

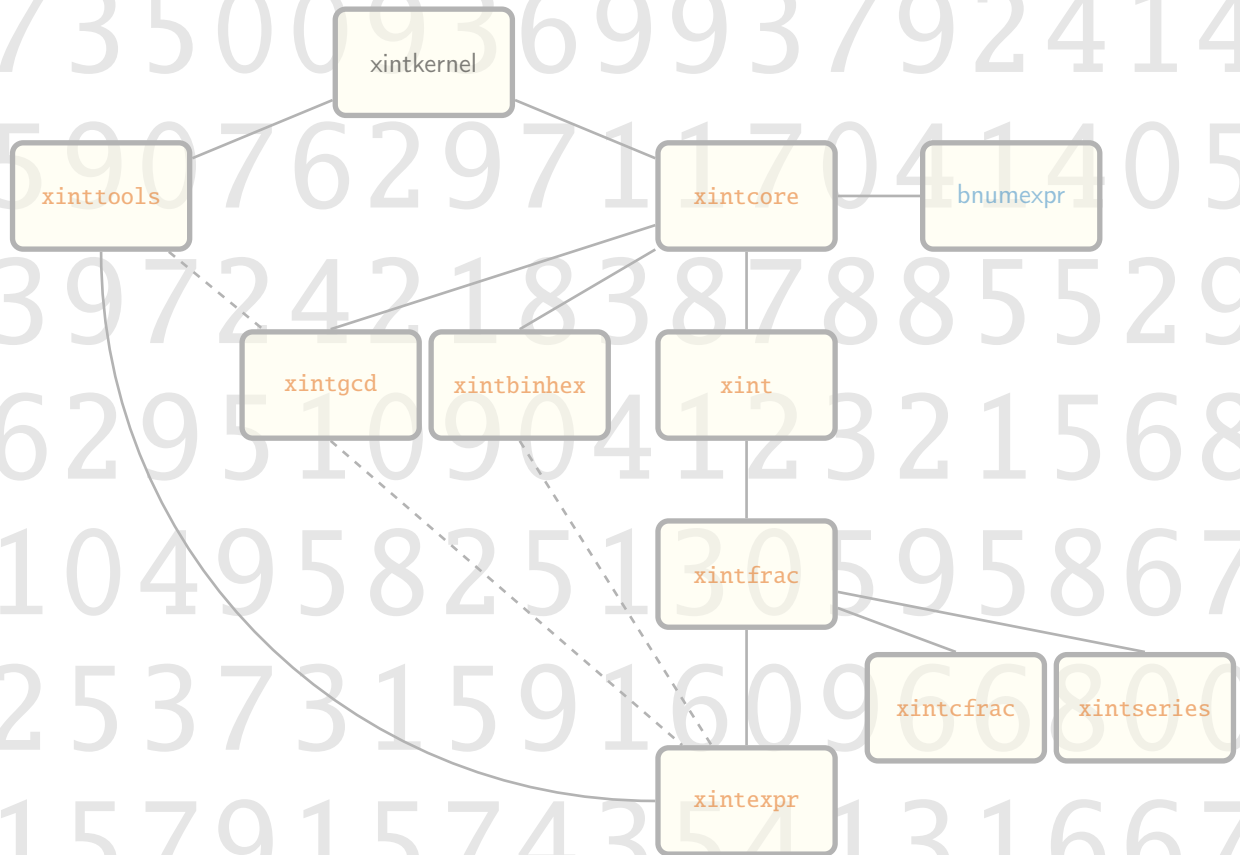
The **xint** bundle

JEAN-FRANÇOIS BURNOL

jfbu (at) free (dot) fr

Package version: 1.1a (2014/11/07); documentation date: 2014/11/07.

From source file xint.dtx. Time-stamp: <07-11-2014 at 23:44:15 CET>.



Dependency graph for the **xint** bundle components: modules at the bottom import modules at the top when connected by a continuous line. No module will be loaded twice, this is managed internally under Plain as well as \LaTeX . Dashed lines indicate a partial dependency, and to enable the corresponding functionalities of the lower module it is necessary for the user to issue the suitable `\usepackage{top_module}` in the preamble (or `\input top_module.sty\relax` in Plain \TeX). The **bnumexpr** package is a separate package (\LaTeX only) by the author.

Contents

1	Read this first	2
1.1	Installation instructions	3
1.2	Presentation of the packages	3
1.3	Changes	4
2	Introduction via examples	7
2.1	Printing big numbers on the page	7
2.2	Randomly chosen examples	8
3	The <code>xint</code> bundle	11
3.1	General overview	11
3.2	Origins of the package	12
4	User interface	15
4.1	Input formats	16
4.2	Output formats	18
4.3	Use of count registers	19
4.4	Dimensions	20
4.5	<code>\ifcase</code> , <code>\ifnum</code> , ... constructs	21
5	Some utilities from the <code>xinttools</code> package	24
5.1	Assignments	24
5.2	Utilities for expandable manipulations	25
6	New features of the <code>xintexpr</code> package	26
6.1	1.1 brought some breaking changes and quite a few novelties	26
7	Commands of the <code>xinttools</code> package	32
8	Commands of the <code>xintcore</code> package	63
9	Commands of the <code>xint</code> package	67
10	Commands of the <code>xintfrac</code> package	75
11	Commands of the <code>xintexpr</code> package	86
12	Commands of the <code>xintbinhex</code> package	107
13	Commands of the <code>xintgcd</code> package	109
14	Commands of the <code>xintseries</code> package	111
15	Commands of the <code>xintcfrac</code> package	125
1.4	User interface	4
1.5	FAQ	6
2.3	More examples, some quite elaborate, within this document	10
3.3	Expansion matters	13
3.4	Efficiency; floating point macros	14
4.6	Expandable implementations of mathematical algorithms	21
4.7	Possible syntax errors to avoid	22
4.8	Error messages	23
4.9	Package namespace, catcodes	23
5.3	A new kind of for loop	25
5.4	A new kind of expandable loop	25
6.2	Examples with the v1.1 <code>\xintexpr</code>	27

1 Read this first

This section provides recommended reading on first discovering the package.

Installation instructions	1.1, p. 3
Presentation of the packages	1.2, p. 3
Changes	1.3, p. 4
User interface	1.4, p. 4
FAQ	1.5, p. 6

1.1 Installation instructions

`xint` is made available under the [LaTeX Project Public License 1.3c](#) (or any later version). It 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.

After installation, issuing in terminal `texdoc --list xint`, on installations with a "texdoc" or similar utility, will offer the choice to display one of the documentation files: `xint.pdf` (this file), `sourcexint.pdf` (source code), `README`, `README.pdf`, `README.html`, `CHANGES.pdf`, and `CHANGES.html`.

For manual installation, follow the instructions from the `README` file which is to be found on [CTAN](#); it is also available there in PDF and HTML formats. The simplest method proposed is to use the archive file `xint.tds.zip`, downloadable from the same location.

The next simplest one is to make use of the `Makefile`, which is also downloadable from [CTAN](#). This is for GNU/Linux systems and Mac OS X, and necessitates use of the command line. If for some reason you have `xint.dtx` but no internet access, you can recreate `Makefile` as a file with this name and the following contents:

```
include Makefile.mk
Makefile.mk: xint.dtx ; etex xint.dtx
```

Then run `make` in a working repertory where there is `xint.dtx` and the file named `Makefile` and having only the two lines above. The `make` will extract the package files from `xint.dtx` and display some further instructions.

If you have `xint.dtx`, no internet access and can not use the `Makefile` method : `etex xint.dtx` extracts all files and among them the `README` as a file with name `README.md`. Further help and options will be found therein.

1.2 Presentation of the packages

The components of the `xint` bundle provide macros dedicated to *expandable* computations on numbers exceeding the \TeX (and $\varepsilon\text{-}\TeX$) limit of `2147483647`.

The $\varepsilon\text{-}\TeX$ extensions (dating back to 1999) must be enabled; this is the case by default in modern distributions, except for the `tex` executable itself which has to be the pure D. KNUTH software with no additions. The name for the extended binary is `etex`. In [TL2014](#) for example `etex` is a symbolic link to the `pdfTeX` executable which will then run in DVI output mode, the $\varepsilon\text{-}\TeX$ extensions being automatically active.

All components may be loaded with \LaTeX `\usepackage` or `\RequirePackage` or, for any other format based on \TeX , directly via `\input`, e.g. `\input xint.sty\relax`. There are no package options. Each package automatically loads those not already loaded it depends on.

xinttools provides utilities of independent interest such as expandable and non-expandable loops. It is `not` loaded automatically (nor needed) by the other bundle packages, apart from `xintexpr`.

xintcore provides the expandable \TeX macros doing additions, subtractions, multiplications, divisions and powers on arbitrarily long numbers (loaded automatically by `xint`, and also by package `bnumexpr` in its default configuration).

xint extends `xintcore` with additional operations on big integers.

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

xintexpr extends `xintfrac` with expandable parsers doing algebra (exact or float, or limited to integers) on comma separated expressions using standard infix notations with parentheses, numbers in decimal notation, and scientific notation, comparison operators, Boolean logic, twofold and threefold way conditionals, sub-expressions, some functions with one or many arguments, user-definable variables, evaluation of sub-expressions over a dummy variable range, with possible recursion, omit, abort, break instructions, nesting.

Further modules:

Changed →

Formerly, →
part of `xint`.

1 Read this first

`xintbinhex` is for conversions to and from binary and hexadecimal bases.
`xintseries` provides some basic functionality for computing in an expandable manner partial sums of series and power series with fractional coefficients.
`xintgcd` implements the Euclidean algorithm and its typesetting.
`xintcffrac` deals with the computation of continued fractions.

1.3 Changes

See file [CHANGES.pdf](#) for the cumulative change log since the initial release.

1.3.1 Changes from 1.1 to 1.1a

Bug fix in `\xintNewExpr` which should have accepted comma separated replacement text.

New `\xintiiSqrtR` for rounded integer square root, the already provided `\xintiiSqrt` computing truncated integral square root.

Numerous small fixes to the documentation.

1.3.2 Changes from 1.09n to 1.1

1.1 brings many extensions, particularly to `xintexpr`, among them the evaluation of expressions with dummy variables, possibly iteratively, in a nestable way. See [section 6](#) for these changes.

Apart from that, the two main other changes are that `\xintiexpr...\relax` now maps `/` to the rounded division (the `//` operator does truncated division) to be in synchrony with the habits of `\numexpr`, and that the `xintfrac` macro `\xintAdd` (corresponding naturally to `+` in expressions) does not anymore blindly multiply denominators but at least checks if one is a multiple of the other. However doing systematic reduction to smallest terms, or only computing always the LCM of the denominators would be too costly.

Also worth mentioning is the fact that `xint` does not load `xinttools` anymore (only `xintexpr` does) and that the core arithmetic macros have been moved to a new package `xintcore` which is loaded automatically by `xint`.

The further package by the author `bnumexpr` (~~TeX~~ only) now loads only `xintcore`.

1.4 User interface

The user interface for executing operations on numbers is via macros such as `\xintAdd` or `\xintMul` which have two arguments, or via expressions `\xintexpr...\relax` which use infix notations such as `+`, `-`, `*`, `/` and `^` (or `**`) for the basic operations, and recognize functions of one or more comma separated arguments (such as `max`, or `round`, or `sqrt`), parentheses, logic operators of conjunction `&&`, disjunction `||`, as well as two-way `?` and three-way `??` conditionals and more. A few examples:

```
\begin{enumerate}[nosep]
\item \xintiiAdd {2719873981798137981381789317981279}{13819093809180120910390190}
\item \xintiiMul {2719873981798137981381789317981279}{13819093809180120910390190}
\item \xintthefloatexpr (19317/21913+2198/9291)^3\relax
\item \xintDigits:=64;\xintthefloatexpr (19317/21913+2198/9291)^3\relax
% Let's compute the inner sum exactly, not as a float, before raising to third power:
\item \xintDigits:=16;\xintthefloatexpr \xintexpr 19317/21913+2198/9291\relax^3\relax
\end{enumerate}
```

- 2719873995617231790561910228371469
- 37586193703616733443988913707927291415140138710427005253010
- 1.397807064706433
- 1.397807064706431686226674924718879749420796705201219736804802622
- 1.397807064706432

In `\xintexpr...\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. Algebraic operations are done *exactly*.

The `\xintfloatexpr` variant is for operations which are done using the precision set via the `\xintDigits:=N`; assignment (default is with significands of 16 digits).

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

The `\xintiexpr` variant is only for big integers, it does not know fractions.

There are some important differences of syntax between `\numexpr` and `\xintiexpr` and variants:

- Contrarily to `\numexpr`, the `\xintiexpr` parser will stop expanding only after having encountered (and swallowed) a mandatory `\relax` token.
- In particular, spaces between digits (and not only around infix operators or parentheses) do not stop `\xintiexpr`, contrarily to the situation with `numexpr`: `\the\numexpr 7 + 3 5\relax` expands (in one step) to `105\relax`, whereas `\xintthe\xintiexpr 7 + 3 5\relax` expands (in two steps) to `42`.
- Also worth mentioning is the fact that `\numexpr -(1)\relax` is illegal. But this is perfectly legal and with the expected result in `\xintiexpr...\relax`.
- Inside an `\edef`, expressions `\xintiexpr...\relax` get fully evaluated, but need the prefix `\xintthe` to get printed or used as arguments to some macros, whereas expansion of `\numexpr` in an `\edef` occurs only if prefixed with `\the` or `\number` (except if in a bigger `\numexpr` which will be the one to have to be prefixed. ...)

For macros such as `\xintAdd` or `\xintMul` the arguments are each subjected to the process of *f*-expansion: repeated expansion of the first token until finding something unexpandable (or being stopped by a space token).

Conversely this process of *f*-expansion always provokes the complete expansion of the package macros and `\xintexpr...\relax` also will expand completely under *f*-expansion, but to a private format; the `\xintthe` prefix allows the computation result either to be passed as argument to one of the package macros,¹ or also end up on the printed page (or in an auxiliary file). To recapitulate, all macros dealing with computations

1. *expand completely under the sole process of repeated expansion of the first token, (and two expansions suffice),²*
2. *apply this f-expansion to each one of their arguments.*

Hence they can be nested one within the other up to arbitrary depths. Conditional evaluations either within the macro arguments themselves, or with branches defined in terms of these macros are made possible via macros such as `\xintifSgn` or `\xintifCmp`.



There is no notion of *declaration of a variable* to `xint`, `xintfrac`, or `xintexpr`. The user employs the `\def`, `\edef`, or `\newcommand` (in \TeX) as usual, for example:

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

macro: `->610296344513856/1[0]` (see below for the `A/B[N]` output format; with `\xintiiMul` in place of `\xintMul` there would not be the strange looking `/1[0]`.)

As a faster alternative to `\edef` (when hundreds of digits are involved), the package provides `\oodef` which only expands twice its argument. This provokes full expansion of the `xint` *f*-expandable macros (nested to possibly many levels), inclusive of `\xintexpr` and variants.

¹ the `\xintthe` prefix *f*-expands the `\xintexpr`-ession then unlocks it from its private format; it should not be used for sub-expressions inside a bigger one as it is more efficient for the expression parser to keep the result in the private format. ² see in [subsection 3.3](#) for more details.

1 Read this first

The `xintexpr` package has a private internal representation for the evaluated computation result. With

```
\oodef\z {\xintexpr 3.141^18\relax}
```

the macro `\z` is already fully evaluated (two expansions were applied, and this is enough), and can be reused in other `\xintexpr`-essions, such as for example

```
\edef\zz {\xintexpr \z+1/\z\relax}
```

% (using short macro names such as `\z` and `\zz` is not too recommended in real

% life, some may have already definitions; I did it all in a group).

But to print it, or to use it as argument to one of the package macros, it must be prefixed by `\xintthe` (a synonym for `\xintthe\xintexpr` is `\xinttheexpr`). Application of this `\xintthe` prefix outputs the value in the `xintfrac` semi-private internal format $A/B[N]$,³ representing the fraction $(A/B) \times 10^N$. The `\zz` above produces a somewhat large output:

```
\printnumber{\xintthe\zz }{\approx}{\xintFloat{\xintthe\zz}}
```

```
784234078359583205345821137151811500398338445353033410108931563956743400014629320996271520032
2210036563468096257420564737421041/8855699172620890267194625578191678465948405196109522070462
879321[-54]  $\approx 8.855699172620890e8$ 
```

By default, computations done by the macros of `xintfrac` or within `\xintexpr...\relax` are exact. Inputs containing decimal points or scientific parts do not make the package switch to a ‘floating-point’ mode. The inputs, however long, are converted into exact internal representations.

Manipulating exactly big fractions quickly leads to ... bigger fractions. There is a command `\xintIrr` (or the function `reduce` in an expression) to reduce to smallest terms, but it has to be explicitly requested. Prior to release 1.1 addition and subtraction blindly multiplied denominators; they now check if one is a multiple of the other. But systematic reduction of the result to its smallest terms would be too costly.

`\xinttheexpr 27/25+46/50\relax` is a bit simpler than `\xinttheexpr (27*50+25*46)/(25*50)\relax`, but less so than `\xinttheexpr reduce(27/25+46/50)\relax`. And `\xinttheexpr 3/75+4/50+2/100\relax` looks weird, but systematically reducing fractions would be too costly.

100/50 is a bit simpler than 2500/1250, but less so than 2. And 52500/375000 looks weird, but systematically reducing fractions would be too costly.

The $A/B[N]$ shape is the output format of most `xintfrac` macros, it benefits from accelerated parsing when used on input, compared to the normal user syntax which has no $[N]$ part. An example of valid user input for a fraction is

```
-123.45602e78/+765.987e-123
```

where both the decimal parts, the scientific exponent parts, and the whole denominator are optional components. The corresponding semi-private form in this case would be

```
-12345602/765987[199]
```

The forward slash `/` is simply a delimiter to separate numerator and denominator, in order to allow inputs having such denominators.

Reduction to the irreducible form of the output must be asked for explicitly via the `\xintIrr` macro or the `reduce` function within `\xintexpr...\relax`. Elementary operations on fractions do very little of the simplifications which could be obvious to (some) human beings.

1.5 FAQ

We are honoured to present here this interview with the author, dating back to late March 2014.

Will `xintexpr` implement `exp`, `log`, `cos`, `sin` ... at some point? I guess so.

`xintseries` already provides generic tools. Right, although the casual user of the `xint` bundle will not quite know how to do variable reduction expandably in order to use some series or Padé

³ there is also the notion of `\xintfloatexpr`, for which the output format after the action of `\xintthe` is a number in floating point scientific notation.

approximants. Besides I wrote the code at the beginning of the project and perhaps I could do it better now (I have not looked at it for a while). Anyhow, generic things do not help much if one wants to optimize.

Optimizing? isn't \TeX 's macro expansion mechanism intrinsically slow? Intensive use of `\numexpr` and some token manipulation algorithms exploiting to the best I could \TeX macros with parameters grant `xint` a significant speed up in expandable arithmetic on big integers compared to previously available implementations. You can do some comparisons with multiplication on numbers with 100 digits or division of one of 100 digits by another of 50 digits, for example. However expandability is antagonist of speed, and I agree it is not very exciting to optimize slow things. And I was disappointed last year to realize the slowness of \TeX 's mouth when it has to keep hundreds of tokens in cheek to mix them later with new aliments. Believe me, I try not to think too much about the fact that the whole enterprise is made irrelevant by Lua \TeX 's ability to access external libraries.

Well, why isn't this log etc. . . thing done yet? I have to decide on the maximal precision to achieve: 24, 32, 48, 64, . . . ; to settle that I would need to implement some initial versions and benchmark them.

Fair enough. That's the common lot. So why not yet? I am a bit overworked. It is also an opportunity to think over the basic underlying mathematics, and will need devoted thinking for some not insignificant amount of time. So far I didn't find the time, or rather I found out good means to waste it sillily. I also anticipate that originality could very well not pay off at all, so small is the window for the precision.

Any chance this could be done in time for TL2014? No, sorry.

Release 1.09m of [2014/02/26] was the end of a cycle, and this 1.09n of [2014/04/01] is only for a bug fix and inclusion of this FAQ in the documentation.

and in time for TL2015? ... (indistinct mumbles, something like *too tired, I need a life, get yourself a calculator, we'll see. . .*)

2 Introduction via examples

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

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

It may be used like this:

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

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

An alternative is to suitably configure the thousand separator with the `numprint` package (see footnote 7. This will not allow linebreaks when used in math mode; I also tried `siunitx` but even in text mode could not get it to break numbers accross lines). Recently I became aware of the `seqsplit` package⁴ which can be used to achieve this splitting accross lines, and does work in inline math mode (however it doesn't allow to separate digits by groups of three, for example).

⁴ <http://ctan.org/pkg/seqsplit>

2.2 Randomly chosen examples

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

- 123456^{99} :

```
\xintiiPow {123456}{99}: 1147381811662665566332733300084545867470254804234261029758895452
43735908946970320276226470542663205834690270868221168133415250032403876277616895322211762
34295872033762216088606915850757168019716710712087697033536507377487778737784987816067492
99979836658125172327521549705416595667384911533326748541075607669718906235189958323778262
36999811095323939932351899922205645878127014958776791431677354372538584459487155941215192
74163986661258969837372587167573949494355201709502618658016651990307184144322311696783762
96
```

- $1234/56789$ with 1500 digits after the decimal point:

```
\xintTrunc {1500}{1234/56789}\dots: 0.021729560302171195125816619415731919914067865255592
52737325890577400552923981757030410818996636672595044815016992727464825934600010565426402
30005810984521650319604148690767578228177992216802549789571924140238426455827713113455072
05242212400288788321682015883357692510873584673088098047157019845392593636091496592649982
50323125957491767771927662047227456021412597510081177692863054464773107467995562520910732
97559386500906865766257549877617144165243268942929088379791861099860888552360492348870372
98270791878708904893553328989769145433094437302998820194051664935110672841571431087006282
64287097853457535790381940164468471006709045765905368997517124795294863441863741217489302
25057669619116378171829051400799450597827043969782880487418338058426808008593213474440472
26267410942259944707601824296958918100336332740495518498300727253517406539998943457359692
99418901547834968039585130923242177182200778319745021042807585976157354417228688654492942
75778759971121167831798411664230748912641532691190195284298015460740636390850340735001492
67687404250823222807233795277254397858740248991882230713694553522689253200443747908926022
44061349909313423374245012238285583475673105707091162020813890013911144763950765112962012
72920812129109510644667101023085456690556269700117980594833506488932715842856891299371352
71290214654246420961805983553152899329095423409463100248287520470513655813625878251069742
94233038088362182817094859920054940217295603021711951258166194157319199140678652555952732
732589057740055292398175703041081899663667. . .
```

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

```
\xinttheexpr trunc(.99^-100,200)\relax\dots: 2.731999026429026003846671721257837435505352
16429385720708334305725082464555187053430448143013784806140368055624765019253070342696852
48915319461661227101592067191384034885148574794308647096392073177979303. . .
```

- Just to show off (again), let's print 300 digits (after the decimal point) of the decimal expansion of 0.7^{-25} :⁵

```
% % in the preamble:
% \usepackage[english]{babel}
% \usepackage[autolanguage,np]{numprint}
% \npthousandsep{,\hspace{1pt} plus .5pt minus .5pt}
% \usepackage{xintexpr}
% in the body:
\np {\xinttheexpr trunc(.7^-25,300)\relax}\dots
```

```
7,456.739,985,837,358,837,609,119,727,341,853,488,853,339,101,579,533,584,812,792,108,
394,305,337,246,328,231,852,818,407,506,767,353,741,490,769,900,570,763,145,015,081,436,
139,227,188,742,972,826,645,967,904,896,381,378,616,815,228,254,509,149,848,168,782,309,
```

⁵ the `\np` typesetting macro is from the `numprint` package.

2 Introduction via examples

405,985,245,368,923,678,816,256,779,083,136,938,645,362,240,130,036,489,416,562,067,450,212,897,407,646,036,464,074,648,484,309,937,461,948,589. . .

This computation is with `\xinttheexpr` from package `xintexpr`, which allows to use standard infix notations and function names to access the package macros, such as here `trunc` which corresponds to the `xintfrac` macro `\xintTrunc`.

The fraction 0.7^{-25} is first evaluated exactly; for some more complex inputs, such as 0.7123045678952^{-243} , the exact evaluation before truncation would be rather costly, and one would rather use floating point numbers:

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

Side note: the exponent -243 didn't have to be put inside parentheses, contrarily to what happens with some professional computational software. ;-)

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

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

$-220045702773594816771390169652074193009609478853 \times (7^{200} - 3^{200}) + 143258949362763693185913026832683204654744168633877140891583816724789919211328201191274624371580391777549768571912287693144240605066991456336143205677696774891 \times (2^{200} - 1) = 1803403947125$

- The Euclide algorithm applied to 22,206,980,239,027,589,097 and 8,169,486,210,102,119,257: (with `xintgcd`)⁶

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

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

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

⁶ this example is computed tremendously faster than the other ones, but we had to limit the space taken by the output hence picked up rather small big integers as input.

0.062366080

The complete series, extended to infinity, has value $\frac{\pi^2}{144} - \frac{1}{162} = 0.062,366,079,945,836,595,346,844,45\dots$ ⁷ I also used (this is a lengthier computation than the one above) `xintseries` to evaluate the sum with 100,000 terms, obtaining 16 correct decimal digits for the complete sum. The coefficient macro must be redefined to avoid a `\numexpr` overflow, as `\numexpr` inputs must not exceed $2^{31} - 1$; my choice was:

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

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

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

where the `numprint` package was used (footnote 7), directly in text mode (it can also naturally be used from inside math mode). `xint` provides a simple-minded `\xintFrac` typesetting macro,⁸ which is math-mode only:

```
$_\xintFrac{\xintFloatPow [24]{2}{999999999}}$
230648800058453469655806 · 10301029972
```

The exponent differs, but this is because `\xintFrac` does not use a decimal mark in the significant of the output. Admittedly most users will have the need of more powerful (and customizable) number formatting macros than `\xintFrac`.⁹ We have already mentioned `\numprint` which is used above, there is also `\num` from package `siunitx`. The raw output from

```
\xintFloatPow [24]{2}{999999999}
```

is 2.30648800058453469655806e301029995.

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

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

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

2.3 More examples, some quite elaborate, within this document

- The utilities provided by `xinttools` (section 7), some completely expandable, others not, are of independent interest. Their use is illustrated through various examples: among those, it is shown in subsection 7.30 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 7.13), another one with `\xintFor*` (subsection 7.23).

⁷ This number is typeset using the `numprint` package, with `\npthousandsep {,}\hskip 1pt plus .5pt minus .5pt`. But the breaking across lines works only in text mode. The number itself was (of course...) computed initially with `xint`, with 30 digits of π as input. See [how xint may compute \$\pi\$ from scratch](#). ⁸ Plain T_EX users of `xint` have `\xintFwOver`. ⁹ There should be a `\xintFloatFrac`, but it is lacking.

- One has also a `computation of primes within an \edef` (subsection 7.15), with the help of `\xintilooop`. Also with `\xintilooop` an automatically generated table of factorizations (subsection 7.17).
- The code for the title page fun with Fibonacci numbers is given in subsection 7.24 with `\xint-For*` joining the game.
- The computations of π and $\log 2$ (subsection 14.11) using `xint` and the computation of the `convergents of e` with the further help of the `xintcfrac` package are among further examples.
- There is also an example of an `interactive session`, where results are output to the log or to a file.
- The new functionalities of `xintexpr` are illustrated with various examples in section 6.

Almost all of the computational results interspersed throughout the documentation are not hard-coded in the source of the document. They are the result of evaluation of the package macros, and were selected to not impact too much the compilation time of this documentation. Nevertheless, there are so many computations done that compilation time is nevertheless significantly increased compared to a \TeX run on a typical document of about the same size.

3 The `xint` bundle

3.1 General overview

The main characteristics are:

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

‘Arbitrarily big’ means with less than $2^{31}-1=2147483647$ digits, as most of the macros will have to compute the length of the inputs and these lengths must be treatable as \TeX integers, which are at most 2147483647 in absolute value. This is a distant irrelevant upper bound, as no such thing can fit in \TeX ’s memory! And besides, the true limitation is from the *time* taken by the expansion-compatible algorithms, as will be commented upon soon.

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

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

The `bigintcalc` package by HERKO OBERDIEK provided expandable operations (using some of `\numexpr` possibilities, when available) on arbitrarily big integers, beyond the \TeX bound. The present package does this again, using more of `\numexpr` (`xint` requires the ε - \TeX extensions) for higher speed, and also on fractions, not only integers. Arbitrary precision floating points operations are a derivative, and not the initial design goal.^{10, 11}

¹⁰ currently (v1.08), the only non-elementary operation implemented for floating point numbers is the square-root extraction; no signed infinities, signed zeroes, NaN’s, error trapes... have been implemented, only the notion of ‘scientific notation with a given number of significant figures’.¹¹ multiplication of two floats with $P=\text{\code{xinttheDigits}}$ 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.)

The \TeX 3 project has implemented expandably floating-point computations with 16 significant figures (*l3fp*), including special functions such as exp, log, sine and cosine.¹² There is also *l3bigint*, which (a.t.t.o.w.) is part of the experimental trunk of the \TeX 3 Project. Like *bigintcalc* and *xint* it provides macros for big integer arithmetics. All three implementations of the basic arithmetic macros can be mapped to easier infix notations via the services of the *bnumexpr* package.

The *xint* package can be used for 24, 40, etc. . . significant figures but one rather quickly (not much beyond 100 figures) hits against a ‘wall’ created by the constraint of expandability: currently, multiplying out two one-hundred digits numbers takes circa 80 or 90 times longer than for two ten-digits numbers, which is reasonable, but multiplying out two one-thousand digits numbers takes more than 500 times longer than for two one hundred-digits numbers. This shows that the algorithm is drifting from quadratic to cubic in that range. On my laptop multiplication of two 1000-digits numbers takes some seconds, so it can not be done routinely in a document.¹³

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

The second conclusion, somewhat depressing after all the hard work, is that if one really wants to do computations with *hundreds* of digits, one should drop the expandability requirement. And indeed, as clearly demonstrated long ago by the *pi computing file* by D. ROEGEL one can program \TeX to compute with many digits at a much higher speed than what *xint* achieves: but, direct access to memory storage in one form or another seems a necessity for this kind of speed and one has to renounce at the complete expandability.^{14 15}

3.2 Origins of the package

Package *bigintcalc* by HEIKO OBERDIEK already provides expandable arithmetic operations on ‘big integers’, exceeding the \TeX limits (of $2^{31} - 1$), so why another¹⁶ 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.¹⁷ What I had learned in this other thread thanks to interaction with ULRICH DIEZ and GL on expandable manipulations of tokens motivated me to try my hands at addition and multiplication.

