number`, and thus fail). Do not use `\the` but only `\number` with a dimen or skip, as the `\xintexpr` parser doesn't understand pt and its presence is a syntax error. To use a dimension expressed in terms of points or other TeX recognized units, incorporate it in `\dimexpr...\relax`.

Regarding how dimensional expressions are converted by TeX into scaled points see also subsection 5.7.

15.3. Catcodes and spaces

The main problems are caused by active characters, because `\xintexpr` et al. expand forward whatever comes from token stream; they apply `\string` only in a second step. For example the catcode of & from && Boolean disjunction is not really important as long as it is not active, or comment, or escape... or brace... or ignored... in brief, as long as it is reasonable, and in particular whether @ is of catcode letter or other does not matter.

It is always possible to insert manually the `\string` in the expression before a problematic (but reasonable) character catcode, or even to use `\detokenize` for a big chunk.

15.3.1. `\xintexprSafeCatcodes`

For an even more radical way, there is `\xintexprSafeCatcodes` which sets the catcodes of many characters to safe values. This is a non-expandable step as it changes catcodes.

175

\xintdefvar, \xintdeffunc, et al., execute it before fetching their semi-colon delimited arguments, so they can be used (also in the document body) for example with Babel+French (which makes the semi-colon active in the (LaTeX) document body). This applies also to \xintNewExpr.

But, if used in the body of macro definitions problems may arise from the catcode regime at that location. This applies in particular to the semi-colon as used by \xintdeffunc, \xintdefvar and variants as delimiter. Thus make sure the semi-colon has its normal catcode when issueing \xintdeffunc inside some macro definition.

\xintdeffunc is more lenient than \xintdefvar regarding catcodes of characters in expression bodies as it does some \scantokens which will reset compatible catcodes. And also, characters inside the expression may usually be prefixed with \string; but some aspects of the parsing use delimited macros which need the comma, equality sign and closing parenthesis to have standard catcodes.

Even if used in a context where catcodes are already set, \xintdeffunc, \xintdefvar and variants ignore completely the colon in := so it can have any (reasonable) catcode. Moreover it is optional.

The semi-colon in the syntax of \xintDigits is no real problem either (cf. \xintDigits documentation).

☞ {
It is important to ALWAYS shortly let \xintexprSafeCatcodes be followed by \xintexprRestoreCatcodes. If one uses twice \xintexprSafeCatcodes then the next \xintexprRestoreCatcodes will restore the ancient catcode regime at time of the first one.
}

15.3.2. \xintexprRestoreCatcodes

Restores the catcodes to the earlier state. More precisely, \xintexprSafeCatcodes sets a toggle (with local scope). If the toggle is set already it does not restore the current catcodes. The next \xintexprRestoreCatcodes unsets the toggle. So, in case of nesting, the catcodes are restored to what they were when the *first* un-paired \xintexprSafeCatcodes got executed.

Spaces inside an \xinttheexpr...\relax should mostly be innocuous (except inside macro arguments).

\xintexpr and \xinttheexpr are for the most part agnostic regarding catcodes: (unbraced) digits, binary operators, minus and plus signs as prefixes, dot as decimal mark, parentheses, may be indifferently of catcode letter or other or subscript or superscript, ..., it doesn't matter.[62]

The characters +, -, *, /, ^, !, &, |, ?, :, <, >, =, (,), ", [,], ;, the dot and the comma should not be active if in the expression, as everything is expanded along the way. If one of them is active, it should be prefixed with \string.

The exclamation mark ! should have its standard catcode: with catcode letter it is used internally and hence will confuse the parsers if it comes from the expression.

Digits, slash, square brackets, minus sign, in the output from an \xinttheexpr are all of catcode 12. For \xintthefloatexpr the `e' in the output has its standard catcode ``letter''.

A macro with arguments will expand and grab its arguments before the parser may get a chance to see them, so the situation with catcodes and spaces is not the same within such macro arguments.

15.4. Expandability, \xintexpro

As is the case with all other package macros \xintexpr *f-expan*ds (in two steps) to its final (somewhat protected) result; and \xinttheexpr *f-expan*ds (in two steps) to the chain of digits (and possibly minus sign -, decimal mark ., fraction slash /, scientific e, square brackets [,]) representing the result.

---

[62] Furthermore, although \xintexpr uses \string, it is escape-char agnostic. It should work with any \escapechar setting including -1.

The once expanded \xintexpr is \romannumeral0\xintexpro. is similarly \xintiexpro \xintiiexpro and \xintfloatexpro. For an example see subsection 15.18.

An expression can only be legally finished by a \relax token, which will be absorbed.

It is quite possible to nest expressions among themselves; for example, if one needs inside an \xintiiexpr...\relax to do some computations with fractions, rounding the final result to an integer, one just has to insert \xintiexpr...\relax. The functioning of the infix operators will not be in the least affected from the fact that the outer ``environment'' is the \xintiiexpr one.

15.5. \xintDigits*, \xintSetDigits*

These starred variants of \xintDigits and \xintSetDigits execute \xintreloadxinttrig.

15.6. \xintiexpr, \xinttheiexpr

x ★ Equivalent to doing \xintexpr round(...)\relax (more precisely, round is applied to each leaf item of the ople independently of its depth).

Intermediate calculations are exact, only the final output gets rounded. Half integers are rounded towards +∞ for positive numbers and towards -∞ for negative ones.

An optional parameter D within brackets, immediately after \xintiexpr is allowed: it instructs (for D>0) the expression to do its final rounding to the nearest value with that many digits after the decimal mark, *i.e.*, \xintiexpr [D] <expression>\relax is equivalent (in case of a single expression) to \xintexpr round(<expression>, D)\relax.

\xintiexpr [0] ... is the same as \xintiexpr ... and rounds to an integer.

New with 1.4a The case of negative D gives quantization to an integer multiple of 1e-D.

If truncation rather than rounding is needed on can use \xintexpr trunc(...)\relax for truncation to an integer or \xintexpr trunc(...,D)\relax for quantization to an integer multiple or 1eD. But this works only for a single scalar value.

Already on October 20, 2015, it was suggested by Крум to give some meaning to negative D. The suggestion was to let it act like -D but remove trailing zeroes of the output. Finally, I opted rather for quantization.

15.7. \xintiiexpr, \xinttheiiexpr

x ★ This variant does not know fractions. It deals almost only with long integers. Comma separated lists of expressions are allowed.

> It maps / to the *rounded* quotient. The operator // is, like in \xintexpr...\relax, mapped to *truncated* division. The Euclidean quotient (which for positive operands is like the truncated quotient) was, prior to release 1.1, associated to /. The function quo(a,b) can still be employed.

The \xintiiexpr-essions use the `ii' macros for addition, subtraction, multiplication, power, square, sums, products, Euclidean quotient and remainder.

The round, trunc, floor, ceil functions are still available, and are about the only places where fractions can be used, but / within, if not somehow hidden will be executed as integer rounded division. To avoid this one can wrap the input in qfrac: this means however that none of the normal expression parsing will be executed on the argument.

To understand the illustrative examples, recall that round and trunc have a second (non negative) optional argument. In a normal \xintexpr-essions, round and trunc are mapped to \xintRound and \xintTrunc, in \xintiiexpr-essions, they are mapped to \xintiiRound and \xintiiTrunc.

\xinttheiiexpr 5/3, round(5/3,3), trunc(5/3,3), trunc(\xintDiv {5}{3},3),
trunc(\xintRaw {5/3},3)\relax{} are problematic, but

177

```
%
\xinttheiiexpr 5/3,  round(qfrac(5/3),3), trunc(qfrac(5/3),3), floor(qfrac(5/3)),
ceil(qfrac(5/3))\relax{} work!
```
2, 2000, 2000, 2000, 2000 are problematic, but 2, 1667, 1666, 1, 2 work!

On the other hand decimal numbers and scientific numbers can be used directly as arguments to the num, round, or any function producing an integer.

> Scientific numbers will be represented with as many zeroes as necessary, thus one does not want to insert num(1e100000) for example in an \xintiiexpression!

```
\xinttheiiexpr num(13.4567e3)+num(10000123e-3)\relax % should (num truncates) compute 13456+10000
```
23456

The reduce function is not available and will raise an error. The frac function also. The sqrt function is mapped to \xintiiSqrt which gives a truncated square root. The sqrtr function is mapped to \xintiiSqrtR which gives a rounded square root.

One can use the Float macros if one is careful to use num, or round etc... on their output.
```
\xinttheiiexpr \xintFloatSqrt [20]{2}, \xintFloatSqrt [20]{3}\relax % no operations
```

```
\noindent The next example requires the |round|, and one could not put the |+| inside it:
```

```
\xinttheiiexpr round(\xintFloatSqrt [20]{2},19)+round(\xintFloatSqrt [20]{3},19)\relax
```

```
(the second argument of |round| and |trunc| tells how many digits from after the
decimal mark one should keep.)
```
14142135623730950488[-19], 17320508075688772935[-19]
The next example requires the round, and one could not put the + inside it:
31462643699419723423
(the second argument of round and trunc tells how many digits from after the decimal mark one should keep.)

The whole point of \xintiiexpr is to gain some speed in *integer-only* algorithms, and the above explanations related to how to nevertheless use fractions therein are a bit peripheral. We observed (2013/12/18) of the order of 30% speed gain when dealing with numbers with circa one hundred digits (1.2: this info may be obsolete).

15.8. \xintboolexpr, \xinttheboolexpr

x ★ Equivalent to doing \xintexpr ...\relax and returning True if the result does not vanish, and False if the result is zero. As \xintexpr, this can be used on comma separated lists of expressions, and even bracketed lists.

Changed at 1.4!

It can be customized, one only needs to modify the following:
```
\def\xintboolexprPrintOne#1{\xintiiifNotZero{#1}{True}{False}}%
```
Not only are True and False usable in input, also true and false are pre-declared variables.

Maybe obsolete:

There is slight quirk in case it is used as a sub-expression: the boolean expression needs at least one logic operation else the value is not standardized to 1 or 0, for example we get from
```
\xinttheexpr \xintboolexpr 1.23\relax\relax\newline
```
123e-2
which is to be compared with
```
\xinttheboolexpr 1.23\relax
```
True

15.9. \xintfloatexpr, \xintthefloatexpr

x ★ \xintfloatexpr...\relax is exactly like \xintexpr...\relax but with the four binary operations

178

and the power function are mapped to \xintFloatAdd, \xintFloatSub, \xintFloatMul, \xintFloatDiv
and \xintFloatPower, respectively.[63]

The target precision for the computation is from the current setting of \xintDigits. Comma sep-
arated lists of expressions are allowed.

An optional parameter within brackets is allowed:

- if positive it instructs the macro to round the result to that many digits of precision. It thus
  makes sense to employ it only if this parameter is less than the \xinttheDigits precision.

- if negative it means to trim off that many digits (of course, in the sense of rounding the
  values to shorter mantissas). Don't use it to trim all digits (or more than all)!

Since 1.2f all float operations first round their arguments; a parsed number is not rounded prior
to its use as operand to such a float operation.

\thexintfloatexpr is synonym to \xintthefloatexpr.

\xintDigits:=36\relax

\xintthefloatexpr (1/13+1/121)*(1/179-1/173)/(1/19-1/18)\relax

0.00564487459334466559166166079096852897

\xintthefloatexpr\xintexpr (1/13+1/121)*(1/179-1/173)/(1/19-1/18)\relax\relax

0.00564487459334466559166166079096852912

The latter is the rounding of the exact result. The former one has its last three digits wrong
due to the cumulative effect of rounding errors in the intermediate computations, as compared to
exact evaluations.

I recall here from subsection 5.2 that with release 1.2f the float macros for addition, sub-
traction, multiplication and division round their arguments first to P significant places with P
the asked-for precision of the output; and similarly the power macros and the square root macro.
This does not modify anything for computations with arguments having at most P significant places
already.

15.10. **\xinteval**, **\xintieval**, **\xintiieval**, **\xintfloateval**

x ⋆ \xinteval is an *f*-expandable macro which is basically defined like this (DON'T BELIEVE THIS; it
has been entirely revamped at 1.4):

\def\xinteval#1{\romannumeral-`0\xinttheexpr#1\relax}% OLD DEFINITION < 1.4

thus expands in two steps (its exact definition differs from the one given above in order to achieve
a slight optimization).

\xinteval{add(x^2, x = 100..110), add(x^3, x = 100..110)}

121385, 12768525

x ⋆   \xintieval is similarly related to \xinttheiexpr. Its optional argument must be located inside
the braces:

\xintieval{[7] 355/113}

3.1415929

x ⋆   \xintiieval is similarly related to \xinttheiiexpr.

\xintiieval{add(x^2, x = 100..110), add(x^3, x = 100..110)}

121385, 12768525

x ⋆   \xintfloateval is similarly related to \xintthefloatexpr. Its optional argument must be located
inside the braces:

\xintfloateval{[7] 355/113}

3.141593

When negative it tells how many digits to remove from the prevailing precision (\xinttheDigits):

\xintfloateval{[-2] 355/113} has \xinttheDigits\ minus 2 digits.

---

[63] Since 1.2f the ^ handles half-integer exponents, contrarily to \xintFloatPower.

3.1415929203540 has 16 minus 2 digits.

These macros are useful when one uses some extra wrapper doing some parsing of its input, like the \num macro of siunitx, which would choke on some of the syntax elements allowed inside \xintexpr..⤸ .\relax (for example brackets). As shown in the above examples, these macros, like the underlying parsers accept arbitrarily many comma separated expressions.

15.11. Using an expression parser within another one

This was already illustrated before. In the following:
```
\xintfloatexpr \xintexpr add(1/i, i=1234..1243)\relax ^100\relax
```
5.136088460396579e-210, the inner sum is computed exactly. Then it will be rounded to \xinttheD⤸ igits significant digits, and then its power will be evaluated as a float operation. One should avoid the "\xintthe" parsers in inner positions as this induces digit by digit parsing of the inner computation result by the outer parser. Here is the same computation done with floats all the way:

```
\xintfloatexpr add(1/i, i=1234..1243)^100\relax
```
5.136088460396643e-210

Not surprisingly this differs from the previous one which was exact until raising to the 100th power.

The fact that the inner expression occurs inside a bigger one has nil influence on its behaviour. There is the limitation though that the outputs from \xintexpr and \xintfloatexpr can not be used directly in \xinttheiiexpr integer-only parser. But one can do:
```
\xintiiexpr round(\xintfloatexpr 3.14^10\relax)\relax % or trunc
```
93174

15.12. The \xintthecoords macro

It converts (in two expansion steps) the expansion result of \xintfloatexpr (or \xintexpr or \xintiiexpr) into the (a, b) (c, d) ... format for list of coordinates as expected by the TikZ coordinates syntax.

```
\begin{figure}[htbp]
\centering\begin{tikzpicture}[scale=10]\xintDigits:=8\relax
  \clip (-1.1,-.25) rectangle (.3,.25);
  \draw [blue] (-1.1,0)--(1,0);
  \draw [blue] (0,-1)--(0,+1);
  \draw [red] plot[smooth] coordinates {%
%%% (\xintthecoords converts output of next expression into (x1, y1) (x2, y2)... format)
    \xintthecoords\xintfloatexpr
%%%    This syntax -1+[0..4]/2 is currenty dropped at xint 1.4
%%%    seq((x^2-1,mul(x-t,t=-1+[0..4]/2)),x=-1.2..[0.1]..+1.2)\relax
%%%    Use this:
    seq((x^2-1,mul(x-t,t=seq(-1+u/2, u=0..4))),x=-1.2..[0.1]..+1.2)
    \relax
     };
\end{tikzpicture}
\caption{Coordinates with \cs{xintthecoords}.}
\end{figure}
```

Unstable! It is currently undecided how \xintthecoords should handle bracketed data. Currently, it (or Tik⤸ Z) will break it the input contains nested structures. One can use it with flat() which removes all nesting. And in combination with zip() it is easy to plot data given by some mechanism in separate lists of x- and y-coordinates (see an example in next section)

15.13. The \xintthespaceseparated macro

New with 1.4a It converts (in two expansion steps) the expansion result of \xintfloatexpr (or \xintexpr or \xintiiexpr) into the space separated format suitable for usage with PS-Tricks \listplot macro.
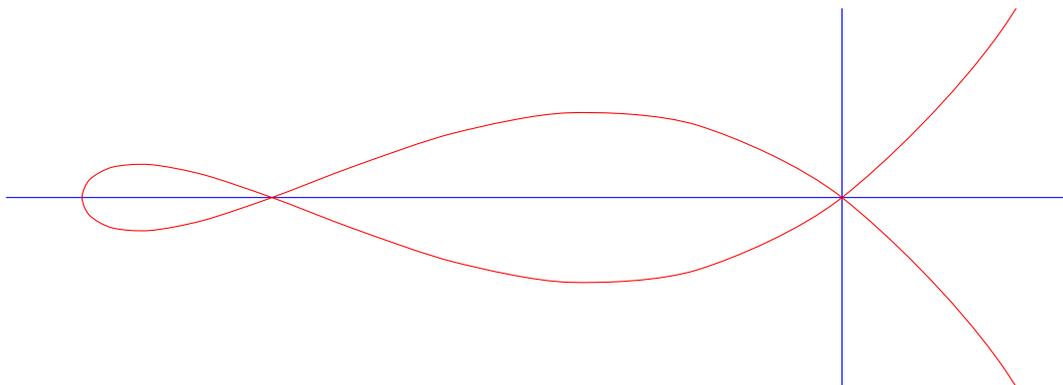
Figure 2: Coordinates with \xintthecoords.

Here is for example some syntax (the replacement text of \foo, which is used here only to show that indeed complete expansion is attained in two steps) which can be used as argument to \listplot. Using 4 fractional decimal digits is sufficient when unit is the centimeter (it gives a fixed point precision of one micron, amply enough for plots...).

```
\oodef\foo{%
\xintthespaceseparated\xintiexpr[4]\xintfloatexpr seq((i, log10(i)), i=1..[0.5]..10)\relax\relax
}\meaning\foo
```

macro:->1.0000 0 1.5000 0.1761 2.0000 0.3010 2.5000 0.3979 3.0000 0.4771 3.5000 0.5441 4.0000 0.6021 4.5000 0.6532 5.0000 0.6990 5.5000 0.7404 6.0000 0.7782 6.5000 0.8129 7.0000 0.8451 7.5000 0.8751 8.0000 0.9031 8.5000 0.9294 9.0000 0.9542 9.5000 0.9777 10.0000 1.0000

Here we don't really need the inner \xintfloatexpr...\relax because the log10() function works the same in the exact parser \xintexpr but in general this is recommended.