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

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

The present package is the result of this initial questioning.

¹² at the time of writing the *l3fp* (exactly represented) floating point numbers have their exponents limited to ± 9999 . ¹³ without entering into too much technical details, the source of this ‘wall’ is that when dealing with two long operands, when one wants to pick some digits from the second one, one has to jump above all digits constituting the first one, which can not be stored away: expandability forbids assignments to memory storage. One may envision some sophisticated schemes, dealing with this problem in less naive ways, trying to move big chunks of data higher up in the input stream and come back to it later, etc...; but each ‘better’ algorithm adds overhead for the smaller inputs. For example, I have another version of addition which is twice faster on inputs with 500 digits or more, but it is slightly less efficient for 50 digits or less. This ‘wall’ dissuaded me to look into implementing ‘intelligent’ multiplication which would be sub-quadratic in a model where storing and retrieving from memory would not cost much. ¹⁴ I could, naturally, be proven wrong! ¹⁵ The Lua \TeX project possibly makes endeavours such as *xint* appear even more insane that they are, in truth. ¹⁶ this section was written before the *xintfrac* package; the author is not aware of another package allowing expandable computations with arbitrarily big fractions. ¹⁷ the *\ReverseOrder* could be avoided in that circumstance, but it does play a crucial rôle here.

3.3 Expansion matters

By convention in this manual f -expansion (‘‘full expansion’’ or ‘‘full first expansion’’) is the process of expanding repeatedly the first token seen until hitting against something not further expandable like an unexpandable \TeX -primitive or an opening brace `{` or a character (inactive). For those familiar with \TeX 3 (which is not used by `xint`) this is what is called in its documentation full expansion. Technically, macro arguments in `xint` which are submitted to such a f -expansion are so via prefixing them with `\romannumeral-`0`. An explicit or implicit space token stops such an expansion and is gobbled. Most of the package macros, and all those dealing with computations, are expandable in the strong sense that they expand to their final result via this f -expansion. Again copied from \TeX 3 documentation conventions, this will be signaled in the description of the macro by a star in the margin. All¹⁸ expandable macros of the `xint` packages completely expand in two steps.

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

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

A few macros from `xinttools` do not expand, or expand only once their argument. This is also signaled in the margin with notations à la \TeX 3.

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

Conversely, wherever the package expects on input a ‘‘big’’ integers, or a ‘‘fraction’’, f -expansion of the argument *must result in a complete expansion* for this argument to be acceptable.¹⁹ The main exception is inside `\xintexpr...\relax` where everything will be expanded from left to right, completely.

Summary of important expansion aspects:

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

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

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

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

2. using `\if...\fi` constructs *inside* the package macro arguments requires suitably mastering \TeX niques (`\expandafter`’s and/or swapping techniques) to ensure that the f -expansion will indeed absorb the `\else` or closing `\fi`, else some error will arise in further processing. Therefore it is highly recommended to use the package provided conditionals such as `\xintifEq`,

¹⁸ except `\xintloop` and `\xintilloop`. ¹⁹ this is not quite as stringent as claimed here, see subsection 4.3 for more details.

`\xintifGt`, `\xintifSgn`, `\xintifOdd`... , or, for \TeX users and when dealing with short integers the `etoolbox`²⁰ expandable conditionals (for small integers only) such as `\ifnumequal`, `\ifnumgreater`, ... Use of *non-expandable* things such as `\ifthenelse` is impossible inside the arguments of `xint` macros.

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

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

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

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

4. With the definition

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

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

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

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

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

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

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

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

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

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

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

3.4 Efficiency; floating point macros

The size of the manipulated numbers is limited by two factors:²¹ (1.) *the available memory as configured in the `tex` executable*, (2.) *the time necessary to fully expand the computations themselves*. The most limiting factor is the second one, the time needed (for multiplication and division, and even more for powers) explodes with increasing input sizes long before the computations could get limited by constraints on \TeX ’s available memory: computations with 100 digits are still reasonably fast, but the situation then deteriorates swiftly, as it takes of the order of seconds (on my laptop) for the package to multiply exactly two numbers each of 1000 digits and it would take hours for numbers each of 20000 digits.²²

To address this issue, floating point macros are provided to work with a given arbitrary precision. The default size for significands is 16 digits. Working with significands of 24, 32, 48, 64,

²⁰ <http://www.ctan.org/pkg/etoolbox> ²¹ there is an intrinsic limit of `2147483647` on the number of digits, but it is irrelevant, in view of the other limiting factors. ²² Perhaps some faster routines could emerge from an approach which, while maintaining expandability would renounce at *f*-expandability (without impacting the input save stack). There is one such routine `\xintXTrunc` which is able to write to a file (or inside an `\edef`) tens of thousands of digits of a (reasonably-sized) fraction.

or even 80 digits is well within the reach of the package. But routine multiplications and divisions will become too slow if the precision goes into the hundreds, although the syntax to set it (`\xintDigits:=P;`) allows values up to 32767.²³ The exponents may be as big as ± 2147483647 .²⁴

Here is such a floating point computation:

```
\xintFloatPower [48] {1.1547}{\xintiPow {2}{35}}
```

which thus computes $(1.1547)^{2^{35}} = (1.1547)^{34359738368}$ to be approximately

```
2.785,837,382,571,371,438,495,789,880,733,698,213,205,183,990,48 × 102,146,424,193
```

Notice that 2^{35} exceeds $\text{T}_{\text{E}}\text{X}$'s bound, but `\xintFloatPower` allows it, what counts is the exponent of the result which, while dangerously close to 2^{31} is not quite there yet. The printing of the result was done via the `\numprint` command from the `numprint` package²⁵.

The same computation can be done via the non-expandable assignment `\xintDigits:=48;` and then

```
\xintthefloatexpr 1.1547^(2^35)\relax
```

Notice though that `2^35` will be evaluated as a floating point number, and if the floating point precision had been too low, this computation would have given an inexact value. It is safer, and also more efficient to code this as:

```
\xintthefloatexpr 1.1547^\xintiexpr 2^35\relax\relax
```

The `\xintiexpr` is a cousin of `\xintexpr` which is big integer-only and skips the overhead of fraction management. Notice on this example that being embedded inside the `floatexpr`-ession has nil influence on the `iiexpr`-ession: expansion proceeds in exactly the same way as if it had been at the 'top' level.

`xintexpr` provides no implementation of the IEEE standard: no NaNs, signed infinities, signed zeroes, error traps, . . . ; what is achieved though is exact rounding for the basic operations. The only non-algebraic operation currently implemented is square root extraction. The power functions (there are three of them: `\xintPow` to which `^` is mapped in `\xintexpr.\relax`, `\xintFloatPower` for `^` in `\xintfloatexpr.\relax`, and `\xintFloatPow` which is slightly faster but limits the exponent to the $\text{T}_{\text{E}}\text{X}$ bound) allow only integral exponents.

4 User interface

Maintaining complete expandability is not for the faint of heart as it excludes doing macro definitions in the midst of the computation; in many cases, one does not need complete expandability, and definitions are allowed. In such contexts, there is no declaration for the user to be made to the package of a "typed variable" such as a long integer, or a (long) fraction, or possibly an `\xi\ntexpr`-ession. Rather, the user has at its disposal the general tools of the $\text{T}_{\text{E}}\text{X}$ language: `\def` and `\edef`. In $\text{M}_{\text{E}}\text{T}_{\text{E}}\text{X}$ there is `\newcommand` as wrapper to `\def`, but $\text{M}_{\text{E}}\text{T}_{\text{E}}\text{X}$ chose not to provide an analogous wrapper for `\edef`. It can still be used directly of course.²⁶

The `xinttools` package provides `\oodef` which expands twice the replacement text²⁷, hence forces complete expansion when the top level of this replacement text is a call to one of the `xint` bundle macros, its arguments being themselves chains of such macros. There is also `\fdef` which will apply *f*-expansion to the replacement text. Both are in such uses faster alternatives to `\edef`.

This section will explain the various inputs which are recognized by the package macros and the format for their outputs. Inputs have mainly five possible shapes:

1. expressions which will end up inside a `\numexpr.\relax`,
2. long integers in the strict format (no `+`, no leading zeroes, a count register or variable must be prefixed by `\the` or `\number`)

²³ for a one-shot conversion of a fraction to float format, or one addition, a precision exceeding 32767 may be passed as optional argument to the used macro. ²⁴ almost . . . as inner manipulations may either add or subtract the precision value to the exponent, arithmetic overflow may occur if the exponents are a bit too close to the $\text{T}_{\text{E}}\text{X}$ bound ± 2147483647 .

²⁵ <http://ctan.org/pkg/numprint> ²⁶ I don't know if $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}3$ will still allow direct use of `\def` and `\edef`. . . ²⁷ only for parameter less unlimited macros.

3. long integers in the general format allowing both `-` and `+` signs, then leading zeroes, and a count register or variable without prefix is allowed,
4. fractions with numerators and denominators as in the previous item, or also decimal numbers, possibly in scientific notation (with a lowercase `e`), and also optionally the semi-private `A/B[N]` format,
5. and finally expandable material understood by the `\xintexpr` parser.

Outputs are mostly of the following types:

1. long integers in the strict format,
2. fractions in the `A/B[N]` format where `A` and `B` are both strict long integers, and `B` is positive,
3. numbers in scientific format (with a lowercase `e`),
4. the private `\xintexpr` format which needs the `\xintthe` prefix in order to end up on the printed page (or get expanded in the log) or be used as argument to the package macros.

Input formats	4.1, p. 16
Output formats	4.2, p. 18
Use of count registers	4.3, p. 19
Dimensions	4.4, p. 20
<code>\ifcase</code> , <code>\ifnum</code> , ... constructs	4.5, p. 21
Expandable implementations of mathematical algorithms	4.6, p. 21
Possible syntax errors to avoid	4.7, p. 22
Error messages	4.8, p. 23
Package namespace, catcodes	4.9, p. 23

4.1 Input formats

`num`
`x`

Some macro arguments are by nature ‘short’ integers, *i.e.* less than (or equal to) in absolute value 2,147,483,647. This is generally the case for arguments which serve to count or index something. They will be embedded in a `\numexpr..\relax` hence on input one may even use count registers or variables and expressions with infix operators. Notice though that `-(..stuff..)` is surprisingly not legal in the `\numexpr` syntax!

But `xint` is mainly devoted to big numbers; the allowed input formats for ‘long numbers’ and ‘fractions’ are:

- `f` 1. the strict format is for some macros of `xint` which only *f*-expand their arguments. After this *f*-expansion the input should be a string of digits, optionally preceded by a unique minus sign. The first digit can be zero only if the number is zero. A plus sign is not accepted. `-0` is not legal in the strict format. A count register can serve as argument of such a macro only if prefixed by `\the` or `\number`. Most macros of `xint` are like `\xintAdd` and accept the extended format described in the next item; they may have a ‘strict’ variant such as `\xintiiAdd` which remains available even with `xintfrac` loaded, for optimization purposes.

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

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

`Num`
`f`

The extended integer format is thus for the arithmetic macros of `xint` which automatically parse their arguments via this `\xintNum`.²⁸

²⁸ A `\value{countname}` is accepted as macro argument.

Frac
f

3. the fraction format is what is expected by the macros of `xintfrac`: a fraction is constituted of a numerator `A` and optionally a denominator `B`, separated by a forward slash `/` and `A` and `B` may be macros which will be automatically given to `\xintNum`. Each of `A` and `B` may be decimal numbers (the decimal mark must be a `.`). Here is an example:²⁹

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

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

```
\xintAdd{10.1e1}{101.010e3}=101111/1[0]
\xintFloatAdd{10.1e1}{101.010e3}=1.011110000000000e5
\xintPow {2}{100}=1267650600228229401496703205376/1[0]
\xintFloat{\xintPow {2}{100}}=1.267650600228229e30
\xintFloatPow {2}{100}=1.267650600228229e30
```

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

```
\xintPraw{\xintAdd{10.1e1}{101.010e3}}=101111
\xintRaw{1.234e5/6.789e3}=1234/6789[2]
```

4. the `expression format` is for inclusion in an `\xintexpr...\relax`, it uses infix notations, function names, complete expansion, and is described in [section 6](#) and [section 11](#).

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

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

Even with `xintfrac` loaded, some macros by their nature can not accept fractions on input. Those parsing their inputs through `\xintNum` will now accept fractions, truncating them first to integers.

With `xintfrac` loaded, a number may be empty or start directly with a decimal point:

```
\xintRaw{ }=\xintRaw{.}=0/1[0]
\xintPow{-.3/.7}{11}=-177147/1977326743[0]
\xinttheexpr (-.3/.7)^11\relax=-177147/1977326743
```

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

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

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

Num
f

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

Frac
f

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

²⁹ the square brackets one sees in various outputs are explained near the end of this section.

The `xintfrac` macros generally output their result in $A/B[n]$ format, representing the fraction A/B times 10^n .

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

It is allowed for user input but the parsing is minimal and it is mandatory to follow the above rules. This reduced flexibility, compared to the format without the square brackets, allows nesting package macros without too much speed impact.

4.2 Output formats

With package `xintfrac` loaded, the routines `\xintAdd`, `\xintSub`, `\xintMul`, `\xintPow`, initially synonyms in `xint` of `\xintiAdd`, `\xintiSub`, `\xintiMul`, `\xintiPow`, are modified to become the fraction handling routines.^{30 31} They produce on output a fractional number $f=A/B[n]$ where A and B are integers, with B positive, and n is a ‘‘short’’ integer. This represents (A/B) times 10^n .³²

The fraction output format for most `xintfrac` macros is $A/B[n]$ which stands for $(A/B) \times 10^n$. The A and B may end in zeroes (i.e., n does not represent all powers of ten), and will generally have a common factor. The denominator B is always strictly positive. Conversely, this format is accepted on input and is parsed more quickly than fractions containing decimal points or in scientific notation; the input denominator is optional.

- A macro `\xintFrac` is provided for the typesetting (math-mode only) of such a ‘raw’ output. The command `\xintFrac` is not accepted as input to the package macros, it is for typesetting only (in math mode).
- `\xintRaw` prints the fraction directly as its internal representation $A/B[n]$.

```
$\xintRaw{273.3734e5/3395.7200e-2}=\xintFrac {273.3734e5/3395.7200e-2}$
```

$2733734/33957200[7] = \frac{2733734}{33957200} 10^7$

- `\xintPraw` does the same but without printing the $[n]$ if $n=0$ and without printing $/1$ if $B=1$.
- `\xintIrr` reduces the fraction to its irreducible form C/D (without a trailing $[0]$), and it prints the D even if $D=1$.

```
$\xintIrr{273.3734e5/3395.7200e-2}$
```

$2971450000/3691$

- `\xintNum` from package `xint` becomes when `xintfrac` is loaded a synonym to its macro `\xintTTrunc` (same as `\xintiTrunc{0}`) which truncates to the nearest integer.
- See also the documentations of `\xintTrunc`, `\xintiTrunc`, `\xintXTrunc`, `\xintRound`, `\xintiRound` and `\xintFloat`.
- The `\xintiAdd`, `\xintiSub`, `\xintiMul`, `\xintiPow` macros and some others accept fractions on input which they truncate via `\xintTTrunc`. On output they still produce an integer with no fraction slash nor trailing $[n]$.

³⁰ the power function does not accept a fractional exponent. Or rather, does not expect, and errors will result if one is provided.

³¹ as commented upon more later, for that very reason use of `\xintAdd` etc... when only `xint` is loaded is strongly discouraged.

³² at each stage of the computations, the sum of n and the length of A , or of the absolute value of n and the length of B , must be kept less than $2^{31}-9$.

- The `\xintiiAdd`, `\xintiiSub`, `\xintiiMul`, `\xintiiPow`, and others with ‘ii’ in their names accept on input only integers in the strict format (they skip the overhead of the `\xintNum` parsing) and naturally they output an integer, with no fraction slash nor trailing [n].

Some macros return a token list of two or more numbers or fractions; they are then each enclosed in braces. Examples are `\xintiDivision` which gives first the quotient and then the remainder of euclidean division, `\xintBezout` from the `xintgcd` package which outputs five numbers, `\xintFtoCv` from the `xintfrac` package which returns the list of the convergents of a fraction, ... [subsection 5.1](#) and [subsection 5.2](#) mention utilities, expandable or not, to cope with such outputs.

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

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

This returns a comma separated list, with a space after each comma.

4.3 Use of count registers

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 eight tokens*.³³ The slash for rounded division in a `\numexpr` should be written with braces `{/}` to not be confused with the `xintfrac` delimiter between numerator and denominator (braces will be removed internally). Example: `\mycountA+\mycountB{/}127/1+\mycountA*\mycountB`, or `\count 0+\count 2{/}17/1+\count 0*\count 2`, but in the latter case the numerator has the maximal allowed number of tokens (the braced slash counts for only one).

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

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

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

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

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

³³ Attention! there is no problem with a \LaTeX `\value{countname}` if it comes first, but if it comes later in the input it will not get expanded, and braces around the name will be removed and chaos will ensue inside a `\numexpr`. One should enclose the whole input in `\the\numexpr...\relax` in such cases.

4.4 Dimensions

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

For \TeX users: a length is a $\langle\text{glue}\rangle$ variable, prefixing a length command defined by `\newlength` with `\number` will thus discard the `plus` and `minus` glue components and return the dimension component as described above, and usable in the `xint` bundle macros.

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` (it thus appears that \TeX truncates to get an integral multiple of the `sp` unit).

Unit	definition	Exact value in <code>sp</code> units	\TeX 's value in <code>sp</code> 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 \TeX Book, `1157dd=1238pt`. The actual internal value of `1dd` in \TeX is `70124sp`. We can use `xintcfac` 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 `1357dd=1452pt` is *far more accurate* than the \TeX Book formula `1157dd=1238pt`! The formula to compute `Ndd` was

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

What's the catch? The catch is that \TeX does not compute `1157dd` 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, \eTeX , but one can check that this works the same in \TeX82), uses indeed `1238/1157` as a conversion factor, and necessarily intermediate computations

are done with more precision than is possible with only integers less than 2^{31} (or 2^{30} for dimensions). Hence the 1452/1357 ratio is irrelevant, a misleading artefact of the necessary rounding (or, as we see, truncating) for one `dd` 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 *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 $\text{T}_{\text{E}}\text{X}$ points.

The actual value in millimeters of exactly one Didot point as implemented in $\text{T}_{\text{E}}\text{X}$ is

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

The difference of circa 5 Å is arguably tiny!

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

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

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, 1885709281/176233151, 543564351/508000000. We do recover the 1238/1157 therein!

4.5 `\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 $\text{T}_{\text{E}}\text{X}$ to stop its scanning for a number: once $\text{T}_{\text{E}}\text{X}$ has finished expanding `\xintSgn{A}` and has so far obtained either 1, 0, or -1, a space (or something 'unexpandable') must stop it looking for more digits. Using `\ifcase\xintSgnA` without the braces is very dangerous, because the blanks (including the end of line) following `A` will be skipped and not serve to stop the number which `\ifcase` is looking for.

```
\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 $\text{T}_{\text{E}}\text{X}$ technical expertise (see also [item 2](#) on page 13).

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

If these tests are to be applied to standard $\text{T}_{\text{E}}\text{X}$ short integers, it is more efficient to use (under $\text{W}_{\text{E}}\text{T}_{\text{E}}\text{X}$) the equivalent conditional tests from the `etoolbox`³⁴ package.

4.6 Expandable implementations of mathematical algorithms

It is possible to chain `\xintexpr`-essions with `\expandafter`'s, like experts do with `\numexpr` to compute multiple things at once. See [subsection 7.24](#) for an example devoted to Fibonacci numbers (this section provides the code which was used on the title page for the `F(1250)` evaluation.) Notice that the 47th Fibonacci number is 2971215073 thus already too big for $\text{T}_{\text{E}}\text{X}$ and $\varepsilon\text{-T}_{\text{E}}\text{X}$.

³⁴ <http://www.ctan.org/pkg/etoolbox>

The `\Fibonacci` macro found in [subsection 7.24](#) is completely expandable, (it is even *f*-expandable in the sense previously explained) hence can be used for example within `\message` to write to the log and terminal.

Also, one can thus 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 {\Fibonacci {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 need³⁵

```
$\xintiigcd{\Fibonacci{1859}}{\Fibonacci{1573}}=\Fibonacci{\xintiigcd{1859}{1573}}$
```

which outputs:

```
343358302784187294870275058337 = 343358302784187294870275058337
```

The `\Fibonacci` 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...).

In practice, whenever one typesets things, one has left the expansion only contexts; hence there is no objection to, on the contrary it is recommended, assign the result of earlier computations to macros via an `\edef` (or an `\oodef`, see [7.29](#)), for later use. The above could thus be coded

```
\begingroup
\def\A {1859} \def\B {1573} \edef\C {\xintiigcd\A\B}
\edef\X {\Fibonacci\A} \edef\Y {\Fibonacci\B}
The identity $\gcd(F(\A),F(\B))=F(\gcd(\A,\B))$ can be checked via evaluation
of both sides: $\gcd(F(\A),F(\B))=\gcd(\printnumber\X,\printnumber\Y)=
\printnumber{\xintiigcd\X\Y} = F(\gcd(\A,\B))$. \par
% some further computations involving \A, \B, \C, \X, \Y
\endgroup % closing the group removes assignments to \A, \B, ...
% or choose longer names less susceptible to overwriting something. Note that 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: $\gcd(F(1859), F(1573)) = \gcd(14405827913044251198771689151504042869913161495023481014222668636701088272597575494722482437753529619459794869227357628882216309358018264080851775319974256956055294350288615852451737250886736422284929082289524558388949544219265576041299929025565979711337876105452217623490841529979811413199660087517689703410997520079993610707576019520876324584695551467505894985013610208598628752325727241, 2443841925195118573328279459776261998539902481570619232605360900784013394036743212445223278959909515869581103189177976905803274151632595307616686661013725200866754096569888951010022888016831459347310131566517721593249344798634399479371195758766544765827958909282390070313197135548122004938644531329524847747273166471511289078393) = 343358302784187294870275058337 = F(143) = F(\gcd(1859, 1573))$.

One may thus 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`? the answer is that the author chose, seemingly, at some point back in his past to waste from then on his time on such useless things!

4.7 Possible syntax errors to avoid

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

- using `-` to prefix some macro: `-\xintiSqr{35}/271`.³⁶
- using one pair of braces too many `\xintIrr{{\xintiPow {3}{13}}/243}` (the computation goes through with no error signaled, but the result is completely wrong).
- using `[]` and decimal points at the same time `1.5/3.5[2]`, or with a sign in the denominator `3/-5[7]`. The scientific notation has no such restriction, the two inputs `1.5/-3.5e-2` and `-1.5e2/3.5` are equivalent: `\xintRaw{1.5/-3.5e-2}=-15/35[2]`, `\xintRaw{-1.5e2/3.5}=-15/35[2]`.

³⁵ The `\xintiigcd` macro is provided by the `xintgcd` package. ³⁶ to the contrary, this *is* allowed inside an `\xintexpr`-ession.

- specifying numerators and denominators with macros producing fractions when `xintfrac` is loaded: `\edef\x{\xintMul {3}{5}/\xintMul{7}{9}}`. This expands to `15/1[0]/63/1[0]` which is invalid on input. Using this `\x` in a fraction macro will most certainly cause a compilation error, with its usual arcane and undecipherable accompanying message. The fix here would be to use `\xintiMul`. The simpler alternative with package `xintexpr`: `\xinttheexpr 3*5/(7*9)\relax`.
- generally speaking, using in a context expecting an integer (possibly restricted to the \TeX bound) a macro or expression which returns a fraction: `\xinttheexpr 4/2\relax` outputs `4/2`, 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`.
- use of square brackets `[,]` in `\xintexpr...name` has some traps, see [section 11](#).

4.8 Error messages

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

<code>\xintError:ArrayIndexIsNegative</code>	<code>\xintError:TooBigDecimalSplit</code>
<code>\xintError:ArrayIndexBeyondLimit</code>	<code>\xintError:RootOfNegative</code>
<code>\xintError:FactorialOfNegativeNumber</code>	<code>\xintError:NoBezoutForZeros</code>
<code>\xintError:FactorialOfTooBigNumber</code>	<code>\xintError:ignored</code>
<code>\xintError:DivisionByZero</code>	<code>\xintError:removed</code>
<code>\xintError:NaN</code>	<code>\xintError:inserted</code>
<code>\xintError:FractionRoundedToZero</code>	<code>\xintError:unknownfunction</code>
<code>\xintError:NotAnInteger</code>	<code>\xintError:we_are_doomed</code>
<code>\xintError:ExponentTooBig</code>	<code>\xintError:missing_xintthe!</code>
<code>\xintError:TooBigDecimalShift</code>	

Don't forget to set `\errorcontextlines` to at least `2` to get from \TeX more meaningful error messages. Errors occurring during the parsing of `\xintexpr-essions` try to provide helpful information about the offending token.

Release `1.1` employs in some situations delimited macros and there is the possibility in case of an ill-formed expression to end up beyond the `\relax` end-marker. The errors inevitably arising could then lead to very cryptic messages; but nothing unusual or especially traumatizing for the daring experienced \TeX / \LaTeX user.

4.9 Package namespace, catcodes

The `xint` bundle packages presuppose that the `\space`, `\empty` and `\m@ne` control sequences are pre-defined with meanings as in Plain \TeX or \LaTeX 2 ϵ .

Private macros of `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).

`xinttools` provides `\odef`, `\oodef`, `\fdef` (if macros with these names already exist `xinttools` will not overwrite them but provide `\xintodef` etc...) but all other public macros from the `xint` bundle packages start with `\xint`.

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

If a character used in the `\xintexpr` syntax is made active, this will surely cause problems; prefixing it with `\string` is one option. There is `\xintexprSafeCatcodes` and `\xintexprRestoreCatcodes` to temporarily turn off potentially active characters (but setting catcodes is an un-expandable action).

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 Some utilities from the *xinttools* package

This is a first overview. Many examples combining these utilities with the arithmetic macros of *xint* are to be found in [section 7](#).

5.1 Assignments

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

```
\xintAssign\xintiDivision{100}{3}\to\A\B
```

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

give: `\meaning\A: macro:->1148132496415075054822783938725510662598055177841861728836634780659826541894704737970419535798876630484358265060061503749531707793118627774829601` and `\meaning\B: macro:->54936294521339832251381287862239128073410500498476050595321899612313276649022883898132878702444582075129603152041054804964625083138567652624386837205668069376`. Another example (which uses `\xintBezout` from the *xintgcd* package):

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

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

Thus, what `\xintAssign` does is to first apply an *f-expansion* to what comes next; it then defines one after the other (using `\def`; an optional argument allows to modify the expansion type, see [subsection 7.26](#) for details), the macros found after `\to` to correspond to the successive braced contents (or single tokens) located prior to `\to`.

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

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

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

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

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

```
% \newcount\cnta
% \newcount\cntb
```

```

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

```

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

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

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

5.2 Utilities for expandable manipulations

The package now has more utilities to deal expandably with ‘lists of things’, which were treated un-expandably in the previous section with \xintAssign and \xintAssignArray : \xintReverseOrder and \xintLength since the first release, \xintApply and \xintListWithSep since 1.04, $\text{\xintRevWithBraces}$, \xintCSVtoList , \xintNthElt since 1.06, $\text{\xintApplyUnbraced}$, since 1.06b, \xintloop and \xintiloop since 1.09g.³⁷

As an example the following code uses only expandable operations:

```

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

```

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

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

Expandably computing primes is done in [subsection 7.12](#).

5.3 A new kind of for loop

As part of the *utilities* coming with the *xinttools* package, there is a new kind of for loop, \xintFor . Check it out ([subsection 7.19](#)).

5.4 A new kind of expandable loop

Also included in *xinttools*, \xintiloop is an expandable loop giving access to an iteration index, without using count registers which would break expandability. Check it out ([subsection 7.15](#)).

³⁷ All these utilities, as well as \xintAssign , \xintAssignArray and the \xintFor loops are now available from the *xinttools* package, independently of the big integers facilities of *xint*.

6 New features of the `xintexpr` package

Release 1.1 has brought many changes to `xintexpr`. This chapter is for people already familiar with earlier versions. A more systematic item per item syntax description is provided in [section 11](#).

6.1 1.1 brought some breaking changes and quite a few novelties

First, there are some breaking changes:

- in `\xintiexpr`, `/` does *rounded* division, rather than as in earlier releases the Euclidean division (for positive arguments, this is truncated division). The new `//` operator does truncated division,
- the `:` operator for three-way branching is gone, replaced with `??`,
- `1e(3+5)` is now illegal. The number parser identifies `e` and `E` in the same way it does for the decimal mark, earlier versions treated `e` as `E` rather as postfix operators,
- the `add` and `mul` have a new syntax, old syntax is with ``+`` and ``*`` (quotes mandatory), `sum` and `prd` are gone,
- no more special treatment for encountered brace pairs `{..}` by the number scanner, `a/b[N]` notation can be used without use of braces (the `N` will end up space-stripped in a `\numexpr`, it is not parsed by the `\xintexpr`-ession scanner).
- although `&` and `|` are still available as Boolean operators the use of `&&` and `||` is strongly recommended. The single letter operators might be assigned some other meaning in later releases (bitwise operations, perhaps). Do not use them.

The novelties are numerous.