Unstable! It is currently undecided how \xintthespaceseparated should handle bracketed data. Currently, it (or \listplot) will break if the input contains nested structures. One can use it with flat() which removes all nesting. And in combination with zip() it is easy to plot data given by some mechanism in separate lists of x- and y-coordinates.

```
% let's imagine we have something like this
\def\Xcoordinates{1, 3, 5, 7, 9}
\def\Ycoordinates{1, 9, 25, 49, 81}
% then:
|\xintthespaceseparated\xintexpr flat(zip([\Xcoordinates], [\Ycoordinates]))\relax|
is suitable to use as argument to |\listplot|, as it expands to
\xintthespaceseparated\xintexpr flat(zip([\Xcoordinates], [\Ycoordinates]))\relax
```

\xintthespaceseparated\xintexpr flat(zip([\Xcoordinates], [\Ycoordinates]))\relax is suitable to use as argument to \listplot, as it expands to 1 1 3 9 5 25 7 49 9 81

15.14. \xintifboolexpr, \xintifboolfloatexpr, \xintifbooliiexpr

xnn ⋆ \xintifboolexpr{⟨*expr*⟩}{⟨*YES*⟩}{⟨*NO*⟩} does \xinttheexpr<expr>\relax and then executes the ⟨*YES*⟩ or the ⟨*NO*⟩ branch depending on whether the outcome was non-zero or zero. Thus one can read *if bool expr* as meaning *if not zero*:

if ⟨*expr*⟩-ession does not vanish do ⟨*YES*⟩ else do ⟨*NO*⟩

The expression is not limited to using only comparison operators and Boolean logic (<, >, ==, !=, &&, ||, all(), any(), xor(), bool(), togl(), ...), it can be the most general computation.

xnn ⋆ \xintifboolfloatexpr{⟨*expr*⟩}{⟨*YES*⟩}{⟨*NO*⟩} does \xintthefloatexpr⟨*expr*⟩\relax and then executes the ⟨*YES*⟩ or the ⟨*NO*⟩ branch depending on whether the outcome was non zero or zero.

xnn ⋆ \xintifbooliiexpr{⟨*expr*⟩}{⟨*YES*⟩}{⟨*NO*⟩} does \xinttheiiexpr⟨*expr*⟩\relax and then executes the ⟨*YES*⟩ or the ⟨*NO*⟩ branch depending on whether the outcome was non zero or zero.

The expression argument must be a single one, comma separated sub-expressions will cause low-level errors.

15.15. \xintifsgnexpr, \xintifsgnfloatexpr, \xintifsgniiexpr

x n n n ★   \xintifsgnexpr{⟨*expr*⟩}{⟨<0⟩}{⟨=0⟩}{⟨>0⟩} evaluates the \xintexpression and chooses the branch corresponding to its sign.

x n n n ★   \xintifsgnfloatexpr{⟨*expr*⟩}{⟨<0⟩}{⟨=0⟩}{⟨>0⟩} evaluates the \xintfloatexpression and chooses the branch corresponding to its sign.

x n n n ★   \xintifsgniiexpr{⟨*expr*⟩}{⟨<0⟩}{⟨=0⟩}{⟨>0⟩} evaluates the \xintiiexpression and chooses the branch corresponding to its sign.

The expression argument must be a single one, comma separated sub-expressions will cause low-level errors.

15.16. The \xintNewExpr, \xintNewIIExpr, \xintNewFloatExpr, \xintNewIExpr, and \xintNewBoolExpr macros

\xintNewExpr macro is used as:

\xintNewExpr{\myformula}[n]{⟨*stuff*⟩}, where

- ⟨*stuff*⟩ will be inserted inside \xinttheexpr . . . \relax,

- n is an integer between zero and nine, inclusive, which is the number of parameters of \myformula,

- the placeholders #1, #2, ..., #n are used inside ⟨*stuff*⟩ in their usual rôle,[64] [65]

- the [n] is *mandatory*, even for n=0.[66]

- 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 protection against active characters is done automatically (as long as the whole thing has not already been fetched as a macro argument and the catcodes correspondingly already frozen).

It (if it succeeds) will be a completely expandable macro entirely built-up using \xintAdd, \xintSub, \xintMul, \xintDiv, \xintPow, etc. . . as corresponds to the expression written with the infix operators. Macros created by \xintNewExpr can thus be nested.

```
    \xintNewFloatExpr \FA [2]{(#1+#2)^10}
    \xintNewFloatExpr \FB [2]{sqrt(#1*#2)}
\begin{enumerate}[nosep]
    \item \FA {5}{5}
    \item \FB {30}{10}
    \item \FA {\FB {30}{10}}{\FB {40}{20}}
\end{enumerate}
```

1. 1.000000000000000e10
2. 17.32050807568877
3. 3.891379490446502e16

The documentation is much shortened here because \xintNewExpr and \xintdeffunc are very much related one with the other.

---

[64] if \xintNewExpr is used inside a macro, the #'s must be doubled as usual.   [65] the #'s will in pratice have their usual catcode, but category code other #'s are accepted too.   [66] there is some use for \xintNewExpr[0] compared to an \edef as \xintNewExpr has some built-in catcode protection.

☞ {
> ATTENTION!
> The original spirit of \xintNewExpr was to define a (possibly very big) macro using only xintfrac, and this means in particular that it must be used only with arguments compatible with the xintfrac input format.
> Thus an \xintexpr declared variable has no chance to work, it must be wrapped explicitly in \xinteval{...} to be fetched as argument to a macro constructed by \xintNewExpr.

They share essentially the same limitations.

Notice though that \xintNewFloatExpr accepts and recognizes the optional argument [Q] of \xintfloatexpr, contrarily to \xintdeffloatfunc. Use an \empty in case the contents are not known in advance.

Historical note: prior to 1.4, xintexpr used a \csname..\endcsname encapsulation technique which impacted the string pool memory. The \xintNewExpr was designed as a method to pre-parse the expression and produce one single, gigantic, nested usage of the relevant xintfrac macros. This way, only those macros were expanded which had nil impact on the TeX string pool.

Later on it was found that this mechanism could be employed to define functions. Basically underneath 98% of \xintNewExpr and \xintdeffunc are using the same shared code.

15.17. Analogies and differences of \xintiiexpr with \numexpr

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

- Contrarily to \numexpr, the \xintiiexpr parser will stop expanding only after having encountered (and swallowed) a *mandatory* \relax token.

- In particular, spaces between digits (and not only around infix operators or parentheses) do not stop \xintiiexpr, contrarily to the situation with numexpr: \the\numexpr 7 + 3 5\relax expands (in one step)[67] to 105\relax, whereas \xintthe\xintiiexpr 7 + 3 5\relax expands (in two steps) to 42.[68]

- Inside an \edef, an expression \xintiiexpr...\relax get fully evaluated, whereas \numexpr without \the or \number prefix would not, if not itself embedded in another \the\numexpr or similar context.

- (ctd.) The private format to which \xintiiexpr...\relax (et al.) evaluates may use \xintthe prefix to turn into explicit digits, (for example in arguments to some macros which expand their arguments). The \the TeX primitive prefix would not work here.

- (ctd.) One can embed a \numexpr...\relax (with its \relax!) inside an \xintiiexpr...\relax without \the or \number, but the reverse situation requires usage of \xintthe or \xinteval user interface,

- \numexpr -(1)\relax is illegal. In contrast \xintiiexpr -(1)\relax is perfectly legal and gives the expected result (what else ?).

- \numexpr 2\cnta\relax is illegal (with \cnta a \count register.) In contrast \xintiiexpr 2\cnta\relax is perfectly legal and will do the tacit multiplication.

- \the\numexpr or \number\numexpr expands in one step, but \xintthe\xintiiexpr or \xinttheiiexpr needs two steps.

---

[67] The \numexpr triggers continued expansion after the space following the 3 to check if some operator like + is upstream. But after having found the 5 it treats it as and end-marker. [68] Since 1.21 one can also use the underscore _ to separate digits for readability of long numbers.

15.18. Chaining expressions for expandable algorithmics

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

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

```
\catcode`_ 11
\def\Fibonacci #1{%  \Fibonacci{N} computes F(N) with F(0)=0, F(1)=1.
    \expandafter\Fibonacci_a\expandafter
        {\the\numexpr #1\expandafter}\expandafter
        {\romannumeral0\xintiiexpro 1\expandafter\relax\expandafter}\expandafter
        {\romannumeral0\xintiiexpro 1\expandafter\relax\expandafter}\expandafter
        {\romannumeral0\xintiiexpro 1\expandafter\relax\expandafter}\expandafter
        {\romannumeral0\xintiiexpro 0\relax}}
%
\def\Fibonacci_a #1{%
    \ifcase #1
          \expandafter\Fibonacci_end_i
    \or
          \expandafter\Fibonacci_end_ii
    \else
          \ifodd #1
              \expandafter\expandafter\expandafter\Fibonacci_b_ii
          \else
              \expandafter\expandafter\expandafter\Fibonacci_b_i
          \fi
    \fi {#1}%
}% * signs are omitted from the next macros, tacit multiplications
\def\Fibonacci_b_i #1#2#3{\expandafter\Fibonacci_a\expandafter
  {\the\numexpr #1/2\expandafter}\expandafter
  {\romannumeral0\xintiiexpro sqr(#2)+sqr(#3)\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiiexpro (2#2-#3)#3\relax}%
}% end of Fibonacci_b_i
\def\Fibonacci_b_ii #1#2#3#4#5{\expandafter\Fibonacci_a\expandafter
  {\the\numexpr (#1-1)/2\expandafter}\expandafter
  {\romannumeral0\xintiiexpro sqr(#2)+sqr(#3)\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiiexpro (2#2-#3)#3\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiiexpro #2#4+#3#5\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiiexpro #2#5+#3(#4-#5)\relax}%
}% end of Fibonacci_b_ii
%          code as used on title page:
%\def\Fibonacci_end_i   #1#2#3#4#5{\xintthe#5}
%\def\Fibonacci_end_ii #1#2#3#4#5{\xinttheiiexpr #2#5+#3(#4-#5)\relax}
%          new definitions:
\def\Fibonacci_end_i   #1#2#3#4#5{{#4}{#5}}% {F(N+1)}{F(N)} in \xintexpr format
\def\Fibonacci_end_ii #1#2#3#4#5%
    {\expandafter
     {\romannumeral0\xintiiexpro #2#4+#3#5\expandafter\relax
      \expandafter}\expandafter
     {\romannumeral0\xintiiexpro #2#5+#3(#4-#5)\relax}}% idem.
% \FibonacciN returns F(N) (in encapsulated format: needs \xintthe for printing)
\def\FibonacciN {\expandafter\xint_secondoftwo\romannumeral-`0\Fibonacci }%
\catcode`_ 8
```

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

xintexpr format.

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

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

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

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

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

But \xintexpr, \xintiiexpr, ..., expand fully in an \edef, with the completely expanded result encapsulated in a private format.

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

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

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

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

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

then \edef is necessary.

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

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

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

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

```
\newcounter{myindex}% not  which would overwrite theindex environment!
% (many have probably been bitten by this trap)
\tabskip 1ex
```

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

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

```
   \fdef\Fibxxx{\FibonacciN {30}}%
   \setcounter{myindex}{30}%
\vbox{\halign{\bfseries#.\hfil&#\hfil &\hfil #\cr
  \xintFor* #1 in {\FibonacciSeq {30}{59}}\do
  {\themyindex &\xintthe#1 &
    \xintiiRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{myindex}\cr }}%
}\vrule
\vbox{\halign{\bfseries#.\hfil&#\hfil &\hfil #\cr
  \xintFor* #1 in {\FibonacciSeq {60}{89}}\do
  {\themyindex &\xintthe#1 &
    \xintiiRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{myindex}\cr }}%
}\vrule
\vbox{\halign{\bfseries#.\hfil&#\hfil &\hfil #\cr
  \xintFor* #1 in {\FibonacciSeq {90}{119}}\do
  {\themyindex &\xintthe#1 &
   \xintiiRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{myindex}\cr }}%
}%
```

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

186

15.19. When expandability is too much

Let's use the macros of subsection 15.18 related to Fibonacci numbers. Notice that the 47th Fibonacci number is 2971215073 thus already too big for TeX and ε-TeX.

The `\FibonacciN` macro found in subsection 15.18 is completely expandable, it is even *f-expan*dable. We need a wrapper with `\xintthe` prefix

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

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

The `\xintthe` prefix also allows its use it as argument to the xint macros: for example if we are interested in knowing how many digits F(1250) has, it suffices to issue `\xintLen {\theFibonacciN {1250}}` (which expands to 261). Or if we want to check the formula gcd(F(1859), F(1573)) = F(gcd(1859, 1573)) = F(143), we only need[69]

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

which produces:

343358302784187294870275058337 = 343358302784187294870275058337

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

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

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

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

The identity gcd(F(1859), F(1573)) = F(gcd(1859, 1573)) can be checked via evaluation of both sides: gcd(F(1859), F(1573)) = gcd(1440582791304425119877168915150404286991316149502348101422668636701088272597575494722482437753529619459794869227357628882216309358018264080851775319974256956055294350288615852451737250886736422228492908228952455838894954421926557604129992902556597971133787610545221762349084152997981141319966008751768970341099752007999361070757601952087632458469495551467505894985013610208598628752325727241, 2443841925195118573328279459777762619985399024815706192326053609007840133940367432124452232789599095158695811031891779769058003274151632595307616686661013725200866754096569888951010022888016831459347310131566517721593249344798634399479371195758766544765827958909282390070313197135548122004938644531329524847747273166471511289078393) = 343358302784187294870275058337 = F(gcd(1859, 1573)) = F(143) = 343358302784187294870275058337.

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

I have no answer: it made definitely sense at the start of xint (see subsection 5.13) and once started I could not stop.

---

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

15.20. Acknowledgements (2013/05/25)

I was greatly helped in my preparatory thinking, prior to producing such an expandable parser, by the commented source of the `l3fp` package, specifically the `l3fp-parse.dtx` file (in the version of April-May 2013; I think there was in particular a text called ``roadmap'' which was helpful). Also the source of the `calc` package was instructive, despite the fact that here for `\xintexpr` the principles are necessarily different due to the aim of achieving expandability.

16. More examples with xinttools or xintexpr or both

Note: `xintexpr`.sty automatically loads `xinttools`.sty.

The examples given here start to feel dated and are currently in need of some rewrite to better illustrate newer features of the package.

16.1. More examples with dummy variables

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

```
Prime numbers are always cool
\xinttheiiexpr seq((seq((subs((x/:m)?{(m*m>x)?{1}{0}}{-1},m=2n+1))
                    ??{break(0)}{omit}{break(1)},n=1++))?{x}{omit},
              x=10001..[2]..10200)\relax
```
Prime numbers are always cool 10007, 10009, 10037, 10039, 10061, 10067, 10069, 10079, 10091, 10093, 10099, 10103, 10111, 10133, 10139, 10141, 10151, 10159, 10163, 10169, 10177, 10181, 10193

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

```
The first Fibonacci number beyond |2^64| bound is
\xinttheiiexpr subs(iterr(0,1;(@1>N)?{break(i)}{@1+@2},i=1++),N=2^64)\relax{}
and the previous number was its index.
```
The first Fibonacci number beyond 2^64 bound is 94 and the previous number was its index.

```
One more recursion:
\def\syr #1{\xinttheiiexpr rseq(#1; (@<=1)?{break(i)}{odd(@)?{3@+1}{@//2}},i=0++)\relax}
The 3x+1 problem: \syr{231}\par
```
The 3x+1 problem: 231, 694, 347, 1042, 521, 1564, 782, 391, 1174, 587, 1762, 881, 2644, 1322, 661, 1984, 992, 496, 248, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1, 127

OK, a final one:[70]

```
\def\syrMax #1{\xintiiexpr iterr(#1,#1;even(i)?
                          {(@2<=1)?{break(@1,i//2)}{odd(@2)?{3@2+1}{@2//2}}}
                          {(@1>@2)?{@1}{@2}},i=0++)\relax }
With initial value 1161, the maximal intermediate value and the number of steps
needed to reach 1 are respectively \syrMax{1161}.\par
```
With initial value 1161, the maximal intermediate value and the number of steps needed to reach 1 are respectively 190996, 181.

Look at the Brent-Salamin algorithm implementation for a more interesting recursion.

---

[70] Prior to `1.4`, the break() worked differently and here one used only break(i/2) for the same result. In retrospect this looks like a bug of break() inside an iterr().

16.2. Completely expandable prime test

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

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

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

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

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

Our integers are short, but without \expandafter's with \@firstoftwo, or some other related techniques, direct use of \ifnum..\fi tests is dangerous. So to make the macro more efficient we are going to use the expandable tests provided by the package etoolbox[71]. The macro becomes:

```
\def\IsPrime #1%
   {\ifnumodd {#1}
    {\xintANDof % odd case
     {\xintApply {\remainder {#1}}{\xintSeq [2]{3}{\xintiiSqrt{#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}{\xintiiSqrt{#1}}}}%
       }}% END OF THE ODD BRANCH
     {\ifnumequal {#1}{2}{1}{0}}% EVEN BRANCH
   }
```

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

We did generate via the \xintSeq too many potential divisors though. Later sections give two variants: one with \xintiloop (subsection 16.3) which is still expandable and another one (subsection 16.6) which is a close variant of the \IsPrime code above but with the \xintFor loop, thus

---

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

breaking expandability. The `xintiloop variant` does not first evaluate the integer square root, the `xintFor variant` still does. I did not compare their efficiencies.

Let us construct with this expandable primality test a table of the prime numbers up to 1000. We need to count how many we have in order to know how many tab stops one shoud add in the last row.[72] 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.
```

The table has been put in `float` which appears on this page. We had to be careful to use in the last row `\xintSeq` with its optional argument `[1]` so as to not generate a decreasing sequence from `1` to `0`, but really an empty sequence in case the row turns out to already have all its cells (which doesn't happen here but would with a number of columns dividing 168).

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

There are 168 prime numbers up to 1000.

---

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

16.3. Another completely expandable prime test

The \IsPrime macro from subsection 16.2 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.