- `\xintiexpr`, `\xinttheiexpr` admit an optional argument within brackets `[d]`, it then presents the computation result (or results, if comma separated) after rounding to `d` digits after decimal mark, (the whole computation is done exactly, as in `xintexpr`),

```
\xinttheiexpr [32] 1.23^50, 1.231^50\relax
```

```
31279.19531849524327730376474278479977, 32576.36681452374761163272615218241811
```

- `\xintfloatexpr`, `\xintthefloatexpr` similarly admit an optional argument which serves to keep only `d` digits of precision, getting rid of cumulated uncertainties in the last digits (the whole computation is done according to the precision set via `\xintDigits`),

```
\xintDigits:=32;\xintthefloatexpr 1.010101^10-1.0101^10\relax
```

```
\xintDigits:=16;\xintthefloatexpr 1.010101^10-1.0101^10\relax
```

```
\xintthefloatexpr [12] 1.010101^10-1.0101^10\relax
```

```
0.000010946651065064088084734476200000
```

```
0.00001094665106500000
```

```
0.0000109466510650
```

- `\xinttheexpr` and `\xintthefloatexpr` ‘pretty-print’ if possible, the former removing unit denominator or `[0]` brackets, the latter avoiding scientific notation if decimal notation is practical,
- the `//` does truncated division and `/:` is the associated modulo,
- multi-character operators `&&`, `||`, `==`, `<=`, `>=`, `!=`, `**`,
- multi-letter infix binary words `'and'`, `'or'`, `'xor'`, `'mod'` (quotes mandatory),

- functions `even`, `odd`, `first`, `last`,
- `\xintdefvar A3:=3.1415`; for variable definitions (non expandable, naturally), usable in subsequent expressions; variable names may contain letters, digits, underscores. They should not start with a digit, the `@` is reserved, and single lowercase and uppercase Latin letters are predefined to work as dummy variables (see next),
- generation of comma separated lists `a..b`, `a..[d]..b`,
- Python syntax-like list extractors `[list][n:]`, `[list][:n]`, `[list][a:b]` and `[list][n]` (`n=0` for the number of list items), the step is always `+1`,
- function `reversed`, to reverse the order of list items,
- itemwise sequence operations `a*[list]`, etc., on both sides `a*[list]^b`,
- dummy variables in `add` and `mul`: `add(x(x+1)(x-1), x=-10..10)`,
- variable substitutions with `subs`: `subs(subs(add(x^2+y^2,x=1..y),y=t),t=20)`,
- sequence generation using `seq` with a dummy variable: `seq(x^3, x=-10..10)`,
- simple recursive sequences with `rseq`, with `@` given the last value, `rseq(1;2@+1,i=1..10)`,
- higher recursion with `rrseq`, `@1`, `@2`, `@3`, `@4`, and `@@(n)` for earlier values, up to `n=K` where `K` is the number of terms of the initial stretch `rrseq(0,1;@1+@2,i=2..100)`,
- iteration with `iter` which is like `rrseq` but outputs only the last `K` terms, where `K` was the number of initial terms,
- inside `seq`, `rseq`, `rrseq`, `iter`, possibility to use `omit`, `abort` and `break` to control termination,
- `n++` potentially infinite index generation for `seq`, `rseq`, `rrseq`, and `iter`, it is advised to use `abort` or `break(..)` at some point,
- the `add`, `mul`, `seq`, ... are nestable,
- `\xintthecoords` converts a comma separated list of an even number of items to the format as expected by the `TikZ coordinates` syntax,
- completely rewritten `\xintNewExpr`, new `protect` function to handle external macros. However not all constructs are compatible with `\xintNewExpr`.

6.2 Examples with the v1.1 `\xintexpr`

- One can define variables (the definition itself is a non expandable step). The allowed names are composed of letters, digits, and underscores. The variable should not start with a digit and single letters `a..z`, `A..Z` are predefined for use as dummy variables --- see below. The `@` is reserved.

```
\begingroup
\xintdefvar a_1 := 3.14159;\xintdefvar a2 := 2.71828;\xinttheiexpr [5] a_1+a2\relax
\endgroup
```

5.85987

- `add` and `mul` have a new syntax requiring a dummy variable:

```
\xinttheexpr add(x, x=1,3,19), mul(x^2, x=1,3,19), add(x(x+1), x= 1,3,19)\relax
```

23, 3249, 394

Use ``+`` and ``*`` (left ticks mandatory) for syntax without dummy variables:

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

23, 3249, 394

- The `seq` function generates sequences according to a given formula:

```
\xinttheexpr seq(x(x+1)(x+2), x=1,3,19), `+`(seq(x(x+1)(x+2), x=1,3,19)),
add(x(x+1)(x+2), x=1,3,19)\relax
```

6, 60, 7980, 8046, 8046

And this is nestable!

```
\xinttheexpr seq(seq(x^y, y=1..5),x=1..5), add(mul(x^y,y=1..5), x=1..5),
add(x^15, x=1..5)\relax % 15 = 1+2+3+4+5
```

And this is nestable! 1, 1, 1, 1, 1, 2, 4, 8, 16, 32, 3, 9, 27, 81, 243, 4, 16, 64, 256, 1024, 5, 25, 125, 625, 3125, 31605701625, 31605701625

One should use parentheses appropriately. The *\xintexpr* parser in normal operation is not bad at identifying missing or extra opening or closing parentheses, but when it handles *seq*, *add*, *mul* or similar constructs it switches to another mode of operation (it starts using delimited macros, something which is almost non-existent in all its other operations) and ill-formed expressions are much more likely to let the parser fetch tokens from beyond the mandatory ending *\relax*. Thus, in case of a missing parenthesis in such circumstances the error message from *T_EX* might be very cryptic, even for the seasoned *xint* user.

- As seen in the last example *a..b* constructs the integers from *a* to *b*. This is (small) integer only. A more general *a..[d]..b* works with big integers, or fractions, from *a* to *b* with step *d*.

```
\xinttheexpr seq(2x+1, x=1..10, 100..110, 3/5..[1/5]..7/5)\relax
```

3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 201, 203, 205, 207, 209, 211, 213, 215, 217, 219, 221, 11/5, 13/5, 15/5, 17/5, 19/5

- itemwise operations on lists are possible, as well as item extractions:

```
\xinttheiexpr 2*[1,10,100]^3, 5+[2*[1,10,100]^3]*100 \relax
```

2, 2000, 2000000, 205, 200005, 20000005

```
\xinttheiexpr 1+[seq(3^j, j=1..10, 21..30)][17], 1+3^27\relax
```

7625597484988, 7625597484988

We used the *[list][n]* construct which gives the *n*th item from the list. In this context there are also the functions *last* and *first*. There is no real concept of a list object, nor list operations, although itemwise manipulation are made possible as shown above via the *[...]* constructor. The list manipulation utilities are so far a bit limited. There is no notion of an ‘nuple’ object. The variable *nil* is reserved, it represents an empty list.

- subs* is similar to *seq* in syntax but is for variable substitution:

```
\xinttheexpr subs(100*subs(10*subs(3*x+5,x=y+50)+2,y=z^2),z=10)\relax % 100(10(3*150+5)+2)
```

455200

```
\xinttheexpr subs(subs(add(x^2+y^2,x=1..y),y=t),t=20)\relax
```

10870

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

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

15129, 289, 1024

- last but not least, *seq* has variants *rseq* and *rrseq* which allow recursive definitions. They start with at least one initial value, then a semi-colon, then the formula, then the list of indices to iterate over. *@* (or *@1*) evaluates to the last computed item, and *rrseq* keeps the memory of the *K* last results, where *K* was the number of initial terms. One accesses them via *@1*, *@2*, *@3*, *@4* and *@@(N)* for *N*>4. It is even possible to nest them and use *@@@* to access the values of the master recursion...

```
\xinttheiexpr rseq(1; 2*@, i=1..10), `+`(rseq(1; 2*@, i=1..10))\relax
```

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2047

```
\xinttheiexpr rseq(2; @(@+1)/2, i=1..5)\relax
```

2, 3, 6, 21, 231, 26796

```
\xinttheiexpr rseq(0,1; (@1,add(x,x=@1)), y=2..10)\relax
```

0, 1, 0, 1, 1, 0, 1, 1, 2, 0, 1, 1, 2, 4, 0, 1, 1, 2, 4, 8, 0, 1, 1, 2, 4, 8, 16, 0, 1, 1, 2, 4, 8, 16, 32, 0, 1, 1, 2, 4, 8, 16, 32, 64, 0, 1, 1, 2, 4, 8, 16, 32, 64, 128, 0, 1, 1, 2, 4, 8, 16, 32, 64, 128, 256

Some Fibonacci fun

```
\xinttheiexpr rrseq(0,1; @1+@2, x=2..10), last(rrseq(0,1; @1+@2, x=2..100))\relax
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 354224848179261915075

```
Sum of previous last three: \xinttheiexpr rrseq(0,0,1; @1+@2+@3, i=1..20)\relax
```

Sum of previous last three: 0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, 504, 927, 1705, 3136, 5768, 10609, 19513, 35890, 66012, 121415

```
Big numbers: \printnumber{\xinttheiexpr rseq(1; @(@+1), j=1..10)\relax }
```

Big numbers: 1, 2, 6, 42, 1806, 3263442, 10650056950806, 113423713055421844361000442, 128264938683278671740537145998360961546653259485195806, 1655066473245199641984681954444391802017513152706377497841851388766535868639572406808911988131737645185442, 2739245030860303142341023429167468628119436436758091462794736794160869202622699363433211840458243863492952487372839923697584879743063177305807538834294603449564100770347613304760167394546498283825541500213920806

Nested recursion often quickly leads to gigantic outputs. This is an experimental feature, susceptible to be removed or altered in the future.

```
\xinttheiexpr rrseq(1; `+`(rrseq(0,1; @@@(1)+@1+@2, i=1..10)), j=1..5)\relax
```

1, 596, 217176, 79052296, 28775035976, 10474113095496

- The special keywords `omit`, `abort` and `break(..)` are available inside `seq`, `rseq`, `rrseq`, as well as the `n++` for a potentially infinite iteration. The `n++` construct in conjunction with an `abort` or `break` is often more efficient, because in other cases the list to iterate over is first completely constructed.

First Fibonacci number at least $|2^{31}|$ and its index

```
\xinttheiexpr iter(0,1; (@1>=2^31)?{break(i)}{+@2+@1}, i=1++)\relax
```

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

Prime numbers are always cool

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

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

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

- The `iter` function is like `rrseq` but does not leave a trace of earlier items, it starts with `K` initial values, then it iterates: either a fixed number of times, or until aborting or breaking. And ultimately it prints `K` final values.

```
The first Fibonacci number beyond the \TeX{} bound is
```

```
\xinttheiexpr subs(iter(0,1;(@1>N)?{break(i)}{@1+@2},i=1++),N=2^31)\relax{}
```

and the previous number was its index.

The first Fibonacci number beyond the $\text{T}_{\text{E}}\text{X}$ bound is 2971215073, 47 and the previous number was its index. But this was a bit too easy, what is the smallest Fibonacci number not representable on 64 bits?

```
The first Fibonacci number beyond |2^64| bound is
\xinttheiexpr subs(iter(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 19740274219868223167, 94 and the previous number was its index.

One more recursion:

```
\def\syr #1{\xinttheiexpr rseq(#1; (@<=1)?{break(i)}{odd(@)?{3@+1}{@//2}},i=0++)\relax}
The 3x+1 problem: \syr{231}\par
```

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

Ok, a final one:

```
\def\syrMax #1{\xinttheiexpr iter(#1,#1;even(i)?
    {(@2<=1)?{break(i/2)}{odd(@2)?{3@2+1}{@2//2}}}
    {(@1>@2)?{@1}{@2}},i=0++)\relax }
With initial value 1161, the maximal number attained is \syrMax{1161} and that latter
number is the number of steps which was needed to reach 1.\par
```

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

Well, one more:

```
\newcommand\GCD [2]{\xinttheiexpr rrseq(#1,#2; (@1=0)?{abort}{@2/:@1}, i=1++)\relax }
\GCD {13^10*17^5*29^5}{2^5*3^6*17^2}
```

4014838863509162883616357, 6741792, 3367717, 6358, 4335, 2023, 289, 0

and the ultimate:

```
\newcommand\Factors [1]{\xinttheiexpr
  subs(seq((i/:3=2)?{omit}{[L][i]},i=1..([L][0])),
  L=rseq(#1;(p^2>[@][1])?{([@][1]>1)?{break(1,[@][1],1)}{abort}}
    {([@][1])/p}?{omit}
    {iter(([@][1])/p; (@/:p)?{break(@,p,e)}{@//p},e=1++))\relax }
\Factors {41^4*59^2*29^3*13^5*17^8*29^2*59^4*37^6}
```

16246355912554185673266068721806243461403654781833, 13, 5, 17, 8, 29, 5, 37, 6, 41, 4, 59, 6

This might look a bit scary, I admit. *xintexpr* has minimal tools and is obstinate about doing everything expandably! We are hampered by absence of a notion of ‘nuple’. The algorithm divides N by 2 until no more possible, then by 3, then by 4 (which is silly), then by 5, then by 6 (silly again),

The variable `L=rseq(#1;...)` expands, if one follows the steps, to a comma separated list starting with the initial (evaluated) $N=\#1$ and then pseudo-triplets where the first item is N trimmed of small primes, the second item is the last prime divisor found, and the third item is its exponent in original N .

The algorithm needs to keep handy the last computed quotient by prime powers, hence all of them, but at the very end it will be cleaner to get rid of them (this corresponds to the first line in the code above). This is achieved in a cumbersome inefficient way; indeed each

item extraction `[L][i]` is costly: it is not like accessing an array stored in memory, due to expandability, nothing can be stored in memory! Nevertheless, this step could be done here in a far less inefficient manner if there was a variant of `seq` which, in the spirit of `\xint-iloopindex`, would know how many steps it had been through so far. This is a feature to be added to `\xintexpr`! (as well as a `++` construct allowing a non unit step).

Notice that in `iter(([@][1])/p;` the `@` refers to the previous triplet (or in the first step to `N`), but the latter `@` showing up in `(@/:p)?` refers to the previous value computed by `iter`.

Parentheses are essential in `..([y][0])` else the parser will see `..[` and end up in ultimate confusion, and also in `([@][1])/p` else the parser will see the itemwise operator `] /` on lists and again be very confused (I could implement a `] /:` on lists, but in this situation this would also be very confusing to the parser.)

For comparison, here is an *f*-expandable macro expanding to the same result, but coded directly with the *xint* macros. Here `#1` can not be itself an expression, naturally. But at least we let `\Factorize` *f*-expand its argument.

```
\makeatletter
\newcommand\Factorize [1]
  {\romannumeral0\expandafter\factorize\expandafter{\romannumeral-\`0#1}}%
\newcommand\factorize [1]{\xintiifOne{#1}{ 1}{\factors@a #1.#1;}}%
\def\factors@a #1.{\xintiifOdd{#1}
  {\factors@c 3.#1.%
    {\expandafter\factors@b \expandafter1\expandafter.\romannumeral0\xinthalff{#1}.}}%
\def\factors@b #1.#2.{\xintiifOne{#2}
  {\factors@end {2, #1}%
    {\xintiifOdd{#2}{\factors@c 3.#2.{2, #1}}%
      {\expandafter\factors@b \the\numexpr #1+\@ne\expandafter.%
        \romannumeral0\xinthalff{#2}.}}}%
}%
\def\factors@c #1.#2.%
  \expandafter\factors@d\romannumeral0\xintiidivision {#2}{#1}{#1}{#2}%
}%
\def\factors@d #1#2#3#4{\xintiifNotZero{#2}
  {\xintiifGt{#3}{#1}
    {\factors@end {#4, 1}}% ultimate quotient is a prime with power 1
    {\expandafter\factors@c\the\numexpr #3+\tw@.#4.}}%
  {\factors@e 1.#3.#1.%
}%
\def\factors@e #1.#2.#3.{\xintiifOne{#3}
  {\factors@end {#2, #1}}%
  {\expandafter\factors@f\romannumeral0\xintiidivision {#3}{#2}{#1}{#2}{#3}}%
}%
\def\factors@f #1#2#3#4#5{\xintiifNotZero{#2}
  {\expandafter\factors@c\the\numexpr #4+\tw@.#5.{#4, #3}}%
  {\expandafter\factors@e\the\numexpr #3+\@ne.#4.#1.%
}%
\def\factors@end #1;{\xintlistwithsep{, }{\xintRevWithBraces {#1}}}%
\makeatother
```

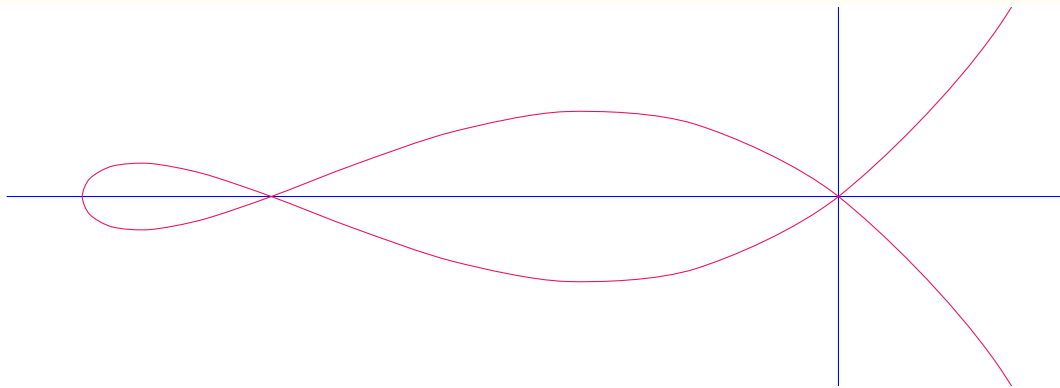
The macro `\Factorize` puts a little stress on the input save stack in order not be bothered with previously gathered things. I timed it to be about eight times faster than `\Factors` in test cases such as `16246355912554185673266068721806243461403654781833` and others. Among the various things explaining the speed-up, there is fact that we step by increments of two, not one, and also that we divide only once, obtaining quotient and remainder in one go. These two things already make for a speed-up factor of about four. Thus, our earlier `\Factors` was not completely inefficient, and was quite easier to come up with than `\Factorize`.

7 Commands of the **xinttools** package

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...

To conclude with this overview of the new features in **xintexpr** 1.1, I will mention `\xintthecoords` which converts a comma separated list as produced by `\xintfloatexpr` or `\xintiexpr [d]` to the format expected by the **TikZ** `coordinates` syntax.

```
{\centering\begin{tikzpicture}[scale=10]\xintDigits:=8;
\clip (-1.1,-.25) rectangle (.3,.25);
\draw [blue] (-1.1,0)--(1,0);
\draw [blue] (0,-1)--(0,+1);
\draw plot[smooth] coordinates {\xintthecoords
% converts into (x1, y1) (x2, y2)... format
\xintfloatexpr seq((x^2-1,mul(x-t,t=-1+[0..4]/2)),x=-1.2..[0.1]..+1.2) \relax };
\end{tikzpicture}\par }
```



`\xintthecoords` should be followed immediately by `\xintfloatexpr` or `\xintiexpr` or `\xintiiexpr`, but not `\xintthefloatexpr`, etc. . .

Besides, as **TikZ** will not understand the `A/B[N]` format which is used on output by `\xintexpr`, `\xiintexpr` is not really usable with `\xintthecoords` for a **TikZ** picture, but one may use it on its own, and the reason for the spaces in and between coordinate pairs is to allow if necessary to print on the page for examination with about correct line-breaks.

```
\oodef\x{\xintthecoords \xintexpr rrseq(1/2,1/3; @1+@2, x=1..20)\relax }
\meaning\x +++
```

```
macro:->(1/2, 1/3) (5/6, 7/6) (12/6, 19/6) (31/6, 50/6) (81/6, 131/6) (212/6, 343/6) (555/6,
898/6) (1453/6, 2351/6) (3804/6, 6155/6) (9959/6, 16114/6) (26073/6, 42187/6)+++
```

7 Commands of the **xinttools** package

.1	<code>\xintReverseOrder</code>	33	.11	<code>\xintApplyUnbraced</code>	37
.2	<code>\xintRevWithBraces</code>	33	.12	<code>\xintSeq</code>	37
.3	<code>\xintLength</code>	33	.13	Completely expandable prime test	38
.4	<code>\xintZapFirstSpaces, \xintZapLastSpaces,</code> <code>\xintZapSpaces, \xintZapSpacesB</code>	34	.14	<code>\xintloop, \xintbreakloop,</code> <code>\xintbreakloopanddo, \xintloopskiptonext</code>	40
.5	<code>\xintCSVtoList</code>	34	.15	<code>\xintiloop, \xintiloopindex,</code> <code>\xintouteriloopindex,</code> <code>\xintbreakiloop, \xintbreakiloopanddo,</code> <code>\xintiloopskiptonext,</code> <code>\xintiloopskipandredo</code>	42
.6	<code>\xintNthElt</code>	35	.16	Another completely expandable prime test ..	45
.7	<code>\xintKeep</code>	36			
.8	<code>\xintTrim</code>	36			
.9	<code>\xintListWithSep</code>	36			
.10	<code>\xintApply</code>	37			

.17	A table of factorizations	46	.24	Some arithmetic with Fibonacci numbers ...	55
.18	<code>\xintApplyInline</code>	47	.25	<code>\xintForpair</code> , <code>\xintForthree</code> , <code>\xintForfour</code>	58
.19	<code>\xintFor</code> , <code>\xintFor*</code>	49	.26	<code>\xintAssign</code>	59
.20	<code>\xintifForFirst</code> , <code>\xintifForLast</code>	51	.27	<code>\xintAssignArray</code>	59
.21	<code>\xintBreakFor</code> , <code>\xintBreakForAndDo</code>	52	.28	<code>\xintRelaxArray</code>	60
.22	<code>\xintintegers</code> , <code>\xintdimensions</code> , <code>\xintrationals</code>	52	.29	<code>\odef</code> , <code>\oodef</code> , <code>\fdef</code>	60
.23	Another table of primes	54	.30	The Quick Sort algorithm illustrated	60

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.

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

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

7.1 `\xintReverseOrder`

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

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

7.2 `\xintRevWithBraces`

- f* ★ `\xintRevWithBraces{⟨list⟩}` first does the *f*-expansion of its argument then it reverses the order of the tokens, or braced material, it encounters, 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.

7.3 `\xintLength`

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

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

7.4 `\xintZapFirstSpaces`, `\xintZapLastSpaces`, `\xintZapSpaces`, `\xintZapSpacesB`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.5 `\xintCSVtoList`

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

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

n ★ 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 T_EX into single spaces. All such spaces around commas³⁸ are removed, as well as the spaces at the start and the spaces at the end of the list.³⁹ The items may contain explicit `\par`'s or empty lines (converted by the T_EX 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 T_EX collapses on input consecutive blanks into one space token), for ‘ , ’ the output consists of two consecutive empty items `{ } { }`. Recall that on output everything is braced, a `{ }` is an ‘empty’ item. Most of the above is mainly irrelevant for every day use, apart perhaps from the fact to be noted that an empty input does not give an empty output but a one-empty-item list (it is as if an ending comma was always added at the end of the input).

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

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

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

Else, we may have direct use:

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

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

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

7.6 `\xintNthElt`

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

³⁸ and multiple space tokens are not a problem; but those at the top level (not hidden inside braces) *must* be of character code 32. ³⁹ let us recall that this is all done completely expandably... There is absolutely no alteration of any sort of the item apart from the stripping of initial and final space tokens (of character code 32) and brace removal if and only if the item apart from initial and final spaces (or more generally multiple `char 32` space tokens) is braced.

```

\uintNthElt {3}{\{agh\}\u{zzz}\v{Z}} is zzz
\uintNthElt {3}{\{agh\}\u{\{zzz\}\v{Z}} is {zzz}
\uintNthElt {2}{\{agh\}\u{\{zzz\}\v{Z}} is \u
\uintNthElt {37}{\xintFac {100}}=9 is the thirty-seventh digit of 100!.
\uintNthElt {10}{\xintFtoCv {566827/208524}}=1457/536
is the tenth convergent of 566827/208524 (uses xintcfrac package).
\uintNthElt {7}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=7
\uintNthElt {0}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=9
\uintNthElt {-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.

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

num
x n ★

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

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

7.7 `\xintKeep`

num
x f ★
New with
1.09m

`\xintKeep{x}{\langle list \rangle}` expands the list argument and returns a new list containing only the first x elements. If $x<0$ the macro returns the last $|x|$ elements (in the same order as in the initial list). If $|x|$ equals or exceeds the length of the list, the list (as arising from expansion of the second argument) is returned. For $x=0$ the empty list is returned.

Naked (non space) tokens from the original count each as one item and they end up braced in the output (if present there): if one later wants to remove all brace pairs (either added to a naked token, or initially present), one may use `\xintListWithSep` with an empty separator.

`\xintKeepNoExpand` does the same without first f -expanding its list argument.

```

\oodef\test {\xintKeep {17}{\xintKeep {-69}{\xintSeq {1}{100}}}\meaning\test
macro:->{32}{33}{34}{35}{36}{37}{38}{39}{40}{41}{42}{43}{44}{45}{46}{47}{48}

```

7.8 `\xintTrim`

num
x f ★
New with
1.09m

`\xintTrim{x}{\langle list \rangle}` expands the list argument and gobbles its first x elements. If $x<0$ the macro gobbles the last $|x|$ elements. If $|x|$ equals or exceeds the length of the list, the empty list is returned. For $x=0$ the full list is returned.

Naked (non space) tokens from the original count each as one item and they end up braced in the output (if present there).

`\xintTrimNoExpand` does the same without first f -expanding its list argument.

```

\oodef\test {\xintTrim {17}{\xintTrim {-69}{\xintSeq {1}{100}}}\meaning\test
macro:->{18}{19}{20}{21}{22}{23}{24}{25}{26}{27}{28}{29}{30}{31}

```

7.9 `\xintListWithSep`

nf ★

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

the original).

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

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

7.10 `\xintApply`

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

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

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

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

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

7.11 `\xintApplyUnbraced`

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

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

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

```
\def\macro #1{\expandafter\def\csname myself#1\endcsname {#1}}
\xintApplyUnbraced\macro{{elta}{eltb}{eltc}}
\begin{enumerate}[nosep,label=(\arabic{*})]
\item \meaning\myselfelta
\item \meaning\myselfeltb
\item \meaning\myselfeltc
\end{enumerate}
```

(1) `macro-->elta`

(2) `macro-->eltb`

(3) `macro-->eltc`

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

7.12 `\xintSeq`

$\begin{smallmatrix} \text{num} \\ \text{[} \text{X} \text{]} \end{smallmatrix}$ $\begin{smallmatrix} \text{num} & \text{num} & \text{num} \\ \text{X} & \text{X} & \text{X} \end{smallmatrix}$ ★ `\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 $y - x$ (beware that `\xintSeq {1}{0}` is thus not empty but `{1}{0}`, use `\xintSeq [1]{1}{N}` if you want an empty sequence for N zero or negative).

The current implementation is only for (short) integers; possibly, a future variant could allow big integers and fractions, although one already has access to similar functionality using `\xintApply` to get any arithmetic sequence of long integers. Currently thus, `x` and `y` are expanded inside a `\numexpr` so they may be count registers or a \TeX `\value{counternam}`, 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
\xintiSum{\xintSeq [3]{1}{1000}}
167167
```



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

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

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

7.13 Completely expandable prime test

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

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

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

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

We used the `xint` provided expandable tests (on big integers or fractions) in 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`⁴¹. The macro becomes:

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

In the odd case however we have to assume the integer is at least 7, as `\xintSeq` generates an empty list if `#1=3` or `5`, and `\xintANDof` returns 1 when supplied an empty list. Let us ease up a bit

⁴⁰ a `\space` will stop the \TeX scanning of a number and be gobbled in the process, maintaining expandability if this is required; the `\relax` stops the scanning but is not gobbled and remains afterwards as a token. ⁴¹ <http://ctan.org/pkg/etoolbox>

`\xintANDof`'s work by letting it work on only 0's and 1's. We could use:

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

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

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

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

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

The input is still assumed positive. There is a deliberate blank before `\IsNotDivisibleBy` to use this feature of `\xintApply`: a space stops the expansion of the applied macro (and disappears). This expansion will be done by `\xintANDof`, which has been designed 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 7.16) which is still expandable and another one (subsection 7.23) which is a close variant of the `\IsPrime` code above but with the `\xintFor` loop, thus breaking expandability. The `\xintiloop` variant does not first evaluate the integer square root, the `\xintFor` variant still does. I did not compare their efficiencies.