```
\let\IsPrime\undefined \let\SmallestFactor\undefined % clean up possible previous mess
\newcommand{\IsPrime}[1] % returns 1 if #1 is prime, and 0 if not
  {\ifnumodd {#1}
    {\ifnumless {#1}{8}
      {\ifnumequal{#1}{1}{0}{1}}% 3,5,7 are primes
      {\if
       \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
{\centering
  \begin{tabular}{|c|*{10}c|}
    \hline
    \xintFor #1 in {0,1,2,3,4,5,6,7,8,9}\do {&\bfseries #1}\\
    \hline
    \bfseries 0&--&--&2&3&2&5&2&7&2&3\\
    \xintFor #1 in {1,2,3,4,5,6,7,8,9}\do
    {\bfseries #1%
      \xintFor #2 in {0,1,2,3,4,5,6,7,8,9}\do
      {&\SmallestFactor{#1#2}}\\}%
    \hline
  \end{tabular}\par
```

```
}
```

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

16.4. Miller-Rabin Pseudo-Primality expandably

The `isPseudoPrime(n)` is usable in `\xintiiexpr`-essions and establishes if its (positive) argument is a Miller-Rabin PseudoPrime to the bases $2, 3, 5, 7, 11, 13, 17$. If this is true and n < 341550071728321 (which has 15 digits) then n really is a prime number.

Similarly n = 3825123056546413051 (19 digits) is the smallest composite number which is a strong pseudo prime for bases $2, 3, 5, 7, 11, 13, 17, 19$ and 23. It is easy to extend the code below to include these additional tests (we could make the list of tested bases an argument too, now that I think about it.)

For more information see

https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Deterministic_variants_of_the_test

and

http://primes.utm.edu/prove/prove2_3.html

In particular, according to Jaeschke *On strong pseudoprimes to several bases,* Math. Comp., 61 (1993) 915-926, if n < $4, 759, 123, 141$ it is enough to establish Rabin-Miller pseudo-primality to bases a = $2, 7, 61$ to prove that n is prime. This range is enough for TeX numbers and we could then write a very fast expandable primality test for such numbers using only `\numexpr`. Left as an exercise...

```
% I ----------------------------- Modular Exponentiation
% Computes x^m modulo n (with m non negative).
% We will always use it with 1 < x < n
%
% With xint 1.4 we should use ? and ?? (although in the case at hand ifsgn()
% and if() would be ok but I should not say that).
%
\xintdefiifunc powmod_a(x, m, n) :=
        isone(m)?
            % m=1, return x modulo n
            {   x /: n  }
            % m > 1 test if odd or even and do recursive call
            {   odd(m)? {  x*sqr(powmod_a(x, m//2, n)) /: n }
                        {    sqr(powmod_a(x, m//2, n)) /: n }
            }
        ;
\xintdefiifunc powmod(x, m, n) := (m)?{powmod_a(x, m, n)}{1};

%% Syntax used before xint 1.4:
% \xintdefiifunc powmod_a(x, m, n) :=
%        ifone(m,
%            % m=1, return x modulo n
%              x /: n,
%            % m > 1 test if odd or even and do recursive call
```

```
%                 if(odd(m), (x*sqr(powmod_a(x, m//2, n))) /: n,
%                           sqr(powmod_a(x, m//2, n))  /: n
%                   )
%           );
% \xintdefiifunc powmod(x, m, n) := if(m, powmod_a(x, m, n), 1);

% II --------------------------- Miller-Rabin compositeness witness

% n=2^k m + 1 with m odd and k at least 1

% Choose 1<x<n.
% compute y=x^m modulo n
% if equals 1 we can't say anything
% if equals n-1 we can't say anything
% else put j=1, and
% compute repeatedly the square, incrementing j by 1 each time,
% thus always we have y^{2^{j-1}}
%   -> if at some point n-1 mod n found, we can't say anything and break out
%   -> if however we never find n-1 mod n before reaching
%        z=y^{2^{k-1}} with j=k
%         we then have z^2=x^{n-1}.
    % Suppose z is not -1 mod n. If z^2 is 1 mod n, then n can be prime only if
    % z is 1 mod n, and we can go back up, until initial y, and we have already
    % excluded y=1. Thus if z is not -1 mod n and z^2 is 1 then n is not prime.
    % But if z^2 is not 1, then n is not prime by Fermat. Hence (z not -1 mod n)
    % implies (n is composite). (Miller test)

% let's use again xintexpr indecipherable (except to author) syntax. Of course
% doing it with macros only would be faster.

% Here \xintdefiifunc is not usable because not compatible with iter, break, ...
% but \xintNewFunction comes to the rescue.

\xintNewFunction{isCompositeWitness}[4]{% x=#1, n=#2, m=#3, k=#4
   subs((y==1)?{0}
          {iter(y;(j=#4)?{break(!(@==#2-1))}
                        {(@==#2-1)?{break(0)}{sqr(@)/:#2}},j=1++)}
          ,y=powmod(#1,#3,#2))}

% added note (2018/03/07) it is possible in the above that m=#3 is never
% zero, so we should rather call powmod_a for a small gain, but I don't
% have time to re-read the code comments and settle this.

% III ------------------------------ Strong Pseudo Primes

% cf
%  http://oeis.org/A014233
%     <http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html>
%     <http://mathworld.wolfram.com/StrongPseudoprime.html>

% check if positive integer <49 si a prime.
% 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
\def\IsVerySmallPrime #1%
    {\ifnum#1=1 \xintdothis0\fi
      \ifnum#1=2 \xintdothis1\fi
      \ifnum#1=3 \xintdothis1\fi
      \ifnum#1=5 \xintdothis1\fi
```

```
        \ifnum#1=\numexpr (#1/2)*2\relax\xintdothis0\fi
        \ifnum#1=\numexpr (#1/3)*3\relax\xintdothis0\fi
        \ifnum#1=\numexpr (#1/5)*5\relax\xintdothis0\fi
        \xintorthat 1}

  \xintNewFunction{isPseudoPrime}[1]{% n = #1
       (#1<49)?% use ? syntax to evaluate only what is needed
% prior to 1.4 we had \xintthe#1 here but the actual tokens represented
% by this #1 when isPseudoPrime() function expands have changed and
% the correct way is now \xintiieval{#1} to hand over explicit digits to
% the \IsVerySmallPrime macro.
        {\IsVerySmallPrime{\xintiieval{#1}}}
        {(even(#1))?
         {0}
         {subs(%
          % L expands to two values m, k hence isCompositeWitness does get
          % its four variables x, n, m, k
          isCompositeWitness(2, #1, L)?
           {0}%
           {isCompositeWitness(3, #1, L)?
            {0}%
            {isCompositeWitness(5, #1, L)?
             {0}%
             {isCompositeWitness(7, #1, L)?
              {0}%
% above enough for N<3215031751 hence all TeX numbers
              {isCompositeWitness(11, #1, L)?
               {0}%
% above enough for N<2152302898747, hence all 12-digits numbers
              {isCompositeWitness(13, #1, L)?
               {0}%
% above enough for N<3474749660383
              {isCompositeWitness(17, #1, L)?
               {0}%
% above enough for N<341550071728321
               {1}%
              }% not needed to comment-out end of lines spaces inside
             }%  \xintexpr but this is too much of a habit for me with TeX!
            }%   I left some after the ? characters.
           }%
          }%
         }% this computes (m, k) such that n = 2^k m + 1, m odd, k>=1
         , L=iter(#1//2;(even(@))?{@//2}{break(@,k)},k=1++))%
        }%
       }%
  }

  % if needed:
  %\def\IsPseudoPrime #1{\xinttheiiexpr isPseudoPrime(#1)\relax}

  \noindent The smallest prime number at least equal to 3141592653589 is
  \xintiiexpr
     seq(isPseudoPrime(3141592653589+n)?
                    {break(3141592653589+n)}{omit}, n=0++)\relax.
  % we could not use 3141592653589++ syntax because it works only with TeX numbers
  \par
```

The smallest prime number at least equal to 3141592653589 is 3141592653601.

16.5. 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`.

   The code next generates a `table` which has been made into a float appearing on the next page. 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 acceptable numbers are treated here, but it would be easy to make a translation and use the `xint` macros, thus extending the scope to big numbers; naturally up to a cost in speed.

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

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

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

\def\factorize_b #1.#2.{\ifnum#1=1 \abortfactorize\fi
        \ifnum\numexpr #1-(#2-1)*#2<#2
               #1\abortfactorize
        \fi
        \ifnum \numexpr #1-#2=\numexpr ((#1/#2)-1)*#2\relax
             \expandafter\xint_firstoftwo
        \else\expandafter\xint_secondoftwo
        \fi
        {#2&\expandafter\factorize_b\the\numexpr#1/#2.#2.}%
        {\expandafter\factorize_b\the\numexpr #1\expandafter.%
                          \the\numexpr #2+2.}}%
\catcode`\_ 8
\begin{figure*}[ht!]
\centering\phantomsection\label{floatfactorize}\normalcolor
\tabskip1ex
\centeredline{\vbox{\halign {\hfil\strut#\hfil&&\hfil#\hfil\cr\noalign{\hrule}
        \xintiloop ["7FFFFFE0+1]
        \expandafter\bfseries\xintiloopindex &
        \ifnum\xintiloopindex="7FFFFFED
             \number"7FFFFFED\cr\noalign{\hrule}
        \expandafter\xintiloopskiptonext
        \fi
        \expandafter\factorize\xintiloopindex.\cr\noalign{\hrule}
        \ifnum\xintiloopindex<"7FFFFFFE
        \repeat
        \bfseries \number"7FFFFFFF&\number "7FFFFFFF\cr\noalign{\hrule}
}}}
```

```
\centeredline{A table of factorizations}
\end{figure*}
```

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

16.6. 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 16.2, here we consider a variant which will be slightly more efficient. This new \IsPrime has two parameters. The first one is a macro which it redefines to expand to the result of the primality test applied to the second argument. For convenience we use the etoolbox wrappers to various \ifnum tests, although here there isn't anymore the constraint of complete expandability (but using explicit \if..\fi in tabulars has its quirks); equivalent tests are provided by xint, but they have some overhead as they are able to deal with arbitrarily big integers.

```
\def\IsPrime #1#2% #1=\Result, #2=tested number (assumed >0).
{\edef\TheNumber {\the\numexpr #2}% hence #2 may be a count or \numexpr.
\ifnumodd {\TheNumber}
```

```
  {\ifnumgreater {\TheNumber}{1}
   {\edef\ItsSquareRoot{\xintiiSqrt \TheNumber}%
     \xintFor ##1 in {\xintintegers [3+2]}\do
     {\ifnumgreater {##1}{\ItsSquareRoot} % ##1 is a \numexpr.
                 {\def#1{1}\xintBreakFor}
                 {}%
      \ifnumequal {\TheNumber}{(\TheNumber/##1)*##1}
                  {\def#1{0}\xintBreakFor }
                  {}%
    }}
  {\def#1{0}}}% 1 is not prime
 {\ifnumequal {\TheNumber}{2}{\def#1{1}}{\def#1{0}}}%
 }
```

As we used `\xintFor` inside a macro we had to double the `#` in its `#1` parameter. Here is now the code which creates the prime table (the table has been put in a float, which should be found on page 198):

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

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

16.7. Factorizing again

Here is an *f-expan*dable macro which computes the factors of an integer. It uses the xint macros only.

```
  \catcode`\@ 11
```

```
\let\factorize\relax
\newcommand\Factorize [1]
        {\romannumeral0\expandafter\factorize\expandafter{\romannumeral-`0#1}}%
\newcommand\factorize [1]{\xintiiifOne{#1}{ 1}{\factors@a #1.{#1};}}%
\def\factors@a #1.{\xintiiifOdd{#1}
    {\factors@c 3.#1.}%
    {\expandafter\factors@b \expandafter1\expandafter.\romannumeral0\xinthalf{#1}.}}%
\def\factors@b #1.#2.{\xintiiifOne{#2}
    {\factors@end {2, #1}}%
    {\xintiiifOdd{#2}{\factors@c 3.#2.{2, #1}}%
                    {\expandafter\factors@b \the\numexpr #1+\@ne\expandafter.%
                        \romannumeral0\xinthalf{#2}.}}%
}%
\def\factors@c #1.#2.{%
     \expandafter\factors@d\romannumeral0\xintiidivision {#2}{#1}{#1}{#2}%
}%
\def\factors@d #1#2#3#4{\xintiiifNotZero{#2}
    {\xintiiifGt{#3}{#1}
        {\factors@end {#4, 1}}%  ultimate quotient is a prime with power 1
        {\expandafter\factors@c\the\numexpr #3+\tw@.#4.}}%
    {\factors@e 1.#3.#1.}%
}%
\def\factors@e #1.#2.#3.{\xintiiifOne{#3}
    {\factors@end {#2, #1}}%
    {\expandafter\factors@f\romannumeral0\xintiidivision {#3}{#2}{#1}{#2}{#3}}%
}%
\def\factors@f #1#2#3#4#5{\xintiiifNotZero{#2}
    {\expandafter\factors@c\the\numexpr #4+\tw@.#5.{#4, #3}}%
    {\expandafter\factors@e\the\numexpr #3+\@ne.#4.#1.}%
}%
\def\factors@end #1;{\xintlistwithsep{, }{\xintRevWithBraces {#1}}}%
\catcode`@ 12
```
The macro will be acceptably efficient only with numbers having somewhat small prime factors.
```
\Factorize{16246355912554185673266068721806243461403654781833}
```
16246355912554185673266068721806243461403654781833, 13, 5, 17, 8, 29, 5, 37, 6, 41, 4, 59, 6

It puts a little stress on the input save stack in order not be bothered with previously gathered things.[73]

Its output is a comma separated list with the number first, then its prime factors with multiplicity. Let's produce something prettier:
```
\catcode`_ 11
\def\ShowFactors #1{\expandafter\ShowFactors_a\romannumeral-`0\Factorize{#1},\relax,\relax,}
\def\ShowFactors_a #1,{#1=\ShowFactors_b}
\def\ShowFactors_b #1,#2,{\if\relax#1\else#1^{#2}\expandafter\ShowFactors_b\fi}
\catcode`_ 8
$$\ShowFactors{16246355912554185673266068721806243461403654781833}$$
```

$$16246355912554185673266068721806243461403654781833 = 13^5 17^8 29^5 37^6 41^4 59^6$$

If we only considered small integers, we could write pure \numexpr methods which would be very much faster (especially if we had a table of small primes prepared first) but still ridiculously slow compared to any non expandable implementation, not to mention use of programming languages directly accessing the CPU registers...

---

[73] 2015/11/18 I have not revisited this code for a long time, and perhaps I could improve it now with some new techniques.

16.8. The Quick Sort algorithm illustrated

First a completely expandable macro which sorts a comma separated list of numbers.[74]

The \QSx macro expands its list argument, which may thus be a macro; its comma separated items must expand to integers or decimal numbers or fractions or scientific notation as acceptable to xintfrac, but if an item is itself some (expandable) macro, this macro will be expanded each time the item is considered in a comparison test! This is actually good if the macro expands in one step to the digits, and there are many many digits, but bad if the macro needs to do many computations. Thus \QSx should be used with either explicit numbers or with items being macros expanding in one step to the numbers (particularly if these numbers are very big).

If the interest is only in TeX integers, then one should replace the \xintifCmp macro with a suitable conditional, possibly helped by tools such as \ifnumgreater, \ifnumequal and \ifnumless from etoolbox (LaTeX only; I didn't see a direct equivalent to \xintifCmp.) Or, if we are dealing with decimal numbers with at most four+four digits, then one should use suitable \ifdim tests. Naturally this will boost consequently the speed, from having skipped all the overhead in parsing fractions and scientific numbers as are acceptable by xintfrac macros, and subsequent treatment.

```
% THE QUICK SORT ALGORITHM EXPANDABLY
% \usepackage{xintfrac} in the preamble (latex)
\makeatletter
% use extra safe delimiters
\catcode`! 3 \catcode`? 3
\def\QSx {\romannumeral0\qsx }%
% first we check if empty list (else \qsx@finish will not find a comma)
\def\qsx   #1{\expandafter\qsx@a\romannumeral-`0#1,!,?}%
\def\qsx@a #1{\ifx,#1\expandafter\qsx@abort\else
                     \expandafter\qsx@start\fi #1}%
\def\qsx@abort #1?{ }%
\def\qsx@start {\expandafter\qsx@finish\romannumeral0\qsx@b,}%
\def\qsx@finish ,#1{ #1}%
%
% we check if empty of single and if not pick up the first as Pivot:
\def\qsx@b ,#1#2,#3{\ifx?#3\xintdothis\qsx@empty\fi
                    \ifx!#3\xintdothis\qsx@single\fi
                    \xintorthat\qsx@separate {#1#2}{}{}{#1#2}#3}%
\def\qsx@empty  #1#2#3#4#5{ }%
\def\qsx@single #1#2#3#4#5?{, #4}%
\def\qsx@separate #1#2#3#4#5#6,%
{%
    \ifx!#5\expandafter\qsx@separate@done\fi
    \xintifCmp {#5#6}{#4}%
          \qsx@separate@appendtosmaller
          \qsx@separate@appendtoequal
          \qsx@separate@appendtogreater {#5#6}{#1}{#2}{#3}{#4}%
}%
%
\def\qsx@separate@appendtoequal   #1#2{\qsx@separate {#2,#1}}%
\def\qsx@separate@appendtogreater #1#2#3{\qsx@separate {#2}{#3,#1}}%
\def\qsx@separate@appendtosmaller #1#2#3#4{\qsx@separate {#2}{#3}{#4,#1}}%
%
\def\qsx@separate@done\xintifCmp #1%
```