Let us construct with this expandable primality test a table of the prime numbers up to 1000. We need to count how many we have in order to know how many tab stops one should add in the last row.⁴² 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}
```

⁴² although a tabular row may have less tabs than in the preamble, there is a problem with the | vertical rule, if one does that.

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.

7.14 `\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 `\xinttiloop` is `\xintloop` with an embedded expandable mechanism allowing to exit from the loop. The iterated commands 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 `\xinttiloop` will primarily be of interest to experienced \TeX macro programmers, my description will assume that the user is knowledgeable enough. Some examples in this document will be perhaps more illustrative than my attempts at explanation of use.

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

One will usually employ some count registers to manage the exit test from the loop; this breaks expandability, see `\xinttiloop` for an expandable integer indexed loop. Use in alignments will be complicated by the fact that cells create groups, and also from the fact that any encountered un-expandable material will cause the \TeX input scanner to insert `\endtemplate` on each encountered `&` or `\cr`; thus `\xintbreakloop` may not work as expected, but the situation can be resolved via `\xint_t_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`.⁴³

```

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

```

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

```

\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}

```

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

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

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

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

7.15 `\xintilooop`, `\xintilooopindex`, `\xintouterilooopindex`, `\xintbreakilooop`, `\xintbreakilooopanddo`, `\xintilooopskiptonext`, `\xintilooopskipandredo`

☆ `\xintilooop[start+delta](stuff)\if<test> ... \repeat` is a completely expandable nestable loop.

complete expandability depends naturally on the actual iterated contents, and complete expansion will not be achievable under a sole *f*-expansion, as is indicated by the hollow star in the margin; thus the loop can be used inside an `\edef` but not inside arguments to the package macros. It can be used inside an `\xintexpr..\relax`.

This loop benefits via `\xintloopindex` 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, `\xintouterloopindex` gives access to the index of the outer loop. If needed one could write on its model a macro giving access to the index of the outer outer loop (or even to the *nth* outer loop).

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

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

One has `\xintbreakilop` and `\xintbreakilopanddo` to abort the loop. The syntax of `\xintbreakilopanddo` 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:

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

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

```
\expandafter\xintbreakilopanddo\expandafter\macro\xintloopindex.%
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\xintbreakilopanddo\expandafter\macro\xintloopindex.}%
\fi etc..etc.. \repeat
```

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

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

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

```
\begingroup
\edef\z
{\xintloop [10001+2]
 {\xintloop [3+2]
  \ifnum\xintouterloopindex<\numexpr\xintloopindex*\xintloopindex\relax
```

```

\ointouteriloopindex,
\expandafter\ointbreakiloop
\fi
\ifnum\ointouteriloopindex=\numexpr
  (\ointouteriloopindex/\ointiloopindex)*\ointiloopindex\relax
\else
\repeat
}% no space here
\ifnum \ointiloopindex < 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 7.16](#) 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
\ointiloop [1+1]
{\expandafter\bfseries\ointiloopindex &
\ointiloop [1+1]
\ifnum\ointouteriloopindex=\numexpr
  (\ointouteriloopindex/\ointiloopindex)*\ointiloopindex\relax
\ointiloopindex&\fi
\ifnum\ointiloopindex<\ointouteriloopindex\space % CRUCIAL \space HERE
\repeat \cr }%
\ifnum\ointiloopindex<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
4	1	2	4	19	1	19	
5	1	5	20	1	2	4	5
6	1	2	3	6	10	20	
7	1	7	21	1	3	7	21
8	1	2	4	8	22	1	2
9	1	3	9	23	1	23	
10	1	2	5	10	24	1	2
11	1	11	25	1	5	25	
12	1	2	3	4	6	8	12
13	1	13	26	1	2	13	26
14	1	2	7	14	27	1	3
15	1	3	5	15	28	1	2
					29	1	29
					30	1	2
						3	5
						6	10
						15	30

We wanted this first entry in bold face, but `\bfseries` leads to unexpandable tokens, so the `\exp`

`andafter` was necessary for `\xintloopindex` and `\xintouterloopindex` not to be confronted with a hard to digest `\endtemplate`. An alternative way of coding:

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

7.16 Another completely expandable prime test

The `\IsPrime` macro from [subsection 7.13](#) checked expandably if a (short) integer was prime, here is a partial rewrite using `\xintloop`. We use the `etoolbox` expandable conditionals for convenience, but not everywhere as `\xintloopindex` can not be evaluated while being braced. This is also the reason why `\xintbreakloopanddo` is delimited, and the next macro `\SmallestFactor` which returns the smallest prime factor exemplifies that. One could write more efficient completely expandable routines, the aim here was only to illustrate use of the general purpose `\xintloop`. A little table giving the first values of `\SmallestFactor` follows, its coding uses `\xintFor`, which is described later; none of this uses count registers.

```
\let\IsPrime\undefined \let\SmallestFactor\undefined % clean up possible previous mess
\newcommand{\IsPrime}[1] % returns 1 if #1 is prime, and 0 if not
{\ifnumodd {#1}
{\ifnumless {#1}{8}
{\ifnumequal{#1}{1}{0}{1}}% 3,5,7 are primes
{\if
\xintloop [3+2]
\ifnum#1<\numexpr\xintloopindex*\xintloopindex\relax
\expandafter\xintbreakloopanddo\expandafter1\expandafter.%
\fi
\ifnum#1=\numexpr (#1/\xintloopindex)*\xintloopindex\relax
\else
\repeat 00\expandafter0\else\expandafter1\fi
}%
}% END OF THE ODD BRANCH
{\ifnumequal {#1}{2}{1}{0}}% EVEN BRANCH
}%
\catcode`_ 11
\newcommand{\SmallestFactor}[1] % returns the smallest prime factor of #1>1
{\ifnumodd {#1}
{\ifnumless {#1}{8}
{#1}% 3,5,7 are primes
{\xintloop [3+2]
\ifnum#1<\numexpr\xintloopindex*\xintloopindex\relax
\xint_afterfi{\xintbreakloopanddo#1.}%
\fi
\ifnum#1=\numexpr (#1/\xintloopindex)*\xintloopindex\relax
\xint_afterfi{\expandafter\xintbreakloopanddo\xintloopindex.}%
\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

7.17 A table of factorizations

As one more example with `\xintilooop` 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 `\xintilooop` as it didn't appear to be the convenient tool. As `\factorize` will have to be used on `\xintilooopindex`, 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 `\xintilooopskiptonext`.

The code next generates a `table` which has been made into a float appearing on page 48. Here is now the code for factorization; the conditionals use the package provided `\xint_firstoftwo` and `\xint_secondoftwo`, one could have employed rather \TeX 's own `\@firstoftwo` and `\@secondoftwo`, or, simpler still in \TeX context, the `\ifnumequal`, `\ifnumless` . . . , utilities from the package `etoolbox` which do exactly that under the hood. Only \TeX acceptable numbers are treated here, but it would be easy to make a translation and use the `xint` macros, thus extending the scope to big numbers; naturally up to a cost in speed.

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

```

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

\def\factorize #1.{\ifnum#1=1 \abortfactorize\fi
\ifnum\numexpr #1-2=\numexpr ((#1/2)-1)*2\relax
\expandafter\xint_firstoftwo
\else\expandafter\xint_secondoftwo
\fi
{2&\expandafter\factorize\the\numexpr#1/2.}%

```

```

{\factorize_b #1.3.}}%

\def\factorize_b #1.#2.{\ifnum#1=1 \abortfactorize\fi
  \ifnum\numexpr #1-(#2-1)*#2<#2
    #1\abortfactorize
  \fi
  \ifnum \numexpr #1-#2=\numexpr ((#1/#2)-1)*#2\relax
    \expandafter\xint_firstoftwo
  \else\expandafter\xint_secondoftwo
  \fi
  {#2&\expandafter\factorize_b\the\numexpr#1/#2.#2.}%
  {\expandafter\factorize_b\the\numexpr #1\expandafter.%
    \the\numexpr #2+2.}}%

\catcode\_ 8
\begin{figure*}[ht!]
\centering\phantomsection\label{floatfactorize}\normalcolor
\tabskip1ex
\centeredline{\vbox{\halign {\hfil\strut#\hfil&\hfil#\hfil\cr\noalign{\hrule}
  \xintilooop ["7FFFFFFE0+1]
  \expandafter\bfseries\xintilooopindex &
  \ifnum\xintilooopindex="7FFFFFFED
    \number"7FFFFFFED\cr\noalign{\hrule}
  \expandafter\xintilooopskiptonext
  \fi
  \expandafter\factorize\xintilooopindex.\cr\noalign{\hrule}
  \ifnum\xintilooopindex<"7FFFFFFE
  \repeat
  \bfseries \number"7FFFFFFF&\number "7FFFFFFF\cr\noalign{\hrule}
}}}
\centeredline{A table of factorizations}
\end{figure*}

```

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

7.18 `\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

7 Commands of the *xinttools* package

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

provides an example:

```
\centerline{\normalcolor\begin{tabular}{ccc}
  $N$ & $N^2$ & $N^3$ \\ \hline
  \def\Row #1{ #1 & \xintiSqr {#1} & \xintiPow {#1}{3} \\ \hline }%
  \xintApplyInline \Row {\xintCSVtoList{17,28,39,50,61}}
\end{tabular}}\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 T_EX'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{&\xintiPow {#1}{#2}}%
\centeredline {\begin{tabular}{ccccccccc} &0&1&2&3&4&5&6&7&8&9\\ \hline
\xintApplyInline \Row {0123456789}
\end{tabular}}
\end{figure*}
```

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

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

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

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

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

But see [\xintFor](#).

7.19 `\xintFor`, `\xintFor*`

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

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

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

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

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

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

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

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

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

The starred variant `\xintFor*` deals with token lists (*spaces between braced items or single tokens are not significant*) and *f-expands* each unbraced list item. This makes it easy to simulate concatenation of various list macros `\x`, `\y`, ... If `\x` expands to `{1}{2}{3}` and `\y` expands to `{4}{5}{6}` then `{\x\y}` as argument to `\xintFor*` has the same effect as `{{1}{2}{3}}{4}{5}{6}`⁴⁴. Spaces at the start, end, or in-between items are gobbled (but naturally not the spaces which may be inside braced items). Except if the list argument is a single macro with no parameters, it must be braced. Each item which is not braced will be fully expanded (as the `\x` and `\y` in the example above). An empty list leads to an empty result.

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

`\xintFor* #1 in {\xintSeq [+2]{-7}{+2}}\do {stuff with #1}` will have `#1=-7,-5,-3,-1, and 1`. The `#1` as issued from the list produced by `\xintSeq` is the literal representation as would be produced by `\arabic` on a \TeX counter, it is not a count register.

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

When used in `\ifnum` tests or other contexts where \TeX looks for a number it should thus be postfixed with `\relax` or `\space`.

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

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

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

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

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

When inserted inside a macro for later execution the `#` characters must be doubled.⁴⁴ For example:

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

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

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

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

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

7.20 `\xintifForFirst`, `\xintifForLast`

nn ★ `\xintifForFirst` {YES branch}{NO branch} and `\xintifForLast` {YES branch}{NO branch} execute the

⁴⁴ 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. ⁴⁵ to be completely honest, both `\xintFor` and `\xintFor*` initially scoop up both the list and the iterated commands; `\xintFor` scoops up a second time the entire comma separated list in order to feed it to `\xintCSVtoList`. The starred variant `\xintFor*` which does not need this step will thus be a bit faster on equivalent inputs.

YES or NO branch if the `\xintFor` or `\xintFor*` loop is currently in its first, respectively last, iteration.

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

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

It is a known feature of these conditionals that they cease to function if put at a location of the `\xintFor` replacement text which has closed a group, for example in the last cell of an alignment created by the loop, assuming the replacement text of the `\xintFor` loop creates a row. The conditional must be used before the first cell is closed. This is not likely to change in future versions. It is not an intrinsic limitation as the branches of the conditional can be the complete rows, inclusive of all `&`'s and the tabular newline `\\`.

7.21 `\xintBreakFor`, `\xintBreakForAndDo`

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

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

7.22 `\xintintegers`, `\xintdimensions`, `\xintrationals`

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

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

⁴⁶ <http://ctan.org/pkg/ifthen> ⁴⁷ <http://ctan.org/pkg/etoolbox> ⁴⁸ the difficulties here are similar to those mentioned in [subsection 4.5](#), although less severe, as complete expandability is not to be maintained; hence the allowed use of `ifthen`. ⁴⁹ 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`.



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

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

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

```
\begingroup\small
\noindent\parbox{\dimexpr\linewidth-3em}{\color[named]{OrangeRed}%
\xintFor #1 in {\xintrationals [10/21+1/21]} \do
{
  #1=\xintifInt {#1}
  {\textcolor{blue}{\xintTrunc{10}{#1}}}
  {\xintTrunc{10}{#1}}% display in blue if an integer
  \xintifGt {#1}{1.123}{\xintBreakFor}{,}%
}}
\endgroup\smallskip
10/21=0.4761904761,    11/21=0.5238095238,    12/21=0.5714285714,    13/21=0.6190476190,
14/21=0.6666666666,    15/21=0.7142857142,    16/21=0.7619047619,    17/21=0.8095238095,
18/21=0.8571428571,    19/21=0.9047619047,    20/21=0.9523809523,    21/21=1.0000000000,
22/21=1.0476190476, 23/21=1.0952380952, 24/21=1.1428571428
```

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

⁵⁰ see [subsubsection 11.7.2](#) for the significance of the `protect`'s: they are needed because the expression has macro parameters inside macros, and not only functions from the `\xintexpr` syntax. The `\FA` turns out to have meaning `macro:#1#2->\romannumeral -'0\xintSPraw::csv {\xintRound::csv {0}{\xintDiv {\xintPow {\DimToNum {#2}}{3}}{\xintPow {\DimToNum {#1}}{2}}}`. The `\romannumeral` part is only to ensure it expands in only two steps, and could be removed. The mysterious `\xintiRound::csv` is what `\xintexpr` uses to round its result (or comma separated results) to an integer. See also the next footnote.

⁵¹ to tell the whole truth we cheated and divided by `10` the computation time through using the following definitions, together with a horizontal step of `.25pt` rather than `.1pt`. The displayed original code would make the slowest computation of all those done in this document using the `xint` bundle

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

in scientific notation with exponents in the hundreds, as they will get converted into as many zeroes.

```
\noindent\parbox{\dimexpr.7\linewidth}{\raggedright
\intFor #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 behavior should be changed? or made optional? Anyhow printing of fixed points numbers should be dealt with via dedicated packages such as `numprint` or `siunitx`.

7.23 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 7.12](#), 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.
\xifnumodd {\TheNumber}
{\ifnumgreater {\TheNumber}{1}
{\edef\ItsSquareRoot{\xintiSqrt \TheNumber}%
\xintFor ##1 in {\xintintegers [3+2]}\do
{\ifnumgreater {##1}{\ItsSquareRoot} % ##1 is a \numexpr.
{\def#1{1}\xintBreakFor}
{}}%
\xifnumequal {\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 55):

```
\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.					

7.24 Some arithmetic with Fibonacci numbers

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\xintiieval 1\expandafter\relax\expandafter}\expandafter
{\romannumeral0\xintiieval 1\expandafter\relax\expandafter}\expandafter
{\romannumeral0\xintiieval 1\expandafter\relax\expandafter}\expandafter
{\romannumeral0\xintiieval 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\xintiieval sqrt(#2)+sqrt(#3)\expandafter\relax\expandafter}\expandafter
{\romannumeral0\xintiieval (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\xintiieval sqr(#2)+sqr(#3)\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiieval (2#2-#3)#3\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiieval #2#4+#3#5\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiieval #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{\xinttheiexpr #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\xintiieval #2#4+#3#5\expandafter\relax
      \expandafter}\expandafter
    {\romannumeral0\xintiieval #2#5+#3(#4-#5)\relax}}% idem.
% \FibacciN returns F(N) (in encapsulated format: needs \xintthe for printing)
\def\FibacciN {\expandafter\xint_secondoftwo\romannumeral-`0\Fibonacci }%
\catcode`_ 8

```

I have modified the ending: we want not only one specific value $F(N)$ but a pair of successive values which can serve as starting point of another routine devoted to compute a whole sequence $F(N)$, $F(N+1)$, $F(N+2)$, This pair is, for efficiency, kept in the encapsulated internal *xintexpr* format. `\FibacciN` outputs the single $F(N)$, also as an *xintexpr*-ession, and printing it will thus 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 `\FibacciN`).

```

\def\Fibo #1.{\xintthe\FibacciN {#1}}%
\message{\xintloop [0+1] \expandafter\Fibo\xintloopindex.,
  \ifnum\xintloopindex<49 \repeat \xintthe\FibacciN{50}.}

```

The various `\romannumeral0\xintiieval` could very well all have been `\xintiiepr`'s but then we would have needed more `\expandafter`'s. Indeed the order of expansion must be controlled for the whole thing to work, and `\romannumeral0\xintiieval` is the first expanded form of `\xintiiepr`.

The way we use `\expandafter`'s to chain successive *xintexpr* evaluations is exactly analogous to well-known expandable techniques made possible by `\numexpr`.

There is a difference though: `\numexpr` is *NOT* expandable, and to force its expansion we must prefix it with `\the` or `\number`. On the other hand `\xintexpr`, `\xintiiepr`, ..., (or `\xinteval`, `\xintiieval`, ...) expand fully when prefixed by `\romannumeral-`0`: the computation is fully executed and its result encapsulated in a private format.

Using `\xintthe` as prefix is necessary to print the result (this is like `\the` for `\numexpr`), but it is not necessary to get the computation done (contrarily to the situation with `\numexpr`).

And, starting with release 1.09j, it is also allowed to expand a non `\xintthe` prefixed *xintexpr*-ession inside an `\edef`: the private format is now protected, hence the error message complaining about a missing `\xintthe` will not be executed, and the integrity of the format will be preserved.

This new possibility brings some efficiency gain, when one writes non-expandable algorithms using *xintexpr*. If `\xintthe` is employed inside `\edef` the number or fraction will be unlocked into its possibly hundreds of digits and all these tokens will possibly weigh on the upcoming shuffling of (braced) tokens. The private encapsulated format has only a few tokens, hence expansion will proceed a bit faster.

see footnote⁵²

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

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

but for the reasons explained above, it is 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 a sequence of braced Fibonacci numbers $\{F(N)\}$, $\{F(N+1)\}$, $\{F(N+2)\}$..., 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\FibonacciSeq\expandafter
    {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2-1}%
}%
\def\FibonacciSeq #1#2{%
  \expandafter\FibonacciSeq_loop\expandafter
    {\the\numexpr #1\expandafter}\romannumeral0\Fibonacci {#1}{#2}%
}%
\def\FibonacciSeq_loop #1#2#3#4{% standard Fibonacci recursion
  {#3}\unless\ifnum #1<#4 \FibonacciSeq_end\fi
  \expandafter\FibonacciSeq_loop\expandafter
    {\the\numexpr #1+1\expandafter}\expandafter
    {\romannumeral0\xintiieval #2+#3\relax}{#2}{#4}%
}%
\def\FibonacciSeq_end\fi\expandafter\FibonacciSeq_loop\expandafter
  #1\expandafter #2#3#4\fi {#3}}%
\catcode\_ 8
```

Deliberately and for optimization, this `\FibonacciSeq` macro is completely expandable but not *f*-expandable. It would be easy to modify it to be so. But I wanted to check that the `\xintFor*` does apply full expansion to what comes next each time it fetches an item from its list argument. Thus, there is no need to generate lists of braced Fibonacci numbers beforehand, as `\xintFor*`, without using any `\edef`, still manages to generate the list via iterated full expansion.

I initially used only one `\halign` in a three-column `multicols` environment, but `multicols` only knows to divide the page horizontally evenly, thus I employed in the end one `\halign` for each column (I could have then used a `tabular` as no column break was then needed).

```
\newcounter{index}
\tabskip 1ex
\fdef\Fibxxx{\FibonacciN {30}}%
\setcounter{index}{30}%
\ vbox{\halign{\bfseries#. \hfil&#\hfil &\hfil #\cr
  \xintFor* #1 in {\FibonacciSeq {30}{59}}\do
    {\theindex &\xintthe#1 &
      \xintiRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{index}\cr }}%
}\vrule
\ vbox{\halign{\bfseries#. \hfil&#\hfil &\hfil #\cr
  \xintFor* #1 in {\FibonacciSeq {60}{89}}\do
    {\theindex &\xintthe#1 &
      \xintiRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{index}\cr }}%
}\vrule
```

⁵² To be completely honest the examination by \TeX of all successive digits was not avoided, as it occurs already in the locking-up of the result, what is avoided is to spend time un-locking, and then have the macros shuffle around possibly hundreds of digit tokens rather than a few control words. Technical note: I decided (somewhat hesitantly) for reasons of optimization purposes to skip in the private `\xintexpr` format a `\protect`-ion for the `.-digits/digits[digits]` control sequences used internally. Thus in the improbable case that some macro package (such control sequence names are unavailable to the casual user) has given a meaning to one such control sequence, there is a possibility of a crash when embedding an `\xintexpr` without `\xintthe` prefix in an `\edef` (the computations by themselves do proceed perfectly correctly even if these control sequences have acquired some non `\relax` meaning).

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 $F(30)=832040$

```
\vbox{\halign{\bfseries#.\hfil&\hfil &\hfil #\cr
\ointFor* #1 in {\FibonacciSeq {90}{119}}\do
{\theindex &\ointthe#1 &
\ointiRem{\ointthe#1}{\ointtheFibxxx}\stepcounter{index}\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...;-).

7.25 `\ointForpair`, `\ointForthree`, `\ointForfour`

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

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

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

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

7.26 `\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 found one after the other on the left of `\to`. It is not expandable.

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

Special case: if after this initial expansion no brace is found immediately after `\xintAssign`, it is assumed that there is only one control sequence following `\to`, and this control sequence is then defined via `\def` to expand to the material between `\xintAssign` and `\to`. Other types of expansions are specified through an optional parameter to `\xintAssign`, see *infra*.

```

\xintAssign \xintiDivision{10000000000000}{133333333}\to\Q\R
\meaning\Q:macro:->7500, \meaning\R: macro:->2500
\xintAssign \xintiPow {7}{13}\to\SevenToThePowerThirteen
\SevenToThePowerThirteen=96889010407
(same as \edef\SevenToThePowerThirteen{\xintiPow {7}{13}})

```

`\xintAssign` admits since 1.09i an optional parameter, for example `\xintAssign [e]... or \xintAssign [oo] ...`. The latter means that the definitions of the macros initially on the right of `\to` will be made with `\oodef` which expands twice the replacement text. The default is simply to make the definitions with `\def`, corresponding to an empty optional parameter []. Possibilities: [], [g], [e], [x], [o], [go], [oo], [goo], [f], [gf].

In all cases, recall that `\xintAssign` starts with an *f*-expansion of what comes next; this produces some list of tokens or braced items, and the optional parameter only intervenes to decide the expansion type to be applied then to each one of these items.

Note: prior to release 1.09j, `\xintAssign` did an `\edef` by default, but it now does `\def`. Use the optional parameter [e] to force use of `\edef`.

It is known that `\xintAssign` is very picky and does not want a space before the `\to` or will give a surprising result if the material prior to `\to` does not start with a brace but does contain brace pairs. Next release of *xinttools* will presumably modify these ‘features’.

7.27 `\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 5, `\Bez{1}` to 1000, `\Bez{2}` to 113, `\Bez{3}` to -20, `\Bez{4}` to -177, and `\Bez{5}` to 1: $(-20) \times 1000 - (-177) \times 113 = 1$. This macro is incompatible with expansion-only contexts.

`\xintAssignArray` admits now an optional parameter, for example `\xintAssignArray [e]...`. This means that the definitions of the macros will be made with `\edef`. The default is [], which makes the definitions 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`.

Note that prior to release 1.09j each item (token or braced material) was submitted to an `\edef`, but the default is now to use `\def`.

7.28 `\xintRelaxArray`

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

7.29 `\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`.

These tools are provided as it is sometimes wasteful (from the point of view of running time) to do an `\edef` when one knows that the contents expand in only two steps for example, as is the case with all (except `\xintloop` and `\xintilloop`) the expandable macros of the *xint* packages. Each will be defined only if *xinttools* finds them currently undefined. They can be prefixed with `\global`.

7.30 The Quick Sort algorithm illustrated

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

If the interest is only in \TeX integers, then one should replace the macros `\QSMORE`, `QSEQUAL`, `QSLess` with versions using the `etoolbox` (\LaTeX only) `\ifnumgreater`, `\ifnumequal` and `\ifnumless` conditionals rather than `\xintifGt`, `\xintifEq`, `\xintifLt`.

```
% THE QUICK SORT ALGORITHM EXPANDABLY
% \usepackage{xintfrac} in the preamble (latex), or \input xintfrac.sty (Plain)
\catcode`@ 11 % = \makeatletter
\def\QSMORE #1#2{\xintifGt {#2}{#1}{{#2}}{{#1}} }
% the spaces stop the \romannumeral-`0 done by \xintapplyunbraced each time
```

```

% it applies its macro argument to an item
\def\QSEqual #1#2{\xintifEq {#2}{#1}{{#2}}{ }}
\def\QSLess #1#2{\xintifLt {#2}{#1}{{#2}}{ }}
%
\def\QSfull {\romannumeral0\qsfull }
\def\qsfull #1{\expandafter\qsfull@a\expandafter{\romannumeral-`0#1}}
\def\qsfull@a #1{\expandafter\qsfull@b\expandafter {\xintLength {#1}}{#1}}
\def\qsfull@b #1{\ifcase #1
      \expandafter\qsfull@empty
    \or\expandafter\qsfull@single
    \else\expandafter\qsfull@c
    \fi }
\def\qsfull@empty #1{ %} the space stops the \QSfull \romannumeral0
\def\qsfull@single #1{ #1}
\def\qsfull@c #1{\qsfull@ci #1\undef {#1}}% we pick up the first as Pivot
\def\qsfull@ci #1#2\undef {\qsfull@d {#1}}
\def\qsfull@d #1#2{\expandafter\qsfull@e\expandafter
      {\romannumeral0\qsfull {\xintApplyUnbraced {\QSMORE {#1}}{#2}}}%
      {\romannumeral0\xintapplyunbraced {\QSEqual {#1}}{#2}}}%
      {\romannumeral0\qsfull {\xintApplyUnbraced {\QSLess {#1}}{#2}}}%
}
\def\qsfull@e #1#2#3{\expandafter\qsfull@f\expandafter {#2}{#3}{#1}}
\def\qsfull@f #1#2#3{\expandafter\space #2#1#3}
\catcode`@ 12 % = \makeatother
% EXAMPLE
\begingroup
\edef\z {\QSfull {{1.0}{0.5}{0.3}{1.5}{1.8}{2.0}{1.7}{0.4}{1.2}{1.4}%
      {1.3}{1.1}{0.7}{1.6}{0.6}{0.9}{0.8}{0.2}{0.1}{1.9}}}
\printnumber{\meaning\z}

\def\A {3.123456789123456789}\def\B {3.123456789123456788}
\def\C {3.123456789123456790}\def\D {3.123456789123456787}
\expandafter\def\expandafter\Z\expandafter
  {\romannumeral0\qsfull {{\A}{\B}{\C}{\D}}}% \A is braced to not be expanded
\printnumber{\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}

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

```

% in LaTeX preamble:
% \usepackage{xintfrac}
% \usepackage{color}
% or, when using Plain TeX:
% \input xintfrac.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{PIVOT}{RGB}{109,8,57}
% Start of macro definitions
\catcode`@ 11 % = \makeatletter in latex
\def\QSMORE #1#2{\xintifGt {#2}{#1}{{#2}}{ }}% space will be gobbled

```

7 Commands of the *xinttools* package

```

\def\QSEqual #1#2{\xintifEq {#2}{#1}{{#2}}{ }}
\def\QSLess #1#2{\xintifLt {#2}{#1}{{#2}}{ }}
%
\def\QS@a #1{\expandafter \QS@b \expandafter {\xintLength {#1}}{#1}}
\def\QS@b #1{\ifcase #1
    \expandafter\QS@empty
    \or\expandafter\QS@single
    \else\expandafter\QS@c
    \fi }
\def\QS@empty #1{}
\def\QS@single #1{\QS@Ir {#1}}
\def\QS@c #1{\QS@d #1!{#1}} % we pick up the first as pivot.
\def\QS@d #1#2!\{\QS@e {#1}} % #1 = first element, #3 = list
\def\QS@e #1#2{\expandafter\QS@f\expandafter
    {\romannumeral0\xintapplyunbraced {\QSMORE {#1}}{#2}}%
    {\romannumeral0\xintapplyunbraced {\QSEqual {#1}}{#2}}%
    {\romannumeral0\xintapplyunbraced {\QSLess {#1}}{#2}}}
\def\QS@f #1#2#3{\expandafter\QS@g\expandafter {#2}{#3}{#1}}
% #2= elements < pivot, #1 = elements = pivot, #3 = elements > pivot
% Here \QSLr, \QSIr, \QSR have been let to \relax, so expansion stops.
\def\QS@g #1#2#3{\QSLr {#2}\QSIr {#1}\QSRr {#3}}
%
\def\DecoLEFT #1{\xintFor* ##1 in {#1} \do {\colorbox{LEFT}{##1}}}
\def\DecoINERT #1{\xintFor* ##1 in {#1} \do {\colorbox{INERT}{##1}}}
\def\DecoRIGHT #1{\xintFor* ##1 in {#1} \do {\colorbox{RIGHT}{##1}}}
\def\DecoPivot #1{\begin{group}\color{Pivot}\advance\fbboxsep-\fbboxrule\fbbox{#1}\end{group}}
\def\DecoLEFTwithPivot #1{%
    \xintFor* ##1 in {#1} \do {\xintifForFirst {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}}
\def\DecoRIGHTwithPivot #1{%
    \xintFor* ##1 in {#1} \do {\xintifForFirst {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}}
%
\def\QS@initialize #1{\def\QS@list{\QSRr {#1}}\let\QSRr\DecoRIGHT
    \par\centerline{\QS@list}}
\def\QS@oneStep {\let\QSLr\DecoLEFTwithPivot \let\QSIr\DecoINERT \let\QSRr\DecoRIGHTwithPivot
    \centerline{\QS@list}%
    \def\QSLr {\noexpand\QS@a}\let\QSIr\relax\def\QSRr {\noexpand\QS@a}%
    \edef\QS@list{\QS@list}%
    \let\QSLr\relax\let\QSRr\relax
    \edef\QS@list{\QS@list}%
    \let\QSLr\DecoLEFT \let\QSIr\DecoINERT \let\QSRr\DecoRIGHT
    \centerline{\QS@list}}
\catcode\@ 12 % = \makeatother in latex
% End of macro definitions.
% The next line is for xint.pdf use only.
\normalcolor\phantomsection\label{quicksort}
% Start of Example
\begin{group}\offinterlineskip
\small
\QS@initialize {{1.0}{0.5}{0.3}{1.5}{1.8}{2.0}{1.7}{0.4}{1.2}{1.4}{1.3}{1.1}{0.7}{1.6}{0.6}{0.9}{0.8}{0.2}{0.1}{1.9}}
    {{1.3}{1.1}{0.7}{1.6}{0.6}{0.9}{0.8}{0.2}{0.1}{1.9}}
\QS@oneStep\QS@oneStep\QS@oneStep\QS@oneStep\QS@oneStep
\end{group}

```

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

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

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

```
\normalcolor
\makeatletter
\def\QS@c #1{\expandafter\QS@e\expandafter {\romannumeral0\xintnthelt {-1}{#1}}{#1}}
\def\DecoLEFTwithPivot #1{%
  \xintFor* ##1 in {#1} \do {\xintifForLast {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}
\def\DecoRIGHTwithPivot #1{%
  \xintFor* ##1 in {#1} \do {\xintifForLast {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}
\def\QSinitialize #1{\def\QS@list{\QSLr {#1}}\let\QSLr\DecoLEFT\par\centerline{\QS@list}}
\makeatother
\begin{group}\offinterlineskip
\small
\QSinitialize {{1.0}{0.5}{0.3}{1.5}{1.8}{2.0}{1.7}{0.4}{1.2}{1.4}%
               {1.3}{1.1}{0.7}{1.6}{0.6}{0.9}{0.8}{0.2}{0.1}{1.9}}
\QSoneStep\QSoneStep\QSoneStep\QSoneStep\QSoneStep
\QSoneStep\QSoneStep\QSoneStep\QSoneStep\QSoneStep
\end{group}
```

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


It is possible to modify this code to let it do `\QSonestep` repeatedly and stop automatically when the sort is finished.⁵³

8 Commands of the **xintcore** package

.1	<code>\xintNum</code>	65	.3	<code>\xintiOpp, \xintiiOpp</code>	65
.2	<code>\xintSgn, \xintiiSgn</code>	65	.4	<code>\xintiAbs, \xintiiAbs</code>	65

⁵³ <http://tex.stackexchange.com/a/142634/4686>

.5	<code>\xintiAdd, \xintiiAdd</code>	65	.11	<code>\xintiQuo, \xintiiQuo</code>	66
.6	<code>\xintiSub, \xintiiSub</code>	65	.12	<code>\xintiRem, \xintiiRem</code>	66
.7	<code>\xintiMul, \xintiiMul</code>	65	.13	<code>\xintiDivRound, \xintiiDivRound</code>	66
.8	<code>\xintiSqr, \xintiiSqr</code>	65	.14	<code>\xintiDivTrunc, \xintiiDivTrunc</code>	67
.9	<code>\xintiPow, \xintiiPow</code>	65	.15	<code>\xintiMod, \xintiiMod</code>	67
.10	<code>\xintiDivision, \xintiiDivision</code>	66	.16	<code>\xintiInc, \xintiDec</code>	67

 Prior to release 1.1 the macros which are now included in the separate package `xintcore` were part of `xint`. Package `xintcore` is automatically loaded by `xint`.

`xintcore` provides the five basic arithmetic operations on big integers: addition, subtraction, multiplication, division and powers. Division may be either rounded (`\xintiiDivRound`) (the rounding of 0.5 is 1 and the one of -0.5 is -1) or Euclidean (`\xintiiQuo`) (which for positive operands is the same as truncated division), or truncated (`\xintiiDivTrunc`).

In the description of the macros the `{N}` and `{M}` symbols stand for explicit (big) integers within braces or more generally any control sequence (possibly within braces) *f-expanding* to such a big integer.


The macros with a single *i* in their names parse their arguments automatically through `\xintNum`. This type of expansion applied to an argument is signaled by a *f* in the margin. The accepted input format is then a sequence of plus and minus signs, followed by some string of zeroes, followed by digits.

Changed → If `xintfrac` additionally to `xintcore` is loaded, `\xintNum` becomes a synonym to `\xintTTrunc`; this means that arbitrary fractions will be accepted as arguments of the macros with a single *i* in their names, but get truncated to integers before further processing. The format of the output will be as with only `xint` loaded. The only extension is in allowing a wider variety of inputs.

The macros with *ii* in their names have arguments which will only be *f*-expanded, but will not be parsed via `\xintNum`. Arguments of this type are signaled by the margin annotation *f*. For such big integers only one minus sign and no plus sign, nor leading zeros, are accepted. -0 is not valid in this strict input format. Loading `xintfrac` does not bring any modification to these macros whether for input or output.

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

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

 `xintcore` also provides macros `\xintAdd, \xintMul, ...` as synonyms to `\xintiAdd, \xintiMul, ...`. Their usage is deprecated for the following reason: with `xintfrac` loaded they become the routines dealing fully with fractions on input. But this means that they now use fraction format on output, even with integer arguments.

Due to this variability of the output format on whether the document uses only `xint` or loads additionally `xintfrac`, code using these macros is fragile, because loading at some later date a package which itself loads `xintfrac` or `xintexpr` will modify their output format, and this is catastrophic for example in locations expanded by `\ifnum`, or even in arguments to those other macros of `xint` with *ii* in their names.

Prefer thus, when writing code loading only `xintcore` or `xint`, to use the macros `\xintiAdd, \xintiMul, ...`, or `\xintiiAdd, \xintiiMul, ...`. They are guaranteed to always output an integer without a trailing `/B[n]`. The latter have the less overhead, and the former do not complain, if `xintfrac` is loaded, even if used with true fractions, as they will then truncate their arguments to integers.

Changed →

It was an error for the `xint` package (now `xintcore`) to provide macros `\xintAdd, \xintMul, \xintSub, ...`. They should be used only with `xintfrac` loaded.

The \star 's in the margin are there to remind of the complete expandability, even *f*-expandability of the macros, as discussed in [subsection 3.3](#).

8.1 `\xintNum`

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

`\xintNum{+---+-----000000000367941789479}=-367941789479`

All *xint* macros with a single *i* in their names, such as `\xintiAdd`, `\xintiMul` apply `\xintNum` to their arguments.

Changed \rightarrow When *xintfrac* is loaded, `\xintNum` becomes a synonym to `\xintTTrunc`.

8.2 `\xintSgn`, `\xintiiSgn`

f \star `\xintiiSgn{N}` returns 1 if the number is positive, 0 if it is zero and -1 if it is negative. It skips the `\xintNum` overhead.

Num f \star `\xintSgn` is the variant using `\xintNum` and getting extended by *xintfrac* to fractions.

8.3 `\xintiOpp`, `\xintiiOpp`

Num f \star `\xintiOpp{N}` return the opposite $-N$ of the number *N*. `\xintOpp` is initially a synonym but gets extended by *xintfrac* to fractions (its output format will be a fraction even if the argument is an integer) and `\xintiiOpp` is the strict integer-only variant which skips the `\xintNum` overhead.

f \star `\xintiiOpp` is the strict integer-only variant which skips the `\xintNum` overhead.

8.4 `\xintiAbs`, `\xintiiAbs`

Num f \star `\xintiAbs{N}` returns the absolute value of the number. `\xintAbs` is a synonym but gets modified by

f \star *xintfrac*. `\xintiiAbs` skips the `\xintNum` overhead.

8.5 `\xintiAdd`, `\xintiiAdd`

Num Num f f \star `\xintiAdd{N}{M}` returns the sum of the two numbers. `\xintAdd` is initially a synonym but gets extended by *xintfrac*. `\xintiiAdd` skips the `\xintNum` overhead.

ff \star `\xintiiAdd` skips the `\xintNum` overhead.

8.6 `\xintiSub`, `\xintiiSub`

Num Num f f \star `\xintiSub{N}{M}` returns the difference $N-M$. `\xintSub` is initially a synonym but gets extended by *xintfrac*. `\xintiiSub` skips the `\xintNum` overhead.

ff \star *xintfrac*. `\xintiiSub` skips the `\xintNum` overhead.

8.7 `\xintiMul`, `\xintiiMul`

Num Num f f \star `\xintiMul{N}{M}` returns the product of the two numbers. `\xintMul` is the initial synonym modified by *xintfrac*, and `\xintiiMul` skips the `\xintNum` overhead.

ff \star by *xintfrac*, and `\xintiiMul` skips the `\xintNum` overhead.

8.8 `\xintiSqr`, `\xintiiSqr`

Num f \star `\xintiSqr{N}` returns the square. `\xintSqr` is the initial synonym extended by *xintfrac* to fractions. `\xintiiSqr` skips the `\xintNum` overhead.

f \star tions. `\xintiiSqr` skips the `\xintNum` overhead.

8.9 `\xintiPow`, `\xintiiPow`

Num f x \star `\xintiPow{N}{x}` returns N^x . When *x* is zero, this is 1. If $N=0$ and $x<0$, if $|N|>1$ and $x<0$, or if $|N|>1$ and $x>1000000$, then an error is raised.

The $x>1000000$ condition should perhaps be made more strict: as it stands it allows launching operations taking hours to complete. Indeed, observe that 2^{500000} already has 15052 digits; as it turns out each exact multiplication done via `\xintiiMul` of two numbers with one thousand digits

each already takes of the order of seconds, and it would take hours for arguments each with circa 15000 digits. Perhaps some completely expandable but not *f*-expandable variants could fare better?

f *x* ^{num} ★ `\xintiiPow` is an integer only variant skipping the `\xintNum` overhead, it produces the same result as `\xintiPow` with stricter assumptions on the inputs, and is thus a tiny bit faster.

`\xintPow` is the initial synonym of `\xintiiPow` which gets extended by *xintfrac* to fractions (see also `\xintFloatPow` for which the exponent must still obey the \TeX bound and `\xintFloatPower` which has no restriction at all on the size of the exponent). Negative exponents do not then raise errors anymore. The float version is able to deal with things such as $2^{999999999}$ without any problem. For example `\xintFloatPow[4]{2}{50000}=3.161e15051` and `\xintFloatPow[4]{2}{999999999}=2.306e301029995`.⁵⁴

Within an `\xintiexpr`..`\relax` the infix operator `^` is mapped to `\xintiiPow`; within an `\xintexpr`-expression it is mapped to `\xintPow` (as extended by *xintfrac*); in `\xintfloatexpr`, it is mapped to `\xintFloatPower`.

8.10 `\xintiDivision`, `\xintiiDivision`

ff ★ `\xintiiDivision{N}{M}` returns `{quotient Q}{remainder R}`. This is euclidean division: $N = QM + R$, $0 \leq R < |M|$. So the remainder is always non-negative and the formula $N = QM + R$ always holds independently of the signs of *N* or *M*. Division by zero is an error (even if *N* vanishes) and returns `{0}{0}`. It skips the overhead of parsing via `\xintNum`.

Num Num
f *f* ★ `\xintiDivision` submits its arguments to `\xintNum` and is extended by *xintfrac* to accept fractions on input, which it truncates first, and is not to be confused with the *xintfrac* macro `\xintDiv` which divides one fraction by another.

Changed → Note: `\xintDivision` was the former name of `\xintiDivision`. Its use is deprecated since release 1.1.

8.11 `\xintiQuo`, `\xintiiQuo`

ff ★ `\xintiiQuo{N}{M}` returns the quotient from the euclidean division. It skips the overhead of parsing via `\xintNum`.

Num Num
f *f* ★ `\xintiQuo` submits its arguments to `\xintNum` and is extended by *xintfrac* to accept fractions on input, which it truncates first.

Changed → Note: `\xintQuo` is the former name of `\xintiQuo`. Its use is deprecated.

8.12 `\xintiRem`, `\xintiiRem`

ff ★ `\xintiiRem{N}{M}` returns the remainder from the euclidean division. It skips the overhead of parsing via `\xintNum`.

Num Num
f *f* ★ `\xintiRem` submits its arguments to `\xintNum` and is extended by *xintfrac* to accept fractions on input, which it truncates first.

Changed → Note: `\xintRem` is the former name of `\xintiRem`. Its use is deprecated.

8.13 `\xintiDivRound`, `\xintiiDivRound`

ff ★ `\xintiiDivRound{N}{M}` returns the rounded value of the algebraic quotient N/M of two big integers. The rounding of half integers is towards the nearest integer of bigger absolute value. The macro skips the overhead of parsing via `\xintNum`. The rounding is away from zero.

Num Num
f *f* ★ `\xintiDivRound` submits its arguments to `\xintNum`. It is extended by *xintfrac* to accept fractions on input, which it truncates first before computing the rounded quotient.

⁵⁴ On my laptop `\xintiiPow {2}{9999}` obtains all 3010 digits in about ten or eleven seconds. In contrast, the float versions for 8, 16, 24, or even more significant figures, do their jobs in less than one hundredth of a second (1.09j; we used in the text only four significant digits only for reasons of space, not time.) This is done without `log/exp` which are not (yet?) implemented in *xintfrac*. The \LaTeX 3 *l3fp* package does this with `log/exp` and is ten times faster, but allows only 16 significant figures and the (exactly represented) floating point numbers must have their exponents limited to ± 9999 .

8.14 **\xintiDivTrunc**, **\xintiiDivTrunc**

ff ★ **\xintiiDivTrunc**{N}{M} computes the truncation towards zero of the algebraic quotient N/M. It skips the overhead of parsing the operands with **\xintNum**. For M > 0 it is the same as **\xintiiQuo**.
 $\$ \backslash \text{xintiiQuo} \{1000\}\{-57\}, \backslash \text{xintiiDivRound} \{1000\}\{-57\}, \backslash \text{xintiiDivTrunc} \{1000\}\{-57\} \$$

Num Num
f f ★

-17, -18, -17
\xintiDivTrunc submits first its arguments to **\xintNum**.

8.15 **\xintiMod**, **\xintiiMod**

ff ★ **\xintiiMod**{N}{M} computes $N - M * t(N/M)$, where $t(N/M)$ is the algebraic quotient truncated towards zero. The macro skips the overhead of parsing the operands with **\xintNum**. For M > 0 it is the same as **\xintiiRem**.

$\$ \backslash \text{xintiiRem} \{1000\}\{-57\}, \backslash \text{xintiiMod} \{1000\}\{-57\},$
 $\backslash \text{xintiiRem} \{-1000\}\{57\}, \backslash \text{xintiiMod} \{-1000\}\{57\} \$$

Num Num
f f ★

31, 31, 26, -31
\xintiMod submits first its arguments to **\xintNum**.

8.16 **\xintInc**, **\xintDec**

f ★ **\xintInc**{N} is N+1 and **\xintDec**{N} is N-1. These macros remain integer-only, even with **xintfrac** loaded. They skip the overhead of parsing via **\xintNum**.

9 Commands of the **xint** package

.1	\xintRev	68	.29	\xintifSgn , \xintiiifSgn	71
.2	\xintLen	68	.30	\xintifZero , \xintiiifZero	71
.3	\xintDigitsOf	68	.31	\xintifNotZero , \xintiiifNotZero	71
.4	\xintCmp , \xintiiCmp	68	.32	\xintifOne , \xintiiifOne	71
.5	\xintEq , \xintiiEq	68	.33	\xintifTrueAelseB , \xintifFalseAelseB ..	72
.6	\xintNeq , \xintiiNeq	68	.34	\xintifCmp , \xintiiifCmp	72
.7	\xintGt , \xintiiGt	69	.35	\xintifEq , \xintiiifEq	72
.8	\xintLt , \xintiiLt	69	.36	\xintifGt , \xintiiifGt	72
.9	\xintLtorEq , \xintiiLtorEq	69	.37	\xintifLt , \xintiiifLt	72
.10	\xintGtorEq , \xintiiGtorEq	69	.38	\xintifOdd , \xintiiifOdd	72
.11	\xintIsZero , \xintiiIsZero	69	.39	\xintiFac	72
.12	\xintNot	69	.40	\xintiiFDg	73
.13	\xintIsNotZero , \xintiiIsNotZero	69	.41	\xintiiLDg	73
.14	\xintIsOne , \xintiiIsOne	69	.42	\xintiiMON , \xintiiMMON	73
.15	\xintAND	69	.43	\xintiiOdd	73
.16	\xintOR	69	.44	\xintiiEven	73
.17	\xintXOR	69	.45	\xintiSqrt , \xintiiSqrt , \xintiiSqrtR , \xintiSquareRoot , \xintiiSquareRoot	73
.18	\xintANDof	69	.46	\xintDouble , \xintHalf	74
.19	\xintORof	70	.47	\xintDSL	74
.20	\xintXORof	70	.48	\xintDSR	74
.21	\xintGeq	70	.49	\xintDSH	74
.22	\xintiMax , \xintiiMax	70	.50	\xintDSHr , \xintDSx	74
.23	\xintiMin , \xintiiMin	70	.51	\xintDecSplit	75
.24	\xintiMaxof	70	.52	\xintDecSplitL	75
.25	\xintiMinof	70	.53	\xintDecSplitR	75
.26	\xintiiSum	70	.54	\xintiiE	75
.27	\xintiiPrd	71			
.28	\xintSgnFork	71			

Version 1.0 was released 2013/03/28. This is 1.1a of 2014/11/07. The core arithmetic macros have
 Changed → been moved to separate package `xintcore`, which is automatically loaded by `xint`.

See the documentation of `xintcore` or subsection 3.3 for the significance of the ^{Num} f , f , ^{num} x and \star margin annotations and some important background information.

9.1 `\xintRev`

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

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

9.2 `\xintLen`

^{Num} $f \star$ `\xintLen{N}` returns the length of the number, not counting the sign.

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

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

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

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

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

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

9.3 `\xintDigitsOf`

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

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

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

9.4 `\xintCmp`, `\xintiiCmp`

^{Num} $f \star$ `\xintCmp{N}{M}` returns 1 if $N > M$, 0 if $N = M$, and -1 if $N < M$. Extended by `xintfrac` to fractions (its output naturally still being either 1, 0, or -1).

$ff \star$ `\xintiiCmp` skips the `\xintNum` overhead.

9.5 `\xintEq`, `\xintiiEq`

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

$ff \star$ `\xintiiEq` skips the `\xintNum` overhead.

9.6 `\xintNeq`, `\xintiiNeq`

^{Num} $f \star$ `\xintNeq{N}{M}` returns 0 if $N = M$, 1 otherwise. Extended by `xintfrac` to fractions.

$ff \star$ `\xintiiNeq` skips the `\xintNum` overhead.

9.7 `\xintGt`, `\xintiiGt`

Num Num
 $f f \star$ `\xintGt{N}{M}` returns 1 if $N > M$, 0 otherwise. Extended by `xintfrac` to fractions.
 $ff \star$ `\xintiiGt` skips the `\xintNum` overhead.

9.8 `\xintLt`, `\xintiiLt`

Num Num
 $f f \star$ `\xintLt{N}{M}` returns 1 if $N < M$, 0 otherwise. Extended by `xintfrac` to fractions.
 $ff \star$ `\xintiiLt` skips the `\xintNum` overhead.

9.9 `\xintLtorEq`, `\xintiiLtorEq`

Num Num
 $f f \star$ `\xintLtorEq{N}{M}` returns 1 if $N \leq M$, 0 otherwise. Extended by `xintfrac` to fractions.
 $ff \star$ `\xintiiLtorEq` skips the `\xintNum` overhead.

9.10 `\xintGtorEq`, `\xintiiGtorEq`

Num Num
 $f f \star$ `\xintGtorEq{N}{M}` returns 1 if $N \geq M$, 0 otherwise. Extended by `xintfrac` to fractions.
 $ff \star$ `\xintiiGtorEq` skips the `\xintNum` overhead.

9.11 `\xintIsZero`, `\xintiiIsZero`

Num
 $f \star$ `\xintIsZero{N}` returns 1 if $N = 0$, 0 otherwise. Extended by `xintfrac` to fractions.
 $f \star$ `\xintiiIsZero` skips the `\xintNum` overhead.

9.12 `\xintNot`

Num
 $f \star$ `\xintNot` is a synonym for `\xintIsZero`.

9.13 `\xintIsNotZero`, `\xintiiIsNotZero`

Num
 $f \star$ `\xintIsNotZero{N}` returns 1 if $N \neq 0$, 0 otherwise. Extended by `xintfrac` to fractions.
 $f \star$ `\xintiiIsNotZero` skips the `\xintNum` overhead.

9.14 `\xintIsOne`, `\xintiiIsOne`

Num
 $f \star$ `\xintIsOne{N}` returns 1 if $N = 1$, 0 otherwise. Extended by `xintfrac` to fractions.
 $f \star$ `\xintiiIsOne` skips the `\xintNum` overhead.

9.15 `\xintAND`

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

9.16 `\xintOR`

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

9.17 `\xintXOR`

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

9.18 `\xintANDof`

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

9.19 `\xintORof`

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

9.20 `\xintXORof`

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

9.21 `\xintGeq`

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

9.22 `\xintiMax`, `\xintiiMax`

$\overset{\text{Num}}{f} \overset{\text{Num}}{f} \star$ `\xintiMax{N}{M}` returns the largest of the two in the sense of the order structure on the relative integers (i.e. the right-most number if they are put on a line with positive numbers on the right): `\xintiMax {-5}{-6}=-5`. The initial synonym `\xintMax` gets modified by `xintfrac` which extends it to fractions. Its usage when only `xint` is loaded is discouraged.

$ff \star$ The `\xintiiMax` macro skips the overhead of parsing the operands with `\xintNum`.

9.23 `\xintiMin`, `\xintiiMin`

$\overset{\text{Num}}{f} \overset{\text{Num}}{f} \star$ `\xintiMin{N}{M}` returns the smallest of the two in the sense of the order structure on the relative integers (i.e. the left-most number if they are put on a line with positive numbers on the right): `\xintiMin {-5}{-6}=-6`. The initial synonym `\xintMin` gets modified by `xintfrac` which extends it to fractions. Its usage when only `xint` is loaded is discouraged.

$ff \star$ The `\xintiiMin` macro skips the overhead of parsing the operands with `\xintNum`.

9.24 `\xintiMaxof`

$f \rightarrow * \overset{\text{Num}}{f} \star$ `\xintiMaxof{{a}{b}{c}...}` returns the maximum. The list argument may be a macro, it is f -expanded first. Each item is submitted to `\xintNum` normalization. `\xintMaxof` is the initial synonym which gets extended by `xintfrac` to fractions. Do not use it when only `xint` is loaded.

9.25 `\xintiMinof`

$f \rightarrow * \overset{\text{Num}}{f} \star$ `\xintiMinof{{a}{b}{c}...}` returns the minimum. The list argument may be a macro, it is f -expanded first. Each item is submitted to `\xintNum` normalization. `\xintMinof` is the initial synonym which gets extended by `xintfrac` to fractions. Do not use it when only `xint` is loaded.

9.26 `\xintiiSum`

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

`\xintSum` is initially a synonym, it gets extended by `xintfrac` to fractions.

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

`\xintiiSum{1234567890}` = 45

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

9.27 `\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. Note: the operands are *not* parsed by `\xintNum`.

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

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

```
$2^{200}3^{100}7^{100}=\printnumber
```

```
{\xintiiPrd {\xintiPow {2}{200}}{\xintiPow {3}{100}}{\xintiPow {7}{100}}}{}
```

$2^{200}3^{100}7^{100} = 267872793166157757576627951700754840232474026637401534897445961481542641296549904900004440072407657271300001653120764065456211801435719940159033435392440282124389668222489207862988084382716133376$

With `xintexpr`, this would be easier:

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

The initial synonym `\xintPrd` is extended by `xintfrac` to fractions.

9.28 `\xintSgnFork`

xnnn ★ `\xintSgnFork{-1|0|1}{⟨A⟩}{⟨B⟩}{⟨C⟩}` expandably chooses to execute either the $\langle A \rangle$, $\langle B \rangle$ or $\langle C \rangle$ code, depending on its first argument. This first argument should be anything expanding to either -1, 0 or 1 in a non self-delimiting way (i.e. a count register must be prefixed by `\the` and a `\numexp` r...`\relax` also must be prefixed by `\the`). This utility is provided to help construct expandable macros choosing depending on a condition which one of the package macros to use, or which values to confer to their arguments.

9.29 `\xintifSgn`, `\xintiiifSgn`

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

f ★ `\xintiiifSgn` skips the `\xintNum` overhead.

9.30 `\xintifZero`, `\xintiiifZero`

Num f nn ★ `\xintifZero{⟨N⟩}{⟨IsZero⟩}{⟨IsNotZero⟩}` expandably checks if the first mandatory argument N (a number, possibly a fraction if `xintfrac` is loaded, or a macro expanding to one such) is zero or not. It then either executes the first or the second branch. Beware that both branches must be present.

f ★ `\xintiiifZero` skips the `\xintNum` overhead.

9.31 `\xintifNotZero`, `\xintiiifNotZero`

Num f nn ★ `\xintifNotZero{⟨N⟩}{⟨IsNotZero⟩}{⟨IsZero⟩}` expandably checks if the first mandatory argument N (a number, possibly a fraction if `xintfrac` is loaded, or a macro expanding to one such) is not zero or is zero. It then either executes the first or the second branch. Beware that both branches must be present.

f ★ `\xintiiifNotZero` skips the `\xintNum` overhead.

9.32 `\xintifOne`, `\xintiiifOne`

Num f nn ★ `\xintifOne{⟨N⟩}{⟨IsOne⟩}{⟨IsNotOne⟩}` expandably checks if the first mandatory argument N (a num-

ber, possibly a fraction if `xintfrac` is loaded, or a macro expanding to one such) is one or not. It then either executes the first or the second branch. Beware that both branches must be present.

`f` ★ `\xintiifOne` skips the `\xintNum` overhead.

9.33 `\xintifTrueAelseB`, `\xintifFalseAelseB`

`Num`
`f nn` ★ `\xintifTrueAelseB{N}{true branch}{false branch}` is a synonym for `\xintifNotZero`.
1. with 1.09i, the synonyms `\xintifTrueFalse` and `\xintifTrue` are deprecated and will be removed in next release.
2. These macros have no lowercase versions, use `\xintifzero`, `\xintifnotzero`.
`Num`
`f nn` ★ `\xintifFalseAelseB{N}{false branch}{true branch}` is a synonym for `\xintifZero`.

9.34 `\xintifCmp`, `\xintiifCmp`

`Num Num`
`f f nnn` ★ `\xintifCmp{A}{B}{if A<B}{if A=B}{if A>B}` compares its arguments and chooses accordingly the correct branch.
`ff` ★ `\xintiifCmp` skips the `\xintNum` overhead.

9.35 `\xintifEq`, `\xintiifEq`

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

9.36 `\xintifGt`, `\xintiifGt`

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

9.37 `\xintifLt`, `\xintiifLt`

`Num Num`
`f f nn` ★ `\xintifLt{A}{B}{YES}{NO}` checks if $A < B$ and in that case executes the YES branch. Extended to fractions (in particular decimal numbers) by `xintfrac`.
`ff` ★ `\xintiifLt` skips the `\xintNum` overhead.

9.38 `\xintifOdd`, `\xintiifOdd`

`Num`
`f nn` ★ `\xintifOdd{A}{YES}{NO}` checks if A is and odd integer and in that case executes the YES branch.
`f` ★ `\xintiifOdd` skips the `\xintNum` overhead.

The macros described next are all integer-only on input. Those with `ii` in their names skip the `\xintNum` parsing. The others, with `xintfrac` loaded, can have fractions as arguments, which will get truncated to integers via `\xintTTrunc`. On output, the macros here always produce integers (with no `/B[N]`).

9.39 `\xintiFac`

`num`
`x` ★ `\xintiFac{x}` returns the factorial. It is an error if the argument is negative or at least 10^5 .
`\xintFac` is a variant using `\xintNum` and thus, when `xintfrac` is loaded, accepting a fraction on input (but it truncates it first).

9.40 \xintiiFDg

f **★** `\xintFDg{N}` returns the first digit (most significant) of the decimal expansion. It skips the overhead of parsing via `\xintNum`. The variant `\xintFDg` uses `\xintNum` and gets extended by `xintfrac`.

9.41 \xintiLDq

`f` ★ `\xintiLDg{N}` returns the least significant digit. When the number is positive, this is the same as the remainder in the euclidean division by ten. It skips the overhead of parsing via `\xintNum`.
`Num`
`f` ★ The variant `\xintLDg` uses `\xintNum` and gets extended by `xintfrac`.

9.42 \xintiiMON, \xintiiMMON

`f★ \xintiiMON{N}` returns $(-1)^N$ and `\xintiiMMON{N}` returns $(-1)^{N-1}$. They skip the overhead of parsing via `\xintNum`.

Num ★ The variants `\xintMON` and `\xintMMON` use `\xintNum` and get extended to fractions by `xintfrac`.

9.43 \xintiiOdd

Num \star `\xintiiOdd{N}` is 1 if the number is odd and 0 otherwise. It skips the overhead of parsing via `\xint-`
 \star `Num`. `\xintOdd` is the variant using `\xintNum` and extended to fractions by `xintfrac`.

9.44 \xintiiEven

xintNum $\text{xintEven}\{N\}$ is 1 if the number is even and 0 otherwise. It skips the overhead of parsing via xintNum . xintEven is the variant using xintNum and extended to fractions by xintfrac .

9.45 `\xintiSqrt`, `\xintiiSqrt`, `\xintiiSqrtR`, `\xintiSquareRoot`, `\xintiiSquareRoot`

`\xintiSqrt{N}` returns the largest integer whose square is at most equal to `N`. `\xintiiSqrt` is the variant skipping the `\xintNum` overhead. `\xintiiSqrtR` also skips the `\xintNum` overhead and it returns the rounded, not truncated, square root.

```
\begin{itemize}[nosep]
\item \xintiiSqrt {3000000000000000000000000000000000}
\item \xintiiSqrtR {3000000000000000000000000000000000}
\item \xintiiSqrt {\xintiiE {3}{100}}
\end{itemize}
```

- 1732050807568877293
- 1732050807568877294
- 173205080756887729352744634150587236694280525381038

- 17320508075688729352744634150587236694280525381038

Num
f ★ \xintiSquareRoot{N} returns {M}{d} with $d > 0$, $M^2 - d = N$ and M smallest (hence $= 1 + \text{\xintiSqrt}\{N\}$).
f ★ \xintiiSquareRoot is the variant skipping the \xintNum overhead.

```
\xintAssign\xintiiSquareRoot {170000000000000000000000000000}\to\A\B
\xintiiSub{\xintiiSqr\A}\B=\A\string^2-\B
```

$17000000000000000000000000=4123105625618^2-2799177881924$

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

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

The macros described next are strictly for integer-only arguments. These arguments are *not* filtered via `\xintNum`. The macros are not usable with fractions, even with `xintfrac` loaded.

9.46 `\xintDouble`, `\xintHalf`

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

9.47 `\xintDSL`

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

9.48 `\xintDSR`

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

9.49 `\xintDSH`

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

9.50 `\xintDSHr`, `\xintDSx`

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

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

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

```
\xintAssign\xintDSx {-1}{-123456789}\to\M
\meaning\M:macro:->-1234567890.
\xintAssign\xintDSx {-20}{123456789}\to\M
\meaning\M:macro:->123456789000000000000000000000.
\xintAssign\xintDSx {0}{-123004321}\to\Q\R
\meaning\Q:macro:->-123004321, \meaning\R:macro:->0.
\xintDSH {0}{-123004321}=-123004321, \xintDSHr {0}{-123004321}=0
\xintAssign\xintDSx {6}{-123004321}\to\Q\R
\meaning\Q:macro:->-123, \meaning\R:macro:->4321.
\xintDSH {6}{-123004321}=-123, \xintDSHr {6}{-123004321}=4321
\xintAssign\xintDSx {8}{-123004321}\to\Q\R
\meaning\Q:macro:->-1, \meaning\R:macro:->23004321.
\xintDSH {8}{-123004321}=-1, \xintDSHr {8}{-123004321}=23004321
\xintAssign\xintDSx {9}{-123004321}\to\Q\R
\meaning\Q:macro:->0, \meaning\R:macro:->-123004321.
\xintDSH {9}{-123004321}=0, \xintDSHr {9}{-123004321}=-123004321
```

9.51 \xintDecSplit

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

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

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

9.52 \xintDecSplitL

$\text{num}_X f \star \text{\texttt{\textbackslash xintDecSplitL}\{x\}\{N\}}$ returns the first piece after the action of `\xintDecSplit`.

9.53 \xintDecSplitR

$\sum_{x \in X} f \star \text{\texttt{\textbackslash xintDecSplitR}\{x\}\{N\}}$ returns the second piece after the action of `\xintDecSplit`.

9.54 \xintiE

$f^{\text{num}}_x \star \backslash \text{xintiiE}\{N\}\{x\}$ serves to add zeros to the right of N .

`\xintiiE {123}{89}`

[illegible]

10 Commands of the **xintfrac** package

.1	\xintNum	76	.10	\xintFrac	78
.2	\xintifInt	77	.11	\xintSignedFrac	78
.3	\xintLen	77	.12	\xintFwOver	78
.4	\xintRaw	77	.13	\xintSignedFwOver	78
.5	\xintPRaw	77	.14	\xintIrr	79
.6	\xintNumerator	77	.15	\xintJrr	79
.7	\xintDenominator	77	.16	\xintTrunc	79
.8	\xintRawWithZeros	78	.17	\xintiTrunc	79
.9	\xintREZ	78	.18	\xintTTrunc	79

.19	<code>\xintXTrunc</code>	80	.39	<code>\xintPow</code>	84
.20	<code>\xintRound</code>	81	.40	<code>\xintFloatPow</code>	84
.21	<code>\xintiRound</code>	81	.41	<code>\xintFloatPower</code>	84
.22	<code>\xintFloor</code> , <code>\xintiFloor</code>	81	.42	<code>\xintFloatSqrt</code>	84
.23	<code>\xintCeil</code> , <code>\xintiCeil</code>	82	.43	<code>\xintSum</code>	85
.24	<code>\xintTFrac</code>	82	.44	<code>\xintPrd</code>	85
.25	<code>\xintE</code>	82	.45	<code>\xintCmp</code>	85
.26	<code>\xintFloatE</code>	82	.46	<code>\xintIsOne</code>	85
.27	<code>\xintDigits</code> , <code>\xinttheDigits</code>	82	.47	<code>\xintGeq</code>	85
.28	<code>\xintFloat</code>	82	.48	<code>\xintMax</code>	85
.29	<code>\xintAdd</code>	83	.49	<code>\xintMin</code>	85
.30	<code>\xintFloatAdd</code>	83	.50	<code>\xintMaxof</code>	86
.31	<code>\xintSub</code>	83	.51	<code>\xintMinof</code>	86
.32	<code>\xintFloatSub</code>	83	.52	<code>\xintAbs</code>	86
.33	<code>\xintMul</code>	83	.53	<code>\xintSgn</code>	86
.34	<code>\xintSqr</code>	83	.54	<code>\xintOpp</code>	86
.35	<code>\xintFloatMul</code>	83	.55	<code>\xintiDivision</code> , <code>\xintiQuo</code> , <code>\xintiRem</code> , <code>\xintFDg</code> , <code>\xintLDg</code> , <code>\xintMON</code> , <code>\xintMMON</code> , <code>\xintOdd</code>	86
.36	<code>\xintDiv</code>	83			
.37	<code>\xintFloatDiv</code>	83			
.38	<code>\xintFac</code>	84			

This package was first included in release 1.03 (2013/04/14) of the `xint` bundle. The general rule of the bundle that each macro first expands (what comes first, fully) each one of its arguments applies.

Frac
f `f` stands for an integer or a fraction (see subsection 4.1 for the accepted input formats) or something which expands to an integer or fraction. It is possible to use in the numerator or the denominator of `f` count registers and even expressions with infix arithmetic operators, under some rules which are explained in the previous [Use of count registers](#) section.

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

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

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

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

10.1 `\xintNum`

f ★ The macro from `xint` is made a synonym to `\xintTTrunc`. (Changed!)⁵⁵

The original (which normalizes big integers to strict format) is still available as `\xintiNum`. It is imprudent to apply `\xintNum` to numbers with a large power of ten given either in scientific notation or with the `[n]` notation, as the macro will according to its definition add all the needed zeroes to produce an explicit integer in strict format.

⁵⁵ In earlier releases than 1.1, `\xintNum` did `\xintIrr` and then complained if the denominator was not 1, else, it silently removed the denominator.

10.2 `\xintifInt`

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

10.3 `\xintLen`

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

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

10.4 `\xintRaw`

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]
```

10.5 `\xintPRaw`

`PRaw` stands for ‘pretty raw’. It does not show the `[n]` if `n=0` and does not show the `B` if `B=1`.