---

[74] The code in earlier versions of this manual handled inputs composed of braced items. I have switched to comma separated inputs on the occasion of (link removed) The version here is like code 3 on (link removed) (which is about 3x faster than the earlier code it replaced in this manual) with a modification to make it more efficient if the data has many repeated values. A faster routine (for sorting hundreds of values) is provided as code 6 at the link mentioned in the footnote, it is based on Merge Sort, but limited to inputs which one can handle as TeX dimensions. This code 6 could be extended to handle more general numbers, as acceptable by xintfrac. I have also written a non expandable version, which is even faster, but this matters really only when handling hundreds or rather thousands of values.

```
                \qsx@separate@appendtosmaller
                \qsx@separate@appendtoequal
                \qsx@separate@appendtogreater #2#3#4#5#6#7?%
    {%
        \expandafter\qsx@f\expandafter {\romannumeral0\qsx@b #4,!,?}{\qsx@b #5,!,?}{#3}%
    }%
    %
    \def\qsx@f #1#2#3{#2, #3#1}%
    %
    \catcode`! 12 \catcode`? 12
    \makeatother

    % EXAMPLE
    \begingroup
    \edef\z {\QSx {1.0, 0.5, 0.3, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
                   1.3, 1.1, 0.7, 1.6, 0.6, 0.9, 0.8, 0.2, 0.1, 1.9}}
    \meaning\z

    \def\a {3.123456789123456789}\def\b {3.123456789123456788}
    \def\c {3.123456789123456790}\def\d {3.123456789123456787}
    \oodef\z {\QSx { \a, \b, \c, \d}}%
    % The space before \a to let it not be expanded during the conversion from CSV
    % values to List. The \oodef expands exactly twice (via a bunch of \expandafter's)
    \meaning\z
    \endgroup
```

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

macro:->\d , \b , \a , \c (the spaces after \d, etc... come from the use of the \meaning primitive.)

The choice of pivot as first element is bad if the list is already almost sorted. Let's add a variant which will pick up the pivot index randomly. The previous routine worked also internally with comma separated lists, but for a change this one will use internally lists of braced items (the initial conversion via \xintCSVtoList handles all potential spurious space problems).

```
    % QuickSort expandably on comma separated values with random choice of pivots
    % ====> Requires availability of \pdfuniformdeviate <====
    % \usepackage{xintfrac, xinttools} in preamble
    \makeatletter
    \def\QSx {\romannumeral0\qsx }% This is a f-expandable macro.
    % This converts from comma separated values on input and back on output.
    % **** NOTE: these steps (and the other ones too, actually) are costly if input
    %            has thousands of items.
    \def\qsx #1{\xintlistwithsep{, }%
                {\expandafter\qsx@sort@a\expandafter{\romannumeral0\xintcsvtolist{#1}}}}%
    %
    % we check if empty or single or double and if not pick up the first as Pivot:
    \def\qsx@sort@a #1%
        {\expandafter\qsx@sort@b\expandafter{\romannumeral0\xintlength{#1}}{#1}}%
    \def\qsx@sort@b #1{\ifcase #1
                          \expandafter\qsx@sort@empty
                          \or\expandafter\qsx@sort@single
                          \or\expandafter\qsx@sort@double
                          \else\expandafter\qsx@sort@c\fi {#1}}%
    \def\qsx@sort@empty  #1#2{ }%
    \def\qsx@sort@single #1#2{#2}%
    \catcode`_ 11
    \def\qsx@sort@double #1#2{\xintifGt #2{\xint_exchangetwo_keepbraces}{}#2}%
```

201

```
\catcode`_ 8
\def\qsx@sort@c        #1#2{%
    \expandafter\qsx@sort@sep@a\expandafter
                  {\romannumeral0\xintnthelt{\pdfuniformdeviate #1+\@ne}{#2}}#2?}%
\def\qsx@sort@sep@a #1{\qsx@sort@sep@loop {}{}{}{#1}}%
\def\qsx@sort@sep@loop #1#2#3#4#5%
{%
    \ifx?#5\expandafter\qsx@sort@sep@done\fi
    \xintifCmp {#5}{#4}%
          \qsx@sort@sep@appendtosmaller
          \qsx@sort@sep@appendtoequal
          \qsx@sort@sep@appendtogreater {#5}{#1}{#2}{#3}{#4}%
}%
%
\def\qsx@sort@sep@appendtoequal    #1#2{\qsx@sort@sep@loop {#2{#1}}}%
\def\qsx@sort@sep@appendtogreater #1#2#3{\qsx@sort@sep@loop {#2}{#3{#1}}}%
\def\qsx@sort@sep@appendtosmaller #1#2#3#4{\qsx@sort@sep@loop {#2}{#3}{#4{#1}}}%
%
\def\qsx@sort@sep@done\xintifCmp #1%
          \qsx@sort@sep@appendtosmaller
          \qsx@sort@sep@appendtoequal
          \qsx@sort@sep@appendtogreater #2#3#4#5#6%
{%
    \expandafter\qsx@sort@recurse\expandafter
                  {\romannumeral0\qsx@sort@a {#4}}{\qsx@sort@a {#5}}{#3}%
}%
%
\def\qsx@sort@recurse #1#2#3{#2#3#1}%
%
\makeatother

% EXAMPLES
\begingroup
\edef\z {\QSx {1.0, 0.5, 0.3, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
                1.3, 1.1, 0.7, 1.6, 0.6, 0.9, 0.8, 0.2, 0.1, 1.9}}
\meaning\z

\def\a {3.123456789123456789}\def\b {3.123456789123456788}
\def\c {3.123456789123456790}\def\d {3.123456789123456787}
\oodef\z {\QSx { \a, \b, \c, \d}}%
% The space before \a to let it not be expanded during the conversion from CSV
% values to List. The \oodef expands exactly twice (via a bunch of \expandafter's)
\meaning\z

\def\somenumbers{%
3997.6421, 8809.9358, 1805.4976, 5673.6478, 3179.1328, 1425.4503, 4417.7691,
2166.9040, 9279.7159, 3797.6992, 8057.1926, 2971.9166, 9372.2699, 9128.4052,
1228.0931, 3859.5459, 8561.7670, 2949.6929, 3512.1873, 1698.3952, 5282.9359,
1055.2154, 8760.8428, 7543.6015, 4934.4302, 7526.2729, 6246.0052, 9512.4667,
7423.1124, 5601.8436, 4433.5361, 9970.4849, 1519.3302, 7944.4953, 4910.7662,
3679.1515, 8167.6824, 2644.4325, 8239.4799, 4595.1908, 1560.2458, 6098.9677,
3116.3850, 9130.5298, 3236.2895, 3177.6830, 5373.1193, 5118.4922, 2743.8513,
8008.5975, 4189.2614, 1883.2764, 9090.9641, 2625.5400, 2899.3257, 9157.1094,
8048.4216, 3875.6233, 5684.3375, 8399.4277, 4528.5308, 6926.7729, 6941.6278,
9745.4137, 1875.1205, 2755.0443, 9161.1524, 9491.1593, 8857.3519, 4290.0451,
2382.4218, 3678.2963, 5647.0379, 1528.7301, 2627.8957, 9007.9860, 1988.5417,
2405.1911, 5065.8063, 5856.2141, 8989.8105, 9349.7840, 9970.3013, 8105.4062,
```

```
        3041.7779, 5058.0480, 8165.0721, 9637.7196, 1795.0894, 7275.3838, 5997.0429,
        7562.6481, 8084.0163, 3481.6319, 8078.8512, 2983.7624, 3925.4026, 4931.5812,
        1323.1517, 6253.0945}%

        \oodef\z {\QSx \somenumbers}% produced as a comma+space separated list
        % black magic as workaround to the shrinkability of spaces in last line...
        \hsize 87\fontcharwd\font`0
        \lccode`~=32
        \lowercase{\def~}{\discretionary{}{}{\kern\fontcharwd\font`0}}\catcode32 13
        \noindent\phantom{000}\scantokens\expandafter{\meaning\z}\par
        \endgroup
        macro:->0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8,
1.9, 2.0
        macro:->\d , \b , \a , \c
        macro:->1055.2154, 1228.0931, 1323.1517, 1425.4503, 1519.3302, 1528.7301, 1560.2458,
1698.3952, 1795.0894, 1805.4976, 1875.1205, 1883.2764, 1988.5417, 2166.9040, 2382.4218,
2405.1911, 2625.5400, 2627.8957, 2644.4325, 2743.8513, 2755.0443, 2899.3257, 2949.6929,
2971.9166, 2983.7624, 3041.7779, 3116.3850, 3177.6830, 3179.1328, 3236.2895, 3481.6319,
3512.1873, 3678.2963, 3679.1515, 3797.6992, 3859.5459, 3875.6233, 3925.4026, 3997.6421,
4189.2614, 4290.0451, 4417.7691, 4433.5361, 4528.5308, 4595.1908, 4910.7662, 4931.5812,
4934.4302, 5058.0480, 5065.8063, 5118.4922, 5282.9359, 5373.1193, 5601.8436, 5647.0379,
5673.6478, 5684.3375, 5856.2141, 5997.0429, 6098.9677, 6246.0052, 6253.0945, 6926.7729,
6941.6278, 7275.3838, 7423.1124, 7526.2729, 7543.6015, 7562.6481, 7944.4953, 8008.5975,
8048.4216, 8057.1926, 8078.8512, 8084.0163, 8105.4062, 8165.0721, 8167.6824, 8239.4799,
8399.4277, 8561.7670, 8760.8428, 8809.9358, 8857.3519, 8989.8105, 9007.9860, 9090.9641,
9128.4052, 9130.5298, 9157.1094, 9161.1524, 9279.7159, 9349.7840, 9372.2699, 9491.1593,
9512.4667, 9637.7196, 9745.4137, 9970.3013, 9970.4849
```

All the previous examples were with numbers which could have been handled via `\ifdim` tests rather than the `\xintifCmp` macro from `xintfrac`; using `\ifdim` tests would naturally be faster. Even faster routine is `code 6` at (link removed) which uses `\pdfescapestring` and a Merge Sort algorithm.