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

See also `\xintFrac` (or `\xintFwOver`) for math mode. As is exemplified above the `\xintIrr` macro which puts the fraction into irreducible form does not remove the `/1` if the fraction is an integer. One can use `\xintNum{f}` or `\xintPRaw{\xintIrr{f}}` which produces the same output only if `f` is an integer (after simplification).

10.6 `\xintNumerator`

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

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

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

10.7 `\xintDenominator`

This returns the denominator corresponding to the internal representation of the fraction:⁵⁶

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

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

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

10.8 `\xintRawWithZeros`

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

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

10.9 `\xintREZ`

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

```
\xintREZ {178000/256000000[17]}=178/256[15]
\xintREZ {17800000000000e30/25600000000000e15}=178/256[15]
```

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

10.10 `\xintFrac`

Frac f ★ This is a `TeX` only command, to be used in math mode only. It will print a fraction, internally represented as something equivalent to `A/B[n]` as `\frac {A}{B}10^n`. The power of ten is omitted when `n=0`, the denominator is omitted when it has value one, the number being separated from the power of ten by a `\cdot`. `\xintFrac {178.000/256000000}$` gives $\frac{178000}{25600000}10^{-3}$, `\xintFrac {178.000/1}$` gives $178000 \cdot 10^{-3}$, `\xintFrac {3.5/5.7}$` gives $\frac{35}{57}$, and `\xintFrac {\xintNum {\xintFac{10}}/\xintiSqr{\xintFac {5}}}$` gives 252. As shown by the examples, simplification of the input (apart from removing the decimal points and moving the minus sign to the numerator) is not done automatically and must be the result of macros such as `\xintIrr`, `\xintREZ`, or `\xintNum` (for fractions being in fact integers.)

10.11 `\xintSignedFrac`

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

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

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

10.12 `\xintFwOver`

Frac f ★ This does the same as `\xintFrac` except that the `\over` primitive is used for the fraction (in case the denominator is not one; and a pair of braces contains the `A\over B` part). `\xintFwOver {178.000/256000000}$` gives $\frac{178000}{25600000}10^{-3}$, `\xintFwOver {178.000/1}$` gives $178000 \cdot 10^{-3}$, `\xintFwOver {3.5/5.7}$` gives $\frac{35}{57}$, and `\xintFwOver {\xintNum {\xintFac{10}}/\xintiSqr{\xintFac {5}}}$` gives 252.

10.13 `\xintSignedFwOver`

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

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

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

10.14 `\xintIrr`

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

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

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

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

10.15 `\xintJrr`

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

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

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

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

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

10.16 `\xintTrunc`

$\frac{\text{num}}{x} \frac{\text{Frac}}{f}$ ★ `\xintTrunc{x}{f}` returns the integral part, a dot, and then the first `x` digits of the decimal expansion of the fraction `f`. The argument `x` should be non-negative.

In the special case when `f` evaluates to 0, the output is 0 with no decimal point nor decimal digits, else the post decimal mark digits are always printed. A non-zero negative `f` which is smaller in absolute value than 10^{-x} will give -0.000....

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

The digits printed are exact up to and including the last one.

10.17 `\xintiTrunc`

$\frac{\text{num}}{x} \frac{\text{Frac}}{f}$ ★ `\xintiTrunc{x}{f}` returns the integer equal to 10^x times what `\xintTrunc{x}{f}` would produce.

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

The difference between `\xintTrunc{0}{f}` and `\xintiTrunc{0}{f}` is that the latter never has the decimal mark always present in the former except for `f=0`. And `\xintTrunc{0}{-0.5}` returns “-0.” whereas `\xintiTrunc{0}{-0.5}` simply returns “0”.

10.18 `\xintTTrunc`

$\frac{\text{Frac}}{f}$ ★ `\xintTTrunc{f}` truncates to an integer (truncation towards zero; `New`). This is the same as `\xintiTrunc {0}{f}` and as `\xintNum`.

10.19 `\xintXTrunc`

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

`\xintXTrunc{x}{f}` is completely expandable but not *f*-expandable, as is indicated by the hollow star in the margin. It can not be used as argument to the other package macros, but is designed to be used inside an `\edef`, or rather a `\write`. Here is an example session where the user after some warming up checks that $1/66049 = 1/257^2$ has period $257 * 256 = 65792$ (it is also checked here that this is indeed the smallest period).

```
xxx:_xint $ etex -jobname worksheet-66049
This is pdfTeX, Version 3.1415926-2.5-1.40.14 (TeX Live 2013)
restricted \write18 enabled.
**\relax
entering extended mode

*\input xintfrac.sty
(./xintfrac.sty (./xint.sty (./xinttools.sty)))
*\message{\xintTrunc {100}{1/71}}% Warming up!

0.01408450704225352112676056338028169014084507042253521126760563380281690140845
07042253521126760563380
*\message{\xintTrunc {350}{1/71}}% period is 35

0.01408450704225352112676056338028169014084507042253521126760563380281690140845
0704225352112676056338028169014084507042253521126760563380281690140845070422535
2112676056338028169014084507042253521126760563380281690140845070422535211267605
6338028169014084507042253521126760563380281690140845070422535211267605633802816
901408450704225352112676056338028169
*\edef\Z {\xintXTrunc {65792}{1/66049}}% getting serious...

*\def\trim 0.{ }\oodef\Z {\expandafter\trim\Z}% removing 0.

*\edef\W {\xintXTrunc {131584}{1/66049}}% a few seconds

*\oodef\W {\expandafter\trim\W}

*\oodef\ZZ {\expandafter\Z\Z}% doubling the period

*\ifx\W\ZZ \message{YES!}\else\message{BUG!}\fi % xint never has bugs...
YES!
*\message{\xintTrunc {260}{1/66049}}% check visually that 256 is not a period

0.00001514027464458205271843631243470756559523989765174340262532362337052794137
6856576178291874214598252812306015231116292449545034746930309315810988811337037
6538630410755651107511090251177156353616254598858423291798513225029902042423049
5541189117170585474420505
*\edef\X {\xintXTrunc {257*128}{1/66049}}% infix here ok, less than 8 tokens

*\oodef\X {\expandafter\trim\X}% we now have the first 257*128 digits

*\oodef\XX {\expandafter\X\X}% was 257*128 a period?

*\ifx\XX\Z \message{257*128 is a period}\else \message{257 * 128 not a period}\fi
257 * 128 not a period
*\immediate\write-1 {1/66049=0.\Z... (repeat)}

*\oodef\ZA {\xintNum {\Z}}% we remove the 0000, or we could use next \xintiMul

*\immediate\write-1 {10\string^65792-1=\xintiiMul {\ZA}{66049}}
```

```

*% This was slow :( I should write a multiplication, still completely
*% expandable, but not f-expandable, which could be much faster on such cases.

*\bye
No pages of output.
Transcript written on worksheet-66049.log.
xxx:_xint $

```

Using `\xintTrunc` rather than `\xintXTrunc` would be hopeless on such long outputs (and even `\xintXTrunc` needed of the order of seconds to complete here). But it is not worth it to use `\xintXTrunc` for less than hundreds of digits.

Fraction arguments to `\xintXTrunc` corresponding to a $A/B[N]$ with a negative N are treated somewhat less efficiently (additional memory impact) than for positive or zero N . This is because the algorithm tries to work with the smallest denominator hence does not extend B with zeroes, and technical reasons lead to the use of some tricks.⁵⁷

Contrarily to `\xintTrunc`, in the case of the second argument revealing itself to be exactly zero, `\xintXTrunc` will output $0.000\dots$, not 0 . Also, the first argument must be at least 1.

10.20 `\xintRound`

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

```

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

```

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

```
-0.00000000000350493899481392497604003313162598556370...
```

10.21 `\xintiRound`

$\frac{\text{num}}{X} \frac{\text{Frac}}{f} \star$ `\xintiRound{x}{f}` returns the integer equal to 10^x times what `\xintRound{x}{f}` would return.

```

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

```

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

10.22 `\xintFloor`, `\xintiFloor`

$\frac{\text{Frac}}{f} \star$ `\xintFloor {f}` returns the largest relative integer N with $N \leq f$.
 $\frac{\text{Frac}}{\text{New } f} \star \rightarrow$ `\xintiFloor {f}` does the same but without adding the $/1[0]$.

```

\xintFloor {-2.13}=-3/1[0], \xintFloor {-2}=-2/1[0], \xintFloor {2.13}=2/1[0]
\xintiFloor {-2.13}=-3, \xintiFloor {-2}=-2, \xintiFloor {2.13}=2

```

⁵⁷ Technical note: I do not provide an `\xintXFloat` because this would almost certainly mean having to clone the entire core division routines into a “long division” variant. But this could have given another approach to the implementation of `\xintXTrunc`, especially for the case of a negative N . Doing these things with \TeX is an effort. Besides an `\xintXFloat` would be interesting only if also for example the square root routine was provided in an X version (I have not given thought to that). If feasible X routines would be interesting in the `\xintexpr` context where things are expanded inside `\csname ..\endcsname`.

10.23 `\xintCeil`, `\xintiCeil`

$\frac{f}{f}$ ★ `\xintCeil {f}` returns the smallest relative integer N with $N > f$.
`\xintCeil {-2.13}=-2/1[0]`, `\xintCeil {-2}=-2/1[0]`, `\xintCeil {2.13}=3/1[0]`
 New $\frac{f}{f}$ ★→ `\xintiCeil {f}` does the same but without adding the `/1[0]`.

10.24 `\xintTFrac`

$\frac{f}{f}$ ★ `\xintTFrac{f}` returns the fractional part, $f=\text{trunc}(f)+\text{frac}(f)$. The *T* stands for ‘Trunc’, and there could similar macros associated to ‘Round’, ‘Floor’, and ‘Ceil’. Inside `\xintexpr.\relax` x , the function `frac` is mapped to `\xintTFrac`. Inside `\xintfloatexpr.\relax`, `frac` first applies `\xintTFrac` to its argument (which may be in float format, or an exact fraction), and only next makes the float conversion.

`\xintTFrac {1235/97}=71/97[0]` `\xintTFrac {-1235/97}=-71/97[0]`
`\xintTFrac {1235.973}=973/1[-3]` `\xintTFrac {-1235.973}=-973/1[-3]`
`\xintTFrac {1.122435727e5}=5727/1[-4]`

10.25 `\xintE`

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

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

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

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

10.26 `\xintFloatE`

num $\frac{x}{x}$ $\frac{f}{f}$ num $\frac{x}{x}$ ★ `\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 `\xintDigits`.

`\xintFloatE {1.23e37}{53}=1.230000000000000e90`

10.27 `\xintDigits`, `\xinttheDigits`

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

★ macro `\xinttheDigits` serves to print the current value.

10.28 `\xintFloat`

num $\frac{x}{x}$ $\frac{f}{f}$ ★ The macro `\xintFloat [P]{f}` has an optional argument P which replaces the current value of `\xintDigits`. The (rounded truncation of the) fraction f is then printed in scientific form, with P digits, a lowercase *e* and an exponent N . The first digit is from 1 to 9, it is preceded by an optional minus sign and is followed by a dot and $P-1$ digits, the trailing zeroes are not trimmed. In the exceptional case where the rounding went to the next power of ten, the output is $10.0\dots 0eN$ (with a sign, perhaps). The sole exception is for a zero value, which then gets output as $0.e0$ (in an `\xintCmp` test it is the only possible output of `\xintFloat` or one of the ‘Float’ macros which will test positive for equality with zero).

`\xintFloat[32]{1234567/7654321}=1.6129020457856418616360615134902e-1`
`\xintFloat[32]{1/\xintFac{100}}=1.0715102881254669231835467595192e-158`

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

10.29 `\xintAdd`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$ Computes the addition of two fractions. To keep for integers the integer format on output use `\xintiAdd`.
Checks if one denominator is a multiple of the other. Else multiplies the denominators.

10.30 `\xintFloatAdd`

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

10.31 `\xintSub`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$ Computes the difference of two fractions (`\xintSub{F}{G}` computes *F-G*). To keep for integers the integer format on output use `\xintiSub`.
Checks if one denominator is a multiple of the other. Else multiplies the denominators.

10.32 `\xintFloatSub`

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

10.33 `\xintMul`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$ Computes the product of two fractions. To keep for integers the integer format on output use `\xintiMul`.
No reduction attempted.

10.34 `\xintSqr`

$\frac{\text{Frac}}{f} \star$ Computes the square of one fraction. To maintain for integer input an integer format on output use `\xintiSqr`.

10.35 `\xintFloatMul`

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

10.36 `\xintDiv`

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$ Computes the algebraic quotient of two fractions. (`\xintDiv{F}{G}` computes *F/G*). To keep for integers the integer format on output use `\xintiMul`.
No reduction attempted.

10.37 `\xintFloatDiv`

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

10.38 `\xintFac`

$\frac{\text{Num}}{f}$ ★ The original is extended to allow a fraction f which will be truncated first to an integer n (non negative and at most 999999, but already 100000! is prohibitively time-costly). On output $n!$ (with no trailing /1[0]).

$\frac{\text{num}}{x}$ ★ The original macro (which parses its input via `\numexpr`) is still available as `\xintiFac`.

10.39 `\xintPow`

$\frac{\text{Frac Num}}{f f}$ ★ `\xintPow{f}{g}`: computes f^g with f a fraction and g possibly also, but g will first get truncated to an integer.

The output will now always be in the form $A/B[n]$ (even when the exponent vanishes: `\xintPow {2/2 3}{0}=1/1[0]`).

The original is available as `\xintiPow`.

The exponent (after truncation to an integer) will be checked to not exceed 100000. Indeed 2^{50000} already has 15052 digits, and squaring such a number would take hours (I think) with the expandable routine of *xint*.

10.40 `\xintFloatPow`

$\frac{\text{num}}{x}$ $\frac{\text{Frac Num}}{f x}$ ★ `\xintFloatPow [P]{f}{x}` uses either the optional argument P or the value of `\xintDigits`. It computes a floating approximation to f^x . The precision P must be at least 1, naturally.

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

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

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

10.41 `\xintFloatPower`

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

```
\xintFloatPower [8]{1.00000000000001}{1e12}=2.7182818e0
```

```
\xintFloatPower [8]{3.1415}{3e9}=1.4317729e1491411192
```

Note that $3e9 > 2^{31}$. But the number following e in the output must at any rate obey the $\text{T}_{\text{E}}\text{X}$ 2147483647 bound.

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

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

10.42 `\xintFloatSqrt`

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

```
\xintFloatSqrt [50]{12.3456789e12}
```

```
≈ 3.5136418286444621616658231167580770371591427181243e6
```

```
\xintDigits:=50;\xintFloatSqrt {\xintFloatSqrt {2}}
```

```
≈ 1.1892071150027210667174999705604759152929720924638e0
```

10.43 `\xintSum`

$f \rightarrow * \frac{f}{f}$ ★ 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}}
-15113784906302076/18350036010217404[0]
No simplification attempted.
```

10.44 `\xintPrd`

$f \rightarrow * \frac{f}{f}$ ★ This computes the product of fractions. The output will now always be in the form $A/B[n]$. The original, for big integers only (in strict format), is available as `\xintiiPrd`.

```
\xintPrd {{1282/2196921}}{-281710/291927}{4028/28612}}
-1454721142160/18350036010217404[0]
No simplification attempted.
```

10.45 `\xintCmp`

$\frac{f}{f} \frac{f}{f}$ ★ 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 , the following syntax is recommended: `\xintSgnFork{\xintCmp{f}{g}}{code for f<g}{code for f=g}{code for f>g}`.

10.46 `\xintIsOne`

$\frac{f}{f}$ ★ This returns 1 if the fraction is 1 and 0 if not.

```
\xintIsOne {21921379213/21921379213} but \xintIsOne {1.00000000000000000000000000000001}
1 but 0
```

10.47 `\xintGeq`

$\frac{f}{f} \frac{f}{f}$ ★ This compares the *absolute values* of two fractions. `\xintGeq{f}{g}` returns 1 if $|f| \geq |g|$ and 0 if not. May be used for expandably branching as: `\xintSgnFork{\xintGeq{f}{g}}{code for |f|<|g|}{code for |f|>=|g|}`

10.48 `\xintMax`

$\frac{f}{f} \frac{f}{f}$ ★ The maximum of two fractions. But now `\xintMax {2}{3}` returns $3/1[0]$. The original, for use with (possibly big) integers only with no need of normalization, is available as `\xintiiMax`: `\xintiiMax {2}{3}=3`.

$\frac{ff}{f} \frac{ff}{f}$ ★ There is also `\xintiMax` which works with fractions but first truncates them to integers.

```
\xintMax {2.5}{7.2} but \xintiMax {2.5}{7.2}
72/1[-1] but 7
```

10.49 `\xintMin`

$\frac{f}{f} \frac{f}{f}$ ★ The minimum of two fractions. The original, for use with (possibly big) integers only with no need of normalization, is available as `\xintiiMin`: `\xintiiMin {2}{3}=2`.

$\frac{ff}{f} \frac{ff}{f}$ ★ There is also `\xintiMin` which works with fractions but first truncates them to integers.

```
\xintMin {2.5}{7.2} but \xintiMin {2.5}{7.2}
25/1[-1] but 2
```

10.51 \xintMinof

10.52 \xintAbs

10.53 \xintSgn

10.54 \xint0pp

10.55 \xintiDivision, \xintiQuo, \xintiRem, \xintFDg, \xintLDg, \xintMON,
\xintMMON, \xintOdd

[illegible]

11 Commands of the `xintexpr` package

.1	The <code>\xintexpr</code> expressions	87	.11	<code>\xintfloatexpr</code> , <code>\xintthefloatexpr</code>	104
.2	The syntax.....	88	.12	<code>\xintifboolexpr</code>	105
.3	<code>\numexpr</code> or <code>\dimexpr</code> expressions, count and dimension registers and variables	98	.13	<code>\xintifboolfloatexpr</code>	105
.4	Catcodes and spaces.....	98	.14	<code>\xintifbooliiexpr</code>	105
.5	Expandability, <code>\xinteval</code>	99	.15	<code>\xintNewFloatExpr</code>	105
.6	Memory considerations	99	.16	<code>\xintNewIExpr</code>	106
.7	The <code>\xintNewExpr</code> command	99	.17	<code>\xintNewIIExpr</code>	106
.8	<code>\xintiexpr</code> , <code>\xinttheiexpr</code>	103	.18	<code>\xintNewBoolExpr</code>	106
.9	<code>\xintiiexpr</code> , <code>\xinttheiiexpr</code>	103	.19	<code>\xintthecoords</code>	106
.10	<code>\xintboolexpr</code> , <code>\xinttheboolexpr</code>	104	.20	Technicalities	106
			.21	Acknowledgements	107

86

- for using the `gcd` and `lcm` functions, it is necessary to load package `xintgcd`.

```
\xinttheexpr lcm (2^5*7*13^10*17^5,2^3*13^15*19^3,7^3*13*23^2)\relax
2894379441338000036761046087608864
```

- for allowing hexadecimal numbers (uppercase letters) on input, it is necessary to load package `xintbinhex`.

```
\xinttheexpr "A*"B*"C*"D*"D*"F, "FF.FF, reduce("FF.FFF + 16^-3)\relax
3346200, 25599609375[-8], 256
```

Release 1.1 has brought many changes to `xintexpr`. See [section 6](#) if you are already familiar with the earlier versions.

11.1 The `\xintexpr` expressions

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

During this parsing, braced sub-content may be serving as usual as a macro parameter, or as a branch to the `?` two-way and `??` three-way operators. Prior to release 1.1, there were also some other usage, but this has been removed. It was mainly needed because there was no other way to feed the number parser directly with fractions in the `A/B[N]` notation which is the output format of the `xintfrac` macros. There was no real need to use such macros anyhow. If one really wants to, one can now directly:

```
\xinttheexpr \xintAdd{3/5[2]}{7/13[2]}+199/13[1]\relax
1735/65[1]
```

Notice in passing that the expressions benefit from the improved handling of denominators by `\xintAdd` and `\xintSub` from `xintfrac`, which are the macros to which naturally `+` and `-` are mapped.

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

- x ★** • `\xinttheexpr<expandable_expression>\relax`, or
- x ★** • `\xintthe\xintexpr<expandable_expression>\relax`.

The computations are done *exactly*, and with no simplification of the result. The result can be modified via the functions `round`, `trunc`, `float`, or `reduce`.⁵⁸ Here are some examples

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

With 1.1 one has:

```
\xinttheiexpr [10] 1.99^-2 - 2.01^-2\relax=0.0050002500
```

- the expression may contain arbitrarily many levels of nested parenthesized sub-expressions.
- to let sub-contents evaluate as a sub-unit it should be either
 1. parenthesized,
 2. or a sub-expression `\xintexpr... \relax`.

When the parser scans a number and hits against either an opening parenthesis or a sub-expression it inserts tacitly a `*`.

⁵⁸ In `round` and `trunc` the second optional parameter is the number of digits of the fractional part; in `float` it is the total number of digits of the mantissa.

- to either give an expression as argument to the other package macros, or more generally to macros which expand their arguments, one must use the `\xinttheexpr...\relax` or `\xintthe\xintexpr...\relax` forms. Similarly, printing the result itself must be done with these forms.
- one should not use `\xinttheexpr...\relax` as a sub-constituent of an `\xintexpr...\relax` but rather the `\xintexpr...\relax` form which will be more efficient.
- each `xintexpression`, whether prefixed or not with `\xintthe`, is completely expandable and obtains its result in two expansion steps.

In an algorithm implemented non-expandably, one may define macros to expand to infix expressions to be used within others. One then has the choice between parentheses or `\xintexpr...\relax`: `\def\x{(\a+\b)}` or `\def\x{\xintexpr \a+\b\relax}`. The latter is the better choice as it allows also to be prefixed with `\xintthe`. Furthermore, if `\a` and `\b` are already defined `\edef\x{\xintexpr \a+\b\relax}` will do the computation on the spot. Rather than `\edef` one can use `\oodef`.

11.2 The syntax

An expression is enclosed between either `\xintexpr`, or `\xintiexpr`, or `\xintiexpr`, or `\xintfloatexpr`, or `\xintboolexpr` and a mandatory ending `\relax`. An expression may be a sub-unit of another one.

Apart from `\xintfloatexpr` the evaluations of algebraic operations are exact. The variant `\xintiexpr` does not know fractions and is provided for integer-only calculations. The variant `\xintiexpr` is exactly like `\xintexpr` except that it either rounds the final result to an integer, or in case of an optional parameter `[d]`, rounds to a fixed point number with `d` digits after decimal mark. The variant `\xintfloatexpr` does float calculations according to the current value of the precision set by `\xintDigits`.

The whole expression should be prefixed by `\xintthe` when it is destined to be printed on the typeset page, or given as argument to a macro (assuming this macro systematically expands its argument). As a shortcut to `\xintthe\xintexpr` there is `\xinttheexpr`. One gets used to not forget the two `t`'s.

`\xintexpr`-essions and `\xinttheexpr`-essions are completely expandable, in two steps.

- An expression is built the standard way with opening and closing parentheses, infix operators, and (big) numbers, with possibly a fractional part, and/or scientific notation (except for `\xintiexpr` which only admits big integers). All variants work with comma separated expressions. On output each comma will be followed by a space.
- 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 pdfTeX, activates `!`, `?`, `;` and `:`. Turn off the activity before the expressions.

Alternatively the command `\xintexprSafeCatcodes` resets all characters potentially needed by `\xintexpr` to their standard catcodes and `\xintexprRestoreCatcodes` restores the status prevailing at the time of the previous `\xintexprSafeCatcodes`.

- The infix operators are `+`, `-`, `*`, `/`, `^` (or `**`) for exact or floating point algebra (only integer exponents for power operations), `&&` and `||`⁵⁹ for combining “true” (non zero) and “false” (zero) conditions, as can be formed for example with the `=` (or `==`), `<`, `>`, `<=`, `>=`, `!=` comparison operators.
- The `!` is either a function (the logical not) requiring an argument within parentheses, or a post-fix operator which does the factorial (so far, no float version).

⁵⁹ with releases earlier than 1.1, only single character operators `&` and `|` were available, because the parser did not handle multi-character operators. Their usage in this rôle is now deprecated, as they may be assigned some new meaning in the future.

- The `?` may serve either as a function (the truth value) requiring an argument within parentheses), or as two-way post-fix branching operator `(cond)?{YES}{NO}`. The false branch will *not* be evaluated.
- There is also `??` which branches according to the scheme `(x)??{<0}{=0}{>0}`.
- Comma separated lists may be generated with `a..b` and `a..[d]..b` and they may be manipulated to some extent once put into brackets:

- `a..b` constructs the **small** integers from `[a]` to `[b]` (possibly a decreasing sequence),

```
\xinttheexpr 1.5..11.23\relax
```

2, 3, 4, 5, 6, 7, 8, 9, 10, 11

- `a..[d]..b` allows big integers, or fractions, it proceeds by step of `d`.

```
\xinttheexpr 1.5..[0.97]..11.23\relax
```

15[-1], 247[-2], 344[-2], 441[-2], 538[-2], 635[-2], 732[-2], 829[-2], 926[-2], 1023[-2], 1120[-2]

- `[list][n]` extracts the `n`th element, or give the number of items if `n=0`. If `n<0` it extracts from the tail.

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

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

See the frame coming next for the `\empty` token. As shown, it is perfectly legal to do operations in the index parameter, which will be handled by the parser as everything else. The same remark applies to the next items.

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

```
\xinttheiexpr [1..10][:6]\relax\ and \xinttheiexpr [1..10][: -6]\relax
```

1, 2, 3, 4, 5, 6 and 1, 2, 3, 4

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

```
\xinttheiexpr [1..10][6:]\relax\ and \xinttheiexpr [1..10][-6:]\relax
```

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

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

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

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

- It is naturally possible to nest these things:

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

24, 25, 26, 27

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

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

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



As list operations are implemented using square brackets, it is necessary in `\xintexpr` and `\xintfloatexpr` to insert something before the first bracket if it belongs to a list, to avoid confusion with the bracket of an optional parameter. We need something expandable which does not leave a trace: the `\empty` does the trick.

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

6, 99

An alternative is to use parentheses

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

6, 99

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

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

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

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

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

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

In an other vein, the parser will be confused by `1..[list][3]`, one must write `1..([list][3])`. Also things such as `[100,300,500,700][2]/11` will be confusing because the `]/` is an operator with higher priority than the `]`, and then there will a dangling `/11` which does not make sense. In fact even `[100,300,500,700][2]/11` is a syntax error: one must write `([100,300,500,700][2])/11`.

- 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 (TeX'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 register, variable, or expression is immediately prefixed by a (decimal) number.
- tacit multiplication (the parser inserts a `*`) applies when the parser is currently scanning the digits of a number (or its decimal part or scientific part), or is looking for an infix operator, and: (1.) encounters a register, variable or ϵ -TeX expression (as described in the previous item), (2.) encounters a sub-`\xintexpr`-ession, or (3.) encounters an opening parenthesis.
- when defining a macro to expand to an expression either via


```
\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 but keeps it in an internal private format. Naturally, the `\edef` is only possible if `\a` and `\b` are already defined, either as macros expanding to legal syntax like `37^23*17` or themselves in the same way `\x` above was defined. Indeed in both cases (the 'yet-to-be computed' and the 'already computed') `\x` can then be inserted in other expressions, as for example


```
\edef\y {\xintexpr \x^3\relax}
```

This would have worked also with `\x` defined as `\def\x {(\a+\b)}` but `\edef\x` would not have been an option then, and `\x` could have been used only inside an `\xintexpr`-ession, whereas the previous `\x` can also be used as `\xintthe\x` in any context triggering the expansion of `\xintthe`.

- there is also `\xintboolexpr ... \relax` and `\xinttheboolexpr ... \relax`. Same as `\xintexpr` with the final result converted to 1 if it is not zero. See also `\xintifboolexpr` (subsection 11.12) and the discussion of the `bool` and `togl` functions in section 11. Here is an example:

```
\xintNewBoolExpr \AssertionA[3]{ #1 & (#2|#3) }
\xintNewBoolExpr \AssertionB[3]{ #1 | (#2&#3) }
\xintNewBoolExpr \AssertionC[3]{ xor(#1,#2,#3) }
{\centering\normalcolor\xintFor #1 in {0,1} \do {%
  \xintFor #2 in {0,1} \do {%
    \xintFor #3 in {0,1} \do {%
      #1 AND (#2 OR #3) is \textcolor[named]{OrangeRed}{\AssertionA {#1}{#2}{#3}}\hfil
      #1 OR (#2 AND #3) is \textcolor[named]{OrangeRed}{\AssertionB {#1}{#2}{#3}}\hfil
      #1 XOR #2 XOR #3 is \textcolor[named]{OrangeRed}{\AssertionC {#1}{#2}{#3}}\}}}

```

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

This example used for efficiency `\xintNewBoolExpr`. See also the subsection 11.7.

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

```
\xintthefloatexpr 2^1000000\relax: 9.990020930143845e30102
```

The square-root operation can be used in `\xintexpr`, it is computed as a float with the precision set by `\xintDigits` or by the optional second argument:

```
\xinttheexpr sqrt(2,60)\relax
```

```
Here the [60] is to avoid truncation to |\xintDigits| of precision on output.
\printnumber{\xintthefloatexpr [60] sqrt(2,60)\relax}
```

```
141421356237309504880168872420969807856967187537694807317668[-59]
```

Here the [60] is to avoid truncation to `\xintDigits` of precision on output. 1.41421356237309504880168872420969807856967187537694807317668

- Floats are quickly indispensable when using the power function (which can only have an integer exponent), as exact results will easily have hundreds, if not thousands, of digits.

```
\xintDigits:=48;\xintthefloatexpr 2^1000000\relax
```

```
9.99002093014384507944032764330033590980429139054e30102
```

- hexadecimal \TeX number denotations (i.e., with a " prefix) are recognized by the `\xintexpr` parser and its variants. This requires `xintbinhex`. Except in `\xintiexpr`, a (possibly empty) fractional part with the dot . as "hexadecimal" mark is allowed.

```
\xinttheexpr "FEDCBA9876543210\relax→18364758544493064720
```

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

Letters must be uppercased, as with standard \TeX hexadecimal denotations.

Note that 2^4-10 is perfectly accepted input, no need for parentheses; operators of power $^$, division $/$, and subtraction $-$ are all left-associative: 2^4^8 is evaluated as $(2^4)^8$. The minus sign as prefix has various precedence levels: `\xintexpr -3-4*-5^7\relax` evaluates as $(-3)-(4*(-5^7))$ and -3^4*-5-7 as $(-((3^4)*(-5)))-7$.

If one uses directly macros within `\xintexpr...\relax`, rather than the operators or the functions which are described next, one should obviously take into account that the parser will *not* see the macro arguments.

Here is, listed from the highest priority to the lowest, the complete list of operators and functions.

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

`num`, `reduce`, `abs`, `sgn`, `frac`, `floor`, `ceil`, `sqr`, `sqrt`, `sqrtr`, `float`, `round`, `trunc`, `mod`, `quo`, `rem`, `gcd`, `lcm`, `max`, `min`, `'+'`, `'*'`, `?`, `!`, `not`, `all`, `any`, `xor`, `if`, `ifsgn`, `even`, `odd`, `first`, `last`, `reversed`, `bool`, `togl`, `add`, `mul`, `seq`, `subs`, `rseq`, `rrseq`, `iter`

`quo`, `rem`, `even`, `odd`, `gcd` and `lcm` will first truncate their arguments to integers; the latter two require package `xintgcd`; `togl` requires the `etoolbox` package; `all`, `any`, `xor`, `'+'`, `'*'`, `max` and `min` are functions with arbitrarily many comma separated arguments.

`bool` and `togl` use delimited macros to fetch their argument whose closing parenthesis thus must be explicit, not arising from expansion.

Similarly `add`, `mul`, `subs`, `seq`, `rseq`, `rrseq`, `iter` use at some stages delimited macros. They work with *dummy variables*, represented as one Latin letter (lowercase or uppercase) followed by a mandatory `=` sign, then a comma separated list of values to assign in turn to the dummy variable, which will be substituted in the expression which was parsed as the first, comma delimited, argument to the function; additionally `rseq`, `rrseq` and `iter` have a mandatory initial comma separated list which is separated by a semi-colon from the expression to evaluate iteratively. `seq`, `rseq`, `rrseq`, `iter` but not `add`, `mul`, `subs` admit the `omit`, `abort`, and `break(..)` keywords, possibly but not necessarily in combination with a potentially infinite list generated by a `n++` expression.

They may be nested.

functions with a single (numeric) argument

num truncates to the nearest integer (truncation towards zero)

```
\xinttheexpr num(3.1415^20)\relax
```

8764785276

reduce reduces a fraction to smallest terms

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

141599788807961859400

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

abs absolute value

sgn sign

frac fractional part

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

-16/113, -218921791279[-12]

floor floor function

ceil ceil function

sqr square

sqr in `\xintiexpr`, truncated square root; in `\xintexpr` or `\xintfloatexpr` this is the floating point square root, and there is an optional second argument for the precision.

sqrtr in `\xintiexpr` only, rounded square root.

? ?(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 logical not

even evenness of the truncation

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

```
-5/2, 1, -13/6, 1, -11/6, 0, -9/6, 0, -7/6, 0, -5/6, 1, -3/6, 1, -1/6, 1, 1/6, 1, 3/6, 1,
5/6, 1, 7/6, 0, 9/6, 0, 11/6, 0, 13/6, 1, 15/6, 1
```

odd oddness of the truncation

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

```
-5/2, 0, -13/6, 0, -11/6, 1, -9/6, 1, -7/6, 1, -5/6, 0, -3/6, 0, -1/6, 0, 1/6, 0, 3/6, 0,
5/6, 0, 7/6, 1, 9/6, 1, 11/6, 1, 13/6, 0, 15/6, 0
```

functions with an alphabetical argument

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

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

will return NO if executed in math mode (the computation is then $100 - 100 = 0$) and YES if not (the `if` conditional is described below; the `\xintifboolexpr` test automatically encapsulates its first argument in an `\xintexpr` and follows the first branch if the result is non-zero (see subsection 11.12)).

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

Similarly **togl(name)** returns 1 if the \LaTeX package `etoolbox`⁶⁰ has been used to define a toggle named `name`, and this toggle is currently set to `true`. Using **togl** in an `\xintexpr` `..\relax` without having loaded `etoolbox` will result in an error from `\iftoggle` being a non-defined macro. If `etoolbox` is loaded but **togl** is used on a name not recognized by `etoolbox` the error message will be of the type “ERROR: Missing `\endcsname` inserted.”, with further information saying that `\protect` should have not been encountered (this `\protect` comes from the expansion of the non-expandable `etoolbox` error message).

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

There isn't in `\xintexpr`... a `test` function available analogous to the `test{\ifsometes}t` construct from the `etoolbox` package; but any expandable `\ifsometest` can be inserted

⁶⁰ <http://www.ctan.org/pkg/etoolbox>

directly in an *\xintexpr*-ession as *\ifsometest10* (or *\ifsometest{1}{0}*), for example *if0 \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 *\ifsoqmetest10* is to allow arbitrary boolean combinations using the (described later) *&* and *|* logic operators: *\ifsometest10 & \ifsomeothertest10 | \ifsomehirdtest10*, etc... *YES* or *NO* above stand for material compatible with the *\xintexpr* parser syntax.

See also *\xintifboolexpr*, in this context.

functions with one mandatory and a second but optional argument

round For example *round(-2^9/3^5,12)=-2.106995884774*.

trunc For example *trunc(-2^9/3^5,12)=-2.106995884773*.

float For example *float(-20^9/3^5,12)=-210699588477[-2]*.

sqr in *\xintexpr* and *\xintfloatexpr*, uses the float evaluation with the precision given by the optional second argument.

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

1414213562373095048801688724210[-30] and 1414213562373095048801688724209

functions with two arguments

quo first truncates the arguments then computes the Euclidean quotient.

rem first truncates the arguments then computes the Euclidean remainder.

mod computes the modulo associated to the truncated division, same as */:* infix operator

```
\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
```

3/91, 11/7, 0, 20

the if conditional (twofold way)

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

the ifsgn conditional (threefold way)

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

functions with an arbitrary number of arguments

This argument may well be generated by one or many *a..b* or *a..[d]..b* constructs, separated by commas.

all inserts a logical *AND* in between arguments and evaluates,

any inserts a logical *OR* in between all arguments and evaluates,

xor inserts a logical *XOR* in between all arguments and evaluates,

`+` adds (left ticks mandatory),

``* multiplies (left ticks mandatory),

max maximum,

min minimum,

gcd first truncates to integers then computes the *GCD*, requires *xintgcd*,

lcm first truncates to integers then computes the *LCM*, requires *xintgcd*,

first first among comma separated items, *first(list)* is like *[list][:1]*.

```
\xinttheiexpr first(-7..3), [-7..3][:1]\relax
```

$-7, -7$

last last among comma separated items, `last(list)` is like `[list][-1:]`.

```
\xinttheiiexpr last(-7..3), [-7..3][-1:]\relax
```

3, 3

reversed reverses the order

```
\xinttheiiexpr reversed(123..150)\relax
```

150, 149, 148, 147, 146, 145, 144, 143, 142, 141, 140, 139, 138, 137, 136, 135, 134, 133, 132, 131, 130, 129, 128, 127, 126, 125, 124, 123

functions using dummy variables

They are nestable to arbitrary depth if suitably parenthesized.

subs for variable substitution, useful to get something evaluated only once

```
% ATTENTION that xz generates an error, 'unknown variable xz', one must use x*z
```

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

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

add addition

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

1625625

mul multiplication

```
\xinttheiiexpr subs(mul(2n+1,n=1..N),N=30)\relax
```

1782151988659863326386101665566204817109375

seq comma separated values generated according to a formula

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

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

```
\xinttheiiexpr 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 recursive sequence, @ for the previous value.

```
% ATTENTION y/2@ would give (y/2)@, that is (y/2)*@ !!
```

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

1.0000000000000000, 500.50000000000000, 251.24900099990010, 127.6145581634591, 67.72532732
6082604, 41.24542607499115, 32.74526934448864, 31.64201586865079, 31.62278245070105, 31.62277660168434, 31.62277660168379

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’.

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

1.0000000000000000, 1000000.0000000000, 1000.000000000000, 500000.5000000000, 22360.69092
5533499, 250500.2500000000, 74842.22521066670, 136430.4704776675, 101048.3052657827, 102
5636.3478441671, 103316.8617608946, 103342.3265549749, 103329.5933734841, 103329.594152
79348, 103329.5937657094, 103329.5937657095, 103329.5937657094, 103329.5937657095, 10332
9.5937657094, 103329.5937657095, 103329.5937657094, 103329.5937657095

rrseq recursive sequence with multiple initial terms. Say, there are *K* of them. Then @1, ..., @4 and then @@(n) up to n=K refer to the last *K* values. Notice the difference with *rseq* for which @ refers to the complete list of all initial terms (if there are more than one).

```
\xinttheiexpr 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
```

```
\xinttheiexpr 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.

iter same as *rrseq* but does not print any value until the last *K*.

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

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

```
34, 55
```

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

With *seq*, *rseq*, *rrseq*, *iter*, but not with *subs*, *add*, *mul*, one has:

abort stop here and now.

omit omit this value.

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

n++ serves to generate a potentially infinite list

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

```
% this would not work with i=1,2,3++. Only n++ syntax, nothing before.
```

```
10889035741470030830827987437816582766592
```

Refer to [section 6](#) for more examples.

- The postfix operators **!** and the branching conditionals **?**, **??**.

! computes the factorial of an integer. This is the exact factorial even when used inside *\xintfloatexpr*.

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

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

is legal and computes *5+62^3=238333*. Note though that it would be better practice to include here the *2^3* inside the branches. The contents of the branches may be arbitrary as long as once glued to what is next the syntax is respected: *\xintexpr (3>2)?{5+(6){7-(1}2^3}\relax* also works. Differs thus from the *if* conditional in two ways: the false branch is not at all computed, and the number scanner is still active on exit, more digits may follow.

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

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

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

```
\xinttheexpr .^2+2^.\relax
which is  $0^2+2^0$  and produces 1.
```

- Changed →
- The `e` and `E` for scientific notation. They are parsed like the decimal mark is. Thus `1e(3+2)` is no legal syntax anymore, one must use `10^(3+2)` in such cases. `1e3^2` is `1[6]`
 - The `"` for hexadecimal numbers: it is treated with highest priority, allowed only at locations where the parser expects to start forming a numeric operand, once encountered it triggers the hexadecimal scanner which looks for successive hexadecimal digits (as usual skipping spaces and expanding forward everything) possibly a unique optional dot (allowed directly in front) and then an optional (possibly empty) fractional part. The dot and fractional part are not allowed in `\xintiexpr.\relax`. The `"` functionality `[requires package xintbinhex]` (there is no warning, but an ‘undefined control sequence’ error will naturally results if the package has not been loaded). `"A*"A^"A` is `10000000000`.
 - The power operator `^`, or `**`. It is left associative: `\xinttheiexpr 2^2^3\relax` evaluates to 64, not 256. Note that if the float precision is too low, iterated powers withing `\xintfloatexpr.\relax` may fail: for example with the default setting `(1+1e-8)^(12^16)` will be computed with `12^16` approximated from its 16 most significant digits but it has 18 digits (`=184884258895036416`), hence the result is wrong:

$$1.879,985,375,897,266 \times 10^{802,942,130}$$
 One should code


```
\xintthe\xintfloatexpr (1+1e-8)^\xintiexpr 12^16\relax \relax
```

 to obtain the correct floating point evaluation

$$1.000,000,01^{12^{16}} \approx 1.879,985,676,694,948 \times 10^{802,942,130}$$
 - Multiplication and division `*`, `/`. The division is left associative, too: `\xinttheiexpr 100/50/2\relax` evaluates to 1, not 4. Inside `\xintiexpr`, `/` does rounded division.
 - Truncated division `//` and modulo `/:` (equivalently `'mod'`, quotes mandatory) are at the same level of priority than multiplication and division, thus left-associative with them. Apply parentheses for disambiguation.


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

`7692, 4, 4, 7692.3076923076, 0.3076923076`
 - The list itemwise operators `*[`, `/[`, `^[`, `**[`, `]*`, `]/`, `]^`, `]**` are at the same precedence level as, respectively, `*` and `/` or `^` and `**`.
 - Addition and subtraction `+`, `-`. Again, `-` is left associative: `\xinttheiexpr 100-50-2\relax` evaluates to 48, not 52.
 - The list itemwise operators `+[`, `-[`, `]+`, `]-`, are at the same precedence level as `+` and `-`,
 - Comparison operators `<`, `>`, `=` (same as `==`), `<=`, `>=`, `!=` all at the same level of precedence, use parentheses for disambiguation.
 - Conjunction (logical and): `&&` or equivalently `'and'` (quotes mandatory).
 - Inclusive disjunction (logical or): `||` and equivalently `'or'` (quotes mandatory).
 - XOR: `'xor'` with mandatory quotes is at the same level of precedence as `||`.

- The comma: With `\xinttheexpr 2^3,3^4,5^6\relax` one obtains as output 8, 81, 15625.
- The parentheses.

See [subsection 11.3](#) for count and dimen registers and variables.

11.3 `\numexpr` or `\dimexpr` expressions, count and dimension registers and variables

Count registers, count control sequences, dimen registers, dimen control sequences, skips and skip control sequences, `\numexpr`, `\dimexpr`, `\glueexpr` can be inserted directly, they will be unpacked using `\number` (which gives the internal value in terms of scaled points for the dimensional variables: `1pt = 65536sp`; stretch and shrink components are thus discarded). Tacit multiplication is implied, when a number or decimal number prefixes such a register or control sequence.

\TeX lengths are skip control sequences and \TeX counters should be inserted using `\value`.

In the case of numbered registers like `\count255` or `\dimen0`, the resulting digits will be re-parsed, so for example `\count255 0` is like `100` if `\the\count255` would give `10`. Control sequences define complete numbers, thus cannot be extended that way with more digits, on the other hand they are more efficient as they avoid the re-parsing of their unpacked contents.

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 [subsection 4.4](#).

11.4 Catcodes and spaces

Active characters may (and will) break the functioning of `\xintexpr`. Inside an expression one may prefix, for example a `:` with `\string`. Or, for a more radical way, there is `\xintexprSafeCatcodes`. This is a non-expandable step as it changes catcodes.

11.4.1 `\xintexprSafeCatcodes`

This command sets the catcodes of the relevant characters to safe values. This is used internally by `\xintNewExpr` (restoring the catcodes on exit), hence `\xintNewExpr` does not have to be protected against active characters.

11.4.2 `\xintexprRestoreCatcodes`

Restores the catcodes to the earlier state.

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.⁶¹

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 `!` as either logical negation or postfix factorial operator must be a standard (*i.e.* catcode 12) `!`, more precisely a catcode 11 exclamation point `!` must be avoided as it is used internally by `\xintexpr` for various special purposes.

⁶¹ Furthermore, although `\xintexpr` uses `\string`, it is (we hope) escape-char agnostic.

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.

11.5 Expandability, `\xinteval`

As is the case with all other package macros `\xintexpr` *f*-expands (in two steps) to its final (non-printable) result; and `\xinttheexpr` *f*-expands (in two steps) to the chain of digits (and possibly minus sign `-`, decimal mark `.`, fraction slash `/`, scientific `e`, square brackets `[,]`) representing the result.

Starting with 1.09j, an `\xintexpr...relax` can be inserted without `\xintthe` prefix inside an `\edef`, `f`, or a `\write`. It expands to a private more compact representation (five tokens) than `\xinttheexpr` or `\xintthe\xintexpr`.

The material between `\xintexpr` and `\relax` should contain only expandable material.

The once expanded `\xintexpr` is `\romannumeral0\xinteval`. And there is similarly `\xintieval`, `\xintfloateval`, and `\xintfloateval`. For the other cases one can use `\romannumeral-`0` as prefix. For an example of expandable algorithms making use of chains of `\xinteval`-uations connected via `\expand` after see subsection 7.24.

An expression can only be legally finished by a `\relax` token, which will be absorbed.

It is quite possible to nest expressions among themselves; for example, if one needs inside an `\xintiexpr...relax` to do some computations with fractions, rounding the final result to an integer, one just has to insert `\xintiexpr...relax`. The functioning of the infix operators will not be in the least affected from the fact that the surrounding ‘‘environment’’ is the `\xintiexpr` one.

11.6 Memory considerations

The parser creates an undefined control sequence for each intermediate computation evaluation: addition, subtraction, etc... Thus, a moderately sized expression might create 10, or 20 such control sequences. On my \TeX installation, the memory available for such things is of circa 200,000 multi-letter control words. So this means that a document containing hundreds, perhaps even thousands of expressions will compile with no problem.

Besides the hash table, also \TeX main memory is impacted. Thus, if `xintexpr` is used for computing plots⁶², this may cause a problem.

There is a (partial) solution.⁶³

A document can possibly do tens of thousands of evaluations only if some formulas are being used repeatedly, for example inside loops, with counters being incremented, or with data being fetched from a file. So it is the same formula used again and again with varying numbers inside.

With the `\xintNewExpr` command, it is possible to convert once and for all an expression containing parameters into an expandable macro with parameters. Only this initial definition of this macro actually activates the `\xintexpr` parser and will (very moderately) impact the hash-table: once this unique parsing is done, a macro with parameters is produced which is built-up recursively from the `\xintAdd`, `\xintMul`, etc... macros, exactly as it would be necessary to do without the facilities of the `xintexpr` package.

11.7 The `\xintNewExpr` command

The command is used as:

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

⁶² this is not very probable as so far `xint` does not include a mathematical library with floating point calculations, but provides only the basic operations of algebra. ⁶³ which convinced me that I could stick with the parser implementation despite its potential impact on the hash-table and other parts of \TeX 's memory.

- `<stuff>` will be inserted inside `\xinttheexpr . . . \relax`,
- `n` is an integer between zero and nine, inclusive, and tells how many parameters will `\myformula` have (it is mandatory even if `n=0`⁶⁴)
- the placeholders `#1`, `#2`, ..., `#n` are used inside `<stuff>` in their usual rôle.

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

The definition of `\myformula` made by `\xintNewExpr` is global (i.e. it does not obey the scope of environments). The protection against active characters is done automatically.

It will be a completely expandable macro entirely built-up using `\xintAdd`, `\xintSub`, `\xintMul`, `\xintDiv`, `\xintPow`, etc. . . as corresponds to the expression written with the infix operators. 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. 10000000000.00000
2. 17.32050807568877
3. 3.891379490446502e16

The whole point of using `\xintNewExpr` is to produce a macro which does not do any of the `\xintexpr` for expandable manipulations, manipulations which may have an impact on some regions of \TeX 's memory (this impact gets noticeable only if many of thousands of `\xintexpr`-essions are evaluated).

A ‘‘formula’’ created by `\xintNewExpr` is thus a macro whose parameters are given to a possibly very complicated combination of the various macros of `xint` and `xintfrac`. Consequently, one can not use at all any infix notation there, but only the input format recognized by the `xintfrac` macros.

The situation is thus quite distinct from a macro with parameters defined via a simple `\def` such as:

```
\def\myformula #1{\xinttheexpr (#1)^3\relax}
```

`\xintNewExpr` tries to do as many evaluations as are possible at the time the macro parameters are still parameters. Let's see a few examples. For this I will use `\meaning` which reveals the contents of a macro. We will thus see some private macros of the `xint` bundle, which should not be directly used. If the things look a bit complicated, it is because they have to cater for many possibilities. With `\meaning` we will see what has already been evaluated.

Remark: In these examples we sometimes use `\printnumber` to avoid for the meaning to go into the right margin, but this zaps all spaces originally in the output from `\meaning`.

```
\xintNewIIExpr\FA [1]{13*25*78*#1+2826*292}\meaning\FA
```

```
macro:#1->\romannumeral -'0\xintCSV::csv {\xintiiAdd {\xintiiMul {25350}{#1}}{825192}}
```

```
\xintNewIExpr\FA [2]{(3/5*9/7*13/11*#1-#2)*3^7}
```

```
\printnumber{\meaning\FA}
```

```
macro:#1#2->\romannumeral -'0\xintSPRaw::csv{\xintRound::csv{\xintMul{\xintSub{\xintMul{3521/385[0]}{#1}}{#2}}{2187/1[0]}}
```

```
% an example with optional parameter
```

⁶⁴ there is some use for `\xintNewExpr[0]` compared to an `\edef` as `\xintNewExpr` has some built-in catcode protection.

```
\xintNewExpr\FA [3]{[24] (#1+#2)/(#1-#2)^#3}
\printnumber{\meaning\FA}
macro:#1#2#3->\romannumeral-'0\xintSPRaw::csv{\xintRound::csv{24}{\xintDiv{\xintAdd{#1}{#2}}{\xintPow{\xintSub{#1}{#2}}{#3}}}}
```

```
\xintNewFloatExpr\FA [2]{[12] 3.1415^3*#1-#2^5}
\printnumber{\meaning\FA}
macro:#1#2->\romannumeral-'0\xintPFloat::csv{12}{\XINTinFloatSub{\XINTinFloatMul{31003533398237500[-14]}{#1}}{\XINTinFloatPower{#2}{5}}}
```

```
\xintNewExpr\DET[9]{ #1*#5*#9+#2*#6*#7+#3*#4*#8-#1*#6*#8-#2*#4*#9-#3*#5*#7 }
\printnumber{\meaning\DET}
macro:#1#2#3#4#5#6#7#8#9->\romannumeral-'0\xintSPRaw::csv{\xintSub{\xintSub{\xintSub{\xintAdd{\xintAdd{\xintMul{\xintMul{#1}{#5}}{#9}}{\xintMul{\xintMul{#2}{#6}}{#7}}}{\xintMul{\xintMul{#3}{#4}}{#8}}}{\xintMul{\xintMul{#1}{#6}}{#8}}}{\xintMul{\xintMul{#2}{#4}}{#9}}}{\xintMul{\xintMul{#3}{#5}}{#7}}}
```

```
\xintNewExpr\FA[3]{ #1*#1+#2*#2+#3*#3-(#1*#2+#2*#3+#3*#1) }
\printnumber{\meaning\FA }
macro:#1#2#3->\romannumeral-'0\xintSPRaw::csv{\xintSub{\xintAdd{\xintAdd{\xintMul{#1}{#1}}{\xintMul{#2}{#2}}}{\xintMul{#3}{#3}}}{\xintAdd{\xintAdd{\xintMul{#1}{#2}}{\xintMul{#2}{#3}}}{\xintMul{#3}{#1}}}}
```

One can even do some quite daring things:

```
\xintNewExpr\FA[5]{[#1..#2]..#3}[#4:#5]}
And this works:
\begin{itemize}[nosep]
\item \FA{1}{3}{90}{20}{30}
\item \FA{1}{3}{90}{-40}{-15}
\item \FA{1.234}{-0.123}{-10}{3}{7}
\end{itemize}
\oodef\test {\FA {0}{10}{100}{3}{6}}\meaning\test ++
```

And this works:

- 61, 64, 67, 70, 73, 76, 79, 82, 85, 88
- 1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43
- 865[-3], 742[-3], 619[-3], 496[-3]

macro:->30, 40, 50+++

In the last example though, do not hope to use empty #4 or #5: this is possible in an expression, because the parser identifies][: or :] and handles them appropriately. Here the macro \FA is built with idea that there is something non-empty as it found the place holders #4 and #5.

11.7.1 Conditional operators and \NewExpr

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

```
\xintNewExpr\Formula [3]{ if((#1>#2) && (#2>#3), sqrt(#1-#2)*sqrt(#2-#3), #1^2+#3/#2) }%
\printnumber{\meaning\Formula }
macro:#1#2#3->\romannumeral-'0\xintSPRaw::csv{\xintiiifNotZero{\xintAND{\xintGt{#1}{#2}}{\xintGt{#2}{#3}}}{\xintMul{\XINTinFloatSqrtdigits{\xintSub{#1}{#2}}{\XINTinFloatSqrtdigits{\xintSub{#2}{#3}}}}{\xintAdd{\xintPow{#1}{2}}{\xintDiv{#3}{#2}}}}
```

This formula (with its \xintiiifNotZero) will gobble the false branch without evaluating it when used with given arguments.

Remark: the meaning above reveals some of the private macros used by the package. They are not for direct use.

Another example

```
\xintNewExpr\myformula[3]{ ifsgn(#1,#2/#3,#2-#3,#2*#3) }%
```

```
\meaning\myformula
```

```
macro:#1#2#3->\romannumeral -'0\xintSPRaw::csv {\xintiifSgn {#1}{\xintDiv {#2}{#3}}{\xintSub {#2}{#3}}{\xintMul {#2}{#3}}}
```

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

11.7.2 External macros and `\NewExpr`; the `protect` function

Changed → For macros within such a created *xint*-formula command, there are two cases:

- the macro does not involve the numbered parameters in its arguments: it may then be left as is, and will be evaluated once during the construction of the formula,
- it does involve at least one of the macro parameters as argument. Then:



the whole thing (macro + argument) should be `protect`-ed, not in the \TeX sense (!), but in the following way: `protect(\macro {#1})`.

Here is a silly example illustrating the general principle: the macros here have equivalent functional forms which are more convenient; but some of the more obscure package macros of *xint* dealing with integers do not have functions pre-defined to be in correspondance with them, use this mechanism could be applied to them.

```
\xintNewExpr\myformI[2]{protect(\xintRound{#1}{#2}) - protect(\xintTrunc{#1}{#2})}%
\meaning\myformI
```

```
\xintNewIIExpr\formula [3]{rem(#1,quo(protect(\the\numexpr #2\relax),#3))}%
\noindent\meaning\formula
```

```
macro:#1#2->\romannumeral -'0\xintSPRaw::csv {\xintSub {\xintRound {#1}{#2}}{\xintTrunc {#1}{#2}}}
macro:#1#2#3->\romannumeral -'0\xintCSV::csv {\xintiiRem {#1}{\xintiiQuo {\the \numexpr #2\relax }{#3}}}
```

Only macros involving the `#1`, `#2`, etc. . . should be protected in this way; the `+`, `*`, etc. . . symbols, the functions from the *xintexpr* syntax, none should ever be included in a protected string.

11.7.3 Limitations of `\xintNewExpr`

All depends on where the macro parameters arise `#1`, `#2`, . . . we went to some effort to allow many things but not everything goes through. `\xintNewExpr` tries to evaluate completely as many things as possible which do not involve the macro parameters. A somewhat elaborate scheme allows to handle also complicated situations with list operations:

```
\xintNewIExpr \FA [3] {[3] `+`([1.5..[3.5+#1]..#2]*#3)}
\begin{itemize}[nosep]
\item \FA {3.5}{50}{100} (cf. \xinttheiexpr [3] 1.5..[7]..50\relax)
\item \FA {-15}{-100}{20} (cf. \xinttheiexpr [3] 1.5..[-11.5]..-100\relax)
\item \FA {0}{20}{1} (cf. \xinttheiexpr [3] 1.5..[3.5]..20\relax)
\end{itemize}
```

- 15750.000 (cf. 1.500, 8.500, 15.500, 22.500, 29.500, 36.500, 43.500)
- -8010.000 (cf. 1.500, -10.000, -21.500, -33.000, -44.500, -56.000, -67.500, -79.000, -90.500)
- 61.500 (cf. 1.500, 5.000, 8.500, 12.000, 15.500, 19.000)

Some things are definitely expected not to work therein: particularly the `add`, `mul`, `subs`, `seq`, `q`, `rseq`, `rrseq`, `iter` with `omit`, `abort`, `break`. Also, but this is quite anecdotal, `first` and `last` should not work (I did not try; actually I did not try the functions with dummy letters either, because each time I think about compatibility with `\xintNewExpr`, my head starts spinning.)

Also, using sub-`\xintexpr`-essions (including some of the macro parameters) inside something given to `\xintNewExpr` will probably not work.

Naturally, it is always possible to use, after the macro has been constructed, `\xinttheexpr...` `\relax` among the arguments.