We then turn to a graphical illustration of the algorithm.[75] For simplicity the pivot is always chosen as the first list item. Then we also give a variant which picks up the last item as pivot.

```
        % in LaTeX preamble:
        % \usepackage{xintfrac, xinttools}
        % \usepackage{color}
        % or, when using Plain TeX:
        % \input xintfrac.sty \input xinttools.sty
        % \input color.tex
        %
        % Color definitions
        \definecolor{LEFT}{RGB}{216,195,88}
        \definecolor{RIGHT}{RGB}{208,231,153}
        \definecolor{INERT}{RGB}{199,200,194}
        \definecolor{INERTpiv}{RGB}{237,237,237}
        \definecolor{PIVOT}{RGB}{109,8,57}
        % Start of macro defintions
        \makeatletter
        % \catcode`? 3 % a bit too paranoid. Normal ? will do.
        %
        % argument will never be empty
        \def\QS@cmp@a    #1{\QS@cmp@b  #1??}%
```

---

[75] I have rewritten (2015/11/21) the routine to do only once (and not thrice) the needed calls to `\xintifCmp`, up to the price of one additional `\edef`, although due to the context execution time on our side is not an issue and moreover is anyhow overwhelmed by the TikZ's activities. Simultaneously I have updated the code. The variant with the choice of pivot on the right has more overhead: the reason is simply that we do not convert the data into an array, but maintain a list of tokens with self-reorganizing delimiters.

```
\def\QS@cmp@b    #1{\noexpand\QS@sep@A\@ne{#1}\QS@cmp@d {#1}}%
\def\QS@cmp@d    #1#2{\ifx ?#2\expandafter\QS@cmp@done\fi
                     \xintifCmp {#1}{#2}\tw@\@ne\z@{#2}\QS@cmp@d {#1}}%
\def\QS@cmp@done #1?{?}%
%
\def\QS@sep@A #1?{\QSLr\QS@sep@L #1\thr@@?#1\thr@@?#1\thr@@?}%
\def\QS@sep@L #1#2{\ifcase #1{#2}\or\or\else\expandafter\QS@sep@I@start\fi \QS@sep@L}%
\def\QS@sep@I@start\QS@sep@L {\noexpand\empty?\QSIr\QS@sep@I}%
\def\QS@sep@I #1#2{\ifcase#1\or{#2}\or\else\expandafter\QS@sep@R@start\fi\QS@sep@I}%
\def\QS@sep@R@start\QS@sep@I {\noexpand\empty?\QSRr\QS@sep@R}%
\def\QS@sep@R #1#2{\ifcase#1\or\or{#2}\else\expandafter\QS@sep@done\fi\QS@sep@R}%
\def\QS@sep@done\QS@sep@R {\noexpand\empty?}%
%
\def\QS@loop {%
    \xintloop
    % pivot phase
    \def\QS@pivotcount{0}%
    \let\QSLr\DecoLEFTwithPivot  \let\QSIr \DecoINERT
    \let\QSRr\DecoRIGHTwithPivot \let\QSIrr\DecoINERT
    \centerline{\QS@list}%
    % sorting phase
    \ifnum\QS@pivotcount>\z@
            \def\QSLr {\QS@cmp@a}\def\QSRr {\QS@cmp@a}%
            \def\QSIr {\QSIrr}\let\QSIrr\relax
                \edef\QS@list{\QS@list}% compare
            \let\QSLr\relax\let\QSRr\relax\let\QSIr\relax
                \edef\QS@list{\QS@list}% separate
            \def\QSLr ##1##2?{\ifx\empty##1\else\noexpand \QSLr {{##1}##2}\fi}%
            \def\QSIr ##1##2?{\ifx\empty##1\else\noexpand \QSIr {{##1}##2}\fi}%
            \def\QSRr ##1##2?{\ifx\empty##1\else\noexpand \QSRr {{##1}##2}\fi}%
                \edef\QS@list{\QS@list}% gather
            \let\QSLr\DecoLEFT \let\QSRr\DecoRIGHT
            \let\QSIr\DecoINERTwithPivot \let\QSIrr\DecoINERT
            \centerline{\QS@list}%
    \repeat }%
%
% \xintFor* loops handle gracefully empty lists.
\def\DecoLEFT  #1{\xintFor* ##1 in {#1} \do {\colorbox{LEFT}{##1}}}%
\def\DecoINERT #1{\xintFor* ##1 in {#1} \do {\colorbox{INERT}{##1}}}%
\def\DecoRIGHT #1{\xintFor* ##1 in {#1} \do {\colorbox{RIGHT}{##1}}}%
\def\DecoPivot #1{\begingroup\color{PIVOT}\advance\fboxsep-\fboxrule\fbox{#1}\endgroup}%
%
\def\DecoLEFTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
    \xintFor* ##1 in {#1} \do
        {\xintifForFirst {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}}%
\def\DecoINERTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
    \xintFor* ##1 in {#1} \do
        {\xintifForFirst {\colorbox{INERTpiv}{##1}}{\colorbox{INERT}{##1}}}}%
\def\DecoRIGHTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
    \xintFor* ##1 in {#1}  \do
        {\xintifForFirst {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}}%
%
\def\QuickSort #1{% warning: not compatible with empty #1.
    % initialize, doing conversion from comma separated values to a list of braced items
    \edef\QS@list{\noexpand\QSRr{\xintCSVtoList{#1}}}% many \edef's are to follow anyhow
% earlier I did a first drawing of the list, here with the color of RIGHT elements,
% but the color should have been for example white, anyway I drop this first line
```

```
    %\let\QSRr\DecoRIGHT
    %\par\centerline{\QS@list}%
%
    % loop as many times as needed
    \QS@loop }%
%
% \catcode`? 12 % in case we had used a funny ? as delimiter.
\makeatother
%% End of macro definitions.
%% Start of Example
\begingroup\offinterlineskip
\small
% \QuickSort {1.0, 0.5, 0.3, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
%                1.3, 1.1, 0.7, 1.6, 0.6, 0.9, 0.8, 0.2, 0.1, 1.9}
% \medskip
% with repeated values
\QuickSort {1.0, 0.5, 0.3, 0.8, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
                1.3, 1.1, 0.7, 0.3, 1.6, 0.6, 0.3, 0.8, 0.2, 0.8, 0.7, 1.2}
\endgroup
```

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

```
Here is the variant which always picks the pivot as the rightmost element.
\makeatletter
%
\def\QS@cmp@a #1{\noexpand\QS@sep@A\expandafter\QS@cmp@d\expandafter
                  {\romannumeral0\xintnthelt{-1}{#1}}#1??}%
%
\def\DecoLEFTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
    \xintFor* ##1 in {#1} \do
        {\xintifForLast {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}}
\def\DecoINERTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
    \xintFor* ##1 in {#1} \do
        {\xintifForLast {\colorbox{INERTpiv}{##1}}{\colorbox{INERT}{##1}}}}
\def\DecoRIGHTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
    \xintFor* ##1 in {#1}  \do
        {\xintifForLast {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}}
\def\QuickSort #1{%
    % initialize, doing conversion from comma separated values to a list of braced items
    \edef\QS@list{\noexpand\QSLr {\xintCSVtoList{#1}}}% many \edef's are to follow anyhow
    %
    % loop as many times as needed
    \QS@loop }%
\makeatother
\begingroup\offinterlineskip
\small
% \QuickSort {1.0, 0.5, 0.3, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
%                1.3, 1.1, 0.7, 1.6, 0.6, 0.9, 0.8, 0.2, 0.1, 1.9}
% \medskip
```

```
% with repeated values
\QuickSort {1.0, 0.5, 0.3, 0.8, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
            1.3, 1.1, 0.7, 0.3, 1.6, 0.6, 0.3, 0.8, 0.2, 0.8, 0.7, 1.2}
\endgroup
```

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

The choice of the first or last item as pivot is not a good one as nearly ordered lists will take quadratic time. But for explaining the algorithm via a graphical interpretation, it is not that bad. If one wanted to pick up the pivot randomly, the routine would have to be substantially rewritten: in particular the \Deco..withPivot macros need to know where the pivot is, and currently this is implemented by using either \xintifForFirst or \xintifForLast.

This documentation has been compiled without the source code, which is available in the separate file:
sourcexint.pdf,
which will open in a PDF viewer via texdoc sourcexint.pdf.
To produce a single file including both the user documentation and the source code:

- run etex on xint.dtx to generate xint.tex among other files,

- edit xint.tex to set the \NoSourceCode toggle within it to 0,

- run make clean and then make xint.pdf.

This will need latexmk; if not available you will need in replacement of the last step to execute manually
latex on xint.tex (thrice) then dvipdfmx.