11.8 `\xintiexpr`, `\xinttheiexpr`

- x ★ Equivalent to doing `\xintexpr round(...)\relax`. Thus, only the final result is rounded to an integer. Half integers are rounded towards $+\infty$ for positive numbers and towards $-\infty$ for negative ones. Comma separated lists of expressions are allowed.

New → An optional parameter within brackets is allowed: if strictly positive it instructs the expression to do its final rounding to the nearest value with that many digits after the decimal mark.

11.9 `\xintiiexpr`, `\xinttheiiexpr`

- x ★ This variant does not know fractions. It deals almost only with long integers. Comma separated lists of expressions are allowed.

Changed → 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.

Changed → The `round`, `trunc`, `floor`, `ceil` functions are still available, and are about the only places where fractions can be used, but `/` can not be used! This dilemma is solved using `protect`. For understanding the next example, 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 `\xintiRound` and `\xintiTrunc`.

```
\xinttheiiexpr 5/3, round(5/3,3), trunc(5/3,3), trunc(\xintDiv {5}{3},3),
trunc(\xintRaw {5/3},3)\relax{,
but \xinttheiiexpr 5/3, round(protect(5/3),3), trunc(protect(5/3),3), floor(protect(5/3)),
ceil(protect(5/3))\relax{ works!
```

```
\noindent And with negative numbers: \xinttheiiexpr -5/3, round(protect(-5/3),3),
trunc(protect(-5/3),3), floor(protect(-5/3)), ceil(protect(-5/3))\relax.
```

2, 2000, 2000, 2000, 2000, but 2, 1667, 1666, 1, 2 works!

And with negative numbers: -2, -1667, -1666, -2, -1.

Changed → Decimal numbers and numbers using scientific notations must be given as arguments to one of the `num`, or `round`, or etc. . . functions, which will truncate them to an integer. Internally the number will be represented with as many zeros as is necessary, thus one does not want to do `num(1e100000)` for example!

```
\xinttheiiexpr num(13.4567e3)+num(10000123e-3)\relax % should 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 `\xintiSqrt` (this gives the truncated not the rounded square root).

One can use the Float macros if one is careful to use `num`, or `round` 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 `\xintiexpr` is to gain some speed in *integer-only* algorithms, and the above explanations related to how to nevertheless use fractions therein are a bit peripheral. We observed of the order of 30% speed gain when dealing with numbers with circa one hundred digits, but this gain decreases the longer the manipulated numbers become and becomes negligible for numbers with thousand digits: the overhead from parsing fraction format is little compared to other expensive aspects of the expandable shuffling of tokens.

11.10 `\xintboolexpr`, `\xinttheboolexpr`

- x ★ Equivalent to doing `\xintexpr ... \relax` and returning 1 if the result does not vanish, and 0 if the result is zero. As `\xintexpr`, this can be used on comma separated lists of expressions, and will return a comma separated list of 0's and 1's.

11.11 `\xintfloatexpr`, `\xintthefloatexpr`

- x ★ `\xintfloatexpr ... \relax` is exactly like `\xintexpr ... \relax` but with the four binary operations and the power function mapped to `\xintFloatAdd`, `\xintFloatSub`, `\xintFloatMul`, `\xintFloatDiv` and `\xintFloatPower`. The precision for the computation is from the current setting of `\xintDigits`. Comma separated lists of expressions are allowed.

An optional parameter within brackets is allowed: the final float will have that many digits of precision. This is provided to get rid of non-relevant last digits.

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

Note that `1.000000001` and `(1+1e-9)` will not be equivalent for `D=\xinttheDigits` set to nine or less. Indeed the addition implicit in `1+1e-9` (and executed when the closing parenthesis is found) will provoke the rounding to 1. Whereas `1.000000001`, when found as operand of one of the four elementary operations is kept with `D+2` digits, and even more for the power function.

`\xintDigits:= 9; \xintthefloatexpr (1+1e-9)-1 \relax=0`

`\xintDigits:= 9; \xintthefloatexpr 1.000000001-1 \relax=1.00000000e-9`

For the fun of it: `\xintDigits:=20;`

`\xintthefloatexpr (1+1e-7)^1e7 \relax=2.7182816925449662712`

`\xintDigits:=36;`

`\xintthefloatexpr ((1/13+1/121)*(1/179-1/173))/(1/19-1/18) \relax`

`0.00564487459334466559166166079096852897`

`\xintFloat{\xinttheexpr ((1/13+1/121)*(1/179-1/173))/(1/19-1/18) \relax}`

`5.64487459334466559166166079096852912e-3`

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

Using `\xintthefloatexpr` only pays off compared to using `\xinttheexpr` followed with `\xintFloat` if the computations turn out to involve hundreds of digits. For elementary calculations with hand written numbers (not using the scientific notation with exponents differing greatly) it will generally be more efficient to use `\xinttheexpr`. The situation is quickly otherwise if one starts

New

→

using the Power function. Then, `\xintthefloat` is often useful; and sometimes indispensable to achieve the (approximate) computation in reasonable time.

We can try some crazy things:

```
\xintDigits:=12;\xintthefloatexpr 1.000000000000001^1e15\relax
2.71828182846
```

Contrarily to some professional computing software which are our concurrents on this market, the `1.000000000000001` wasn't rounded to 1 despite the setting of `\xintDigits`; it would have been if we had input it as `(1+1e-15)`.

11.12 `\xintifboolexpr`

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. `<expr>` can involve various `&` and `|`, parentheses, `all`, `any`, `xor`, the `bool` or `togl` operators, but is not limited to them: the most general computation can be done, the test is on whether the outcome of the computation vanishes or not.

Will not work on an expression composed of comma separated sub-expressions.

11.13 `\xintifboolfloatexpr`

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.

11.14 `\xintifbooliiexpr`

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.

11.15 `\xintNewFloatExpr`

This is exactly like `\xintNewExpr` except that the created formulas are set-up to use `\xintthefloatexpr`. The precision used for the computation will be the one given by `\xintDigits` at the time of use of the created formulas. However, the numbers hard-wired in the original expression will have been evaluated with the then current setting for `\xintDigits`.

```
\xintNewFloatExpr \f [1] {sqrt(#1)}
\f {2} (with \xinttheDigits{} of precision).
```

```
{\xintDigits := 32;\f {2} (with \xinttheDigits{} of precision).}
```

```
\xintNewFloatExpr \f [1] {sqrt(#1)*sqrt(2)}
\f {2} (with \xinttheDigits {} of precision).
```

```
{\xintDigits := 32;\f {2} (?? we thought we had a higher precision.Explanation next)}
```

The `sqrt(2)` in the second formula was computed with only `\xinttheDigits{} of precision`. Setting `|\xintDigits|` to a higher value at the time of definition will confirm that the result above is from a mismatch of the precision for `|sqrt(2)|` at the time of its evaluation and the precision for the new `|sqrt(2)|` with `|#1=2|` at the time of use.

```
{\xintDigits := 32;\xintNewFloatExpr \f [1] {sqrt(#1)*sqrt(2)}
\f {2} (with \xinttheDigits {} of precision)}
```

`1.414213562373095 (with 16 of precision).`

`1.4142135623730950488016887242097 (with 32 of precision).`

2.0000000000000000 (with 16 of precision).

1.9999999999999999309839899395125 (?? we thought we had a higher precision. Explanation next)

The `\sqrt{2}` in the second formula was computed with only 16 of precision. Setting `\xintDigits` to a higher value at the time of definition will confirm that the result above is from a mismatch of the precision for `\sqrt{2}` at the time of its evaluation and the precision for the new `\sqrt{2}` with `#1=2` at the time of use.

2.00000000000000000000000000000000 (with 32 of precision)

11.16 `\xintNewIExpr`

Like `\xintNewExpr` but using `\xinttheiexpr`.

11.17 `\xintNewIIExpr`

Like `\xintNewExpr` but using `\xinttheiiexpr`.

11.18 `\xintNewBoolExpr`

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

11.19 `\xintthecoords`

From a comma separated output of an even number of items as output by `\xintfloatexpr [P] ... \relax` or by `\xintiexpr [D] ... \relax`, creates coordinate pairs for TikZ. See [an example](#) on page 32.

11.20 Technicalities

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

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

The `\escapechar` setting may be arbitrary when using `\xintexpr`.

The format of the output of `\xintexpr<stuff>\relax` is a `!` (with catcode 11) followed by various things:

```
\oodef\f {\xintexpr 1.23^10\relax }\meaning\f
macro:->!\XINT_expr_usethe \XINT_protectii \XINT_expr_print \.=792594609605189126649/1[-20]
```

Note that `\xintexpr` is thus compatible with complete expansion, contrarily to `\numexpr` which is non-expandable, if not prefixed by `\the` or `\number`, and away from contexts where \TeX is building a number. See [subsection 7.24](#) for some illustration.

I decided to put all intermediate results (from each evaluation of an infix operators, or of a parenthesized subpart of the expression, or from application of the minus as prefix, or of the exclamation sign as postfix, or any encountered braced material) inside `\csname...\endcsname`, as this can be done expandably and encapsulates an arbitrarily long fraction in a single token (left with undefined meaning), thus providing tremendous relief to the programmer in his/her expansion control.

As the `\xintexpr` computations corresponding to functions and infix or postfix operators are done inside `\csname...\endcsname`, the *f*-expandability could possibly be dropped and one could imagine implementing the basic operations with expandable but not *f*-expandable macros (as `\xintXTrunc`.) I have not investigated that possibility.

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

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

However, this mechanism is completely inoperant for parentheses involved in the syntax of the `seq`, `add`, `mul`, `subs`, `rseq` and `rrseq` functions.

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

11.21 Acknowledgements

I was greatly helped in my preparatory thinking, prior to producing such an expandable parser, by the commented source of the `l3fp` package, specifically the `l3fp-parse.dtx` file (in the version of April-May 2013). 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.

12 Commands of the `xintbinhex` package

.1	<code>\xintDecToHex</code>	107	.5	<code>\xintBinToHex</code>	108
.2	<code>\xintDecToBin</code>	107	.6	<code>\xintHexToBin</code>	108
.3	<code>\xintHexToDec</code>	108	.7	<code>\xintCHexToBin</code>	108
.4	<code>\xintBinToDec</code>	108			

This package was first included in the 1.08 (2013/06/07) release of `xint`. It provides expandable conversions of arbitrarily big integers to and from binary and hexadecimal.

The argument is first *f*-expanded. It then may start with an optional minus sign (unique, of category code other), followed with optional leading zeroes (arbitrarily many, category code other) and then ‘‘digits’’ (hexadecimal letters may be of category code letter or other, and must be up-percased). The optional (unique) minus sign (plus sign is not allowed) is kept in the output. Leading zeroes are allowed, and stripped. The hexadecimal letters on output are of category code letter, and uppercased.

12.1 `\xintDecToHex`

f ★ Converts from decimal to hexadecimal.

```
\xintDecToHex{2718281828459045235360287471352662497757247093699959574966967627724076630353}
547594571382178525166427427466391932003}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C6032936BF37DAC918
814C63
```

12.2 `\xintDecToBin`

f ★ Converts from decimal to binary.

```
\xintDecToBin{2718281828459045235360287471352662497757247093699959574966967627724076630353}
547594571382178525166427427466391932003}
```

```
->1000110101001001110010111110001100110100101001001101010010111000001010001111101111101000010
010100000010111100100010100111000111110000010110001011111000100000110110001000111000100100010
011101011101111001010110101011101100000101110110011100011010010011100101111010001101101110010
11001000110110001100000001100101001001101101011111001101111011011001001000110001000000100
100110001100011
```

12.3 \xintHexToDec

f ★ Converts from hexadecimal to decimal.

```
\xintHexToDec{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C603}
2936BF37DAC918814C63}
->2718281828459045235360287471352662497757247093699959574966967627724076630353547594571382170
8525166427427466391932003
```

12.4 \xintBinToDec

f ★ Converts from binary to decimal.

```
\xintBinToDec{10001101010010011100101111100011001101001010010011010100101110000010100011110
101111101000010101000000101111001000101001110001111100000101100010111110001000001101100010000
111000100100010111010111011110010101101010111011000001011101100111000110100100111001011110100
00110110111001110010001101100011000000011001010010011011010111110011011110110110010010000
11000100000010100110001100011}
->2718281828459045235360287471352662497757247093699959574966967627724076630353547594571382170
8525166427427466391932003
```

12.5 \xintBinToHex

f ★ Converts from binary to hexadecimal.

```
\xintBinToHex{10001101010010011100101111100011001101001010010011010100101110000010100011110
101111101000010101000000101111001000101001110001111100000101100010111110001000001101100010000
111000100100010111010111011110010101101010111011000001011101100111000110100100111001011110100
0011011011100111001000110110001100000001100101001001101101011111100110111110110110010010000
11000100000010100110001100011}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C6032936BF37DAC9188
814C63
```

12.6 \xintHexToBin

f ★ Converts from hexadecimal to binary.

```
\xintHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C603}
2936BF37DAC918814C63}
->1000110101001001110010111110001100110100101001001101010010111000001010001111101111101000010
010100000010111100100010100111000111110000010110001011111000100000110110001000111000100100010
011101011101111001010110101011101100000101110110011100011010010011100101111010001101101110010
11001000110110001100000001100101001001101101011111001101111011011001001000110001000000100
100110001100011
```

12.7 \xintCHexToBin

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

```
\xintCHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C60}
32936BF37DAC918814C63}
```

```

->1000110101001001110010111110001100110100101001001101010010111000001010001111101111101000010
010100000010111100100010100111000111110000010110001011111000100000110110001000111000100100010
01110101110111001010110101011101100000101110110011100011010010011100101111010001101101110010
1100100011011000110000000110010100100110110101111100110111101101011001001000110001000000100
100110001100011

```

13 Commands of the `xintgcd` package

.1	<code>\xintGCD</code> , <code>\xintiiGCD</code>	109	.6	<code>\xintEuclideanAlgorithm</code>	110
.2	<code>\xintGCDof</code>	109	.7	<code>\xintBezoutAlgorithm</code>	110
.3	<code>\xintLCM</code> , <code>\xintiiLCM</code>	109	.8	<code>\xintTypesetEuclideanAlgorithm</code>	110
.4	<code>\xintLCMof</code>	109	.9	<code>\xintTypesetBezoutAlgorithm</code>	110
.5	<code>\xintBezout</code>	109			

This package was included in the original release 1.0 (2013/03/28) of the `xint` bundle.

Since release 1.09a the macros filter their inputs through the `\xintNum` macro, so one can use count registers, or fractions as long as they reduce to integers.

Since release 1.1, the two “`typeset`” macros require the explicit loading by the user of package

Changed → `xinttools`.

13.1 `\xintGCD`, `\xintiiGCD`

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

```
\xintGCD{10000}{1113}=1
```

```
\xintiiGCD{123456789012345}{9876543210321}=3
```

`\xintiiGCD` skips the `\xintNum` overhead.

13.2 `\xintGCDof`

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

13.3 `\xintLCM`, `\xintiiLCM`

`\xintGCD{N}{M}` computes the least common multiple. It is 0 if one of the two integers vanishes.

`\xintiiLCM` skips the `\xintNum` overhead.

13.4 `\xintLCMof`

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

13.5 `\xintBezout`

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

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

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

```
\xintAssign {\xintBezout {10000}{1113}}\to\A\B\U\VD
```

```
\A:10000, \B:1113, \U:-131, \V:-1177, \D:1.
```

```
\xintAssign {\xintBezout {123456789012345}{9876543210321}}\to\A\B\U\VD
```

```
\A:123456789012345, \B:9876543210321, \U:256654313730, \V:3208178892607, \D:3.
```

13.6 `\xintEuclideanAlgorithm`

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

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

The first token is the number of steps, the second is *N*, the third is the GCD, the fourth is *M* then the first quotient and remainder, the second quotient and remainder, . . . until the final quotient and last (zero) remainder.

13.7 `\xintBezoutAlgorithm`

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

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

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

13.8 `\xintTypesetEuclideanAlgorithm`

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

```
\xintTypesetEuclideanAlgorithm {123456789012345}{9876543210321}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

13.9 `\xintTypesetBezoutAlgorithm`

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

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

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

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

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

$$1113 = 1 \times 1096 + 17$$

$$9 = 1 \times 8 + 1$$

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

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

14 Commands of the **xintseries** package

.1	<code>\xintSeries</code>	111	.7	<code>\xintFxFtPowerSeries</code>	119
.2	<code>\xintiSeries</code>	112	.8	<code>\xintFxFtPowerSeriesX</code>	120
.3	<code>\xintRationalSeries</code>	113	.9	<code>\xintFloatPowerSeries</code>	121
.4	<code>\xintRationalSeriesX</code>	116	.10	<code>\xintFloatPowerSeriesX</code>	122
.5	<code>\xintPowerSeries</code>	117	.11	Computing $\log 2$ and π	122
.6	<code>\xintPowerSeriesX</code>	119			

This package was first released with version 1.03 (2013/04/14) of the **xint** bundle.

The $\frac{\text{num}}{\text{den}}$ expansion type of various macro arguments is only a $\frac{\text{Num}}{\text{Den}}$ if only **xint** but not **xintfrac** is loaded. The macro `\xintiSeries` is special and expects summing big integers obeying the strict format, even if **xintfrac** is loaded.

The arguments serving as indices are of the $\frac{\text{num}}{\text{den}}$ expansion type.

In some cases one or two of the macro arguments are only expanded at a later stage not immediately.

14.1 `\xintSeries`

$\frac{\text{num}}{\text{den}}$ $\frac{\text{num}}{\text{den}}$ $\frac{\text{Frac}}{\text{den}}$ ★ `\xintSeries{A}{B}{\coeff}` computes $\sum_{n=A}^{n=B} \coeff{n}$. The initial and final indices must obey the `\numexpr` constraint of expanding to numbers at most $2^{31}-1$. The `\coeff` macro must be a one-parameter *f*-expandable command, taking on input an explicit number *n* and producing some number or fraction `\coeff{n}`; it is expanded at the time it is needed.⁶⁵

```

\def\coeff #1{\xintiiMON{#1}/#1.5} % (-1)^n/(n+1/2)
\oodef\w {\xintSeries {0}{50}{\coeff}} % we want to re-use it
\oodef\z {\xintJrr {\w}[0]} % the [0] for a microsecond gain.
% \xintJrr preferred to \xintIrr: a big common factor is suspected.
% But numbers much bigger would be needed to show the greater efficiency.
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n + \frac{1}{2}} = \xintFrac\z \]
```

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

The definition of `\coeff` as `\xintiiMON{#1}/#1.5` is quite suboptimal. It allows *#1* to be a big integer, but anyhow only small integers are accepted as initial and final indices (they are of the $\frac{\text{num}}{\text{den}}$ 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`:

$$\frac{24489212733740439818553118189578822128979076445102691650390625}{154936248757874299375548246172975814272155426442623138427734375} 10^1$$

⁶⁵ `\xintiiMON` is like `\xintMON` but does not parse its argument through `\xintNum`, for efficiency; other macros of this type are `\xintiiAdd`, `\xintiiMul`, `\xintiiSum`, `\xintiiPrd`, `\xintiiMMON`, `\xintiiLDg`, `\xintiiFDg`, `\xintiiOdd`, ...

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.
\oodef\w {\xintSeries {0}{50}\coeff}
\[\sum_{n=0}^{50} \frac{(-1)^n}{n+\frac{1}{2}} = \frac{\xintFrac\w}{103975956465623552858889521778607543952793375}
```

The reduced form `\z` as displayed above only differs from this one by a factor of 945.

```
\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...

14.2 `\xintiSeries`

$\sum_{n=A}^B f$

`\xintiSeries{A}{B}\coeff` computes $\sum_{n=A}^{n=B} \coeff{n}$ where `\coeff{n}` must *f*-expand to a (possibly long) integer in the strict format.

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

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

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.
\l \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}} \approx
  \xintTrunc {40}{\xintiSeries {0}{50}{\coeff}[-40]}\l
\def\exactcoeff #1%
  {\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}}%
\l \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}}
  = \xintTrunc {50}{\xintSeries {0}{50}{\exactcoeff}}\dots\l

```

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

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

This shows indeed that our sum of truncated terms estimated wrongly the 39th and 40th digits of the exact result⁶⁶ and that the sum of rounded terms fared a bit better.

14.3 `\xintRationalSeries`

`\xintRationalSeries{A}{B}{f}{\ratio}` evaluates $\sum_{n=A}^{n=B} F(n)$, where $F(n)$ is specified indirectly via the data of $f=F(A)$ and the one-parameter macro `\ratio` which must be such that `\macro{n}` expands to $F(n)/F(n-1)$. The name indicates that `\xintRationalSeries` was designed to be useful in the cases where $F(n)/F(n-1)$ is a rational function of n but it may be anything expanding to a fraction. The macro `\ratio` must be an expandable-only compatible command and expand to its value after iterated full expansion of its first token. A and B are fed to a `\numexpr` hence may be count registers or arithmetic expressions built with such; they must obey the \TeX bound. The initial term f may be a macro `\f`, it will be expanded to its value representing $F(A)$.

```

\def\ratio #1{2/#1[0]}% 2/n, to compute exp(2)
\cnta 0 % previously declared count
\begin{quote}\normalcolor
\loop \oodef z {\xintRationalSeries {0}{\cnta}{1}{\ratio }}%
\noindent$\sum_{n=0}^{\cnta} \frac{2^n}{n!} =
  \xintTrunc{12}{z}\dots=
  \xintFrac{z}{{}}$\textcolor[named]{OrangeRed}{$\xintFrac{\xintIrr{z}}{}}\vtop to 5pt{}\par
\ifnum\cnta<20 \advance\cnta 1 \repeat
\end{quote}

```

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

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

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

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

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

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

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

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

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

⁶⁶ as the series is alternating, we can roughly expect an error of $\sqrt{40}$ and the last two digits are off by 4 units, which is not contradictory to our expectations.

$$\begin{aligned}
 \sum_{n=0}^9 \frac{2^n}{n!} &= 7.388712522045 \dots = \frac{2681216}{362880} = \frac{20947}{2835} \\
 \sum_{n=0}^{10} \frac{2^n}{n!} &= 7.388994708994 \dots = \frac{26813184}{3628800} = \frac{34913}{4725} \\
 \sum_{n=0}^{11} \frac{2^n}{n!} &= 7.389046015712 \dots = \frac{294947072}{39916800} = \frac{164591}{22275} \\
 \sum_{n=0}^{12} \frac{2^n}{n!} &= 7.389054566832 \dots = \frac{3539368960}{479001600} = \frac{691283}{93555} \\
 \sum_{n=0}^{13} \frac{2^n}{n!} &= 7.389055882389 \dots = \frac{46011804672}{6227020800} = \frac{14977801}{2027025} \\
 \sum_{n=0}^{14} \frac{2^n}{n!} &= 7.389056070325 \dots = \frac{644165281792}{87178291200} = \frac{314533829}{42567525} \\
 \sum_{n=0}^{15} \frac{2^n}{n!} &= 7.389056095384 \dots = \frac{9662479259648}{1307674368000} = \frac{4718007451}{638512875} \\
 \sum_{n=0}^{16} \frac{2^n}{n!} &= 7.389056098516 \dots = \frac{154599668219904}{20922789888000} = \frac{1572669151}{212837625} \\
 \sum_{n=0}^{17} \frac{2^n}{n!} &= 7.389056098884 \dots = \frac{2628194359869440}{355687428096000} = \frac{16041225341}{2170943775} \\
 \sum_{n=0}^{18} \frac{2^n}{n!} &= 7.389056098925 \dots = \frac{47307498477912064}{6402373705728000} = \frac{103122162907}{13956067125} \\
 \sum_{n=0}^{19} \frac{2^n}{n!} &= 7.389056098930 \dots = \frac{898842471080853504}{121645100408832000} = \frac{4571749222213}{618718975875} \\
 \sum_{n=0}^{20} \frac{2^n}{n!} &= 7.389056098930 \dots = \frac{17976849421618118656}{2432902008176640000} = \frac{68576238333199}{9280784638125}
 \end{aligned}$$

```

\def\ratio #1{-1/#1[0]}% -1/n, comes from the series of exp(-1)
\cnta 0 % previously declared count
\begin{quote}\normalcolor
\loop
\oodef\z {\xintRationalSeries {0}{\cnta}{1}{\ratio }}%
\noindent$\sum_{n=0}^{\the\cnta} \frac{(-1)^n}{n!}=
\xintTrunc{20}\z\z\z=\xintFrac{\z}{\z}\textcolor{named}{OrangeRed}{\xintFrac{\xintIrr\z}{\z}}
\vtop to 5pt{}\par
\ifnum\cnta<20 \advance\cnta 1 \repeat
\end{quote}

```

$$\begin{aligned}
 \sum_{n=0}^0 \frac{(-1)^n}{n!} &= 1.00000000000000000000 \dots = 1 = 1 \\
 \sum_{n=0}^1 \frac{(-1)^n}{n!} &= 0 \dots = 0 = 0 \\
 \sum_{n=0}^2 \frac{(-1)^n}{n!} &= 0.50000000000000000000 \dots = \frac{1}{2} = \frac{1}{2} \\
 \sum_{n=0}^3 \frac{(-1)^n}{n!} &= 0.33333333333333333333 \dots = \frac{2}{6} = \frac{1}{3} \\
 \sum_{n=0}^4 \frac{(-1)^n}{n!} &= 0.37500000000000000000 \dots = \frac{9}{24} = \frac{3}{8} \\
 \sum_{n=0}^5 \frac{(-1)^n}{n!} &= 0.36666666666666666666 \dots = \frac{44}{120} = \frac{11}{30} \\
 \sum_{n=0}^6 \frac{(-1)^n}{n!} &= 0.36805555555555555555 \dots = \frac{265}{720} = \frac{53}{144} \\
 \sum_{n=0}^7 \frac{(-1)^n}{n!} &= 0.36785714285714285714 \dots = \frac{1854}{5040} = \frac{103}{280} \\
 \sum_{n=0}^8 \frac{(-1)^n}{n!} &= 0.36788194444444444444 \dots = \frac{14833}{40320} = \frac{2119}{5760} \\
 \sum_{n=0}^9 \frac{(-1)^n}{n!} &= 0.36787918871252204585 \dots = \frac{133496}{362880} = \frac{16687}{45360} \\
 \sum_{n=0}^{10} \frac{(-1)^n}{n!} &= 0.36787946428571428571 \dots = \frac{1334961}{3628800} = \frac{16481}{44800} \\
 \sum_{n=0}^{11} \frac{(-1)^n}{n!} &= 0.36787943923360590027 \dots = \frac{14684570}{39916800} = \frac{1468457}{3991680} \\
 \sum_{n=0}^{12} \frac{(-1)^n}{n!} &= 0.36787944132128159905 \dots = \frac{176214841}{479001600} = \frac{16019531}{43545600} \\
 \sum_{n=0}^{13} \frac{(-1)^n}{n!} &= 0.36787944116069116069 \dots = \frac{2290792932}{6227020800} = \frac{63633137}{172972800}
 \end{aligned}$$

$$\begin{aligned}\sum_{n=0}^{14} \frac{(-1)^n}{n!} &= 0.36787944117216190628 \dots = \frac{32071101049}{87178291200} = \frac{2467007773}{6706022400} \\ \sum_{n=0}^{15} \frac{(-1)^n}{n!} &= 0.36787944117139718991 \dots = \frac{481066515734}{1307674368000} = \frac{34361893981}{93405312000} \\ \sum_{n=0}^{16} \frac{(-1)^n}{n!} &= 0.36787944117144498468 \dots = \frac{7697064251745}{20922789888000} = \frac{15549624751}{42268262400} \\ \sum_{n=0}^{17} \frac{(-1)^n}{n!} &= 0.36787944117144217323 \dots = \frac{130850092279664}{355687428096000} = \frac{8178130767479}{22230464256000} \\ \sum_{n=0}^{18} \frac{(-1)^n}{n!} &= 0.36787944117144232942 \dots = \frac{2355301661033953}{6402373705728000} = \frac{138547156531409}{376610217984000} \\ \sum_{n=0}^{19} \frac{(-1)^n}{n!} &= 0.36787944117144232120 \dots = \frac{44750731559645106}{121645100408832000} = \frac{92079694567171}{250298560512000} \\ \sum_{n=0}^{20} \frac{(-1)^n}{n!} &= 0.36787944117144232161 \dots = \frac{895014631192902121}{2432902008176640000} = \frac{4282366656425369}{11640679464960000}\end{aligned}$$

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

[illegible]

Here is a slightly more complicated evaluation:

```

\cnta 1
\begin{multicols}{2}\normalcolor
\loop \oodef\z {\xintRationalSeries
      {\cnta}
      {2*\cnta-1}
      {\xintiPow {\the\cnta}{\cnta}/\xintFac{\cnta}}
      {\ratioexp{\the\cnta}}}%
\oodef\w {\xintRationalSeries {0}{2*\cnta-1}{1}{\ratioexp{\the\cnta}}}%
\noindent

$$\sum_{n=\text{the}\cnta}^{\text{the}\numexpr 2*\cnta-1\relax} \frac{\text{the}\cnta^n \{n!\}}{\sum_{n=0}^{\text{the}\numexpr 2*\cnta-1\relax} \frac{\text{the}\cnta^n \{n!\}}{\text{ratioexp}\{\text{the}\cnta\}}} = \{\}$$

\textcolor[named]{OrangeRed}{ $\text{xintTrunc}\{8\}\{\text{xintDiv}\z\w\}\text{dots}\}$  \vtop to 5pt}\endgraf
\ifnum\cnta<20 \advance\cnta 1 \repeat
\end{multicols}

```

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

14.4 `\xintRationalSeriesX`

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

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

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

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

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

```

\def\firstterm #1{1[0]}% first term of the exponential series
% although it is the constant 1, here it must be defined as a
% one-parameter macro. Next comes the ratio function for exp:
\def\ratioexp #1#2{\xintDiv {#1}{#2}}% x/n
% These are the (-1)^{n-1}/n of the log(1+h) series:
\def\coefflog #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}%
% Let L(h) be the first 10 terms of the log(1+h) series and
% let E(t) be the first 10 terms of the exp(t) series.
% The following computes E(L(a/10)) for a=1,...,12.
\begin{multicols}{3}\raggedcolumns
\cnta 0
\loop
\noindent\xintTrunc {18}{%
\xintRationalSeriesX {0}{9}{\firstterm}{\ratioexp}
{\xintPowerSeries{1}{10}{\coefflog}{\the\cnta[-1]}}\dots
}\endgraf
\ifnum\cnta < 12 \advance \cnta 1 \repeat
\end{multicols}

```

1.00000000000000000000...	1.499954310225476533...	1.907197560339468199...
1.099999999999083906...	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]))=8375881917087635932335671746871686868390437672500403080051779277524946298661842
007635859046126459421356378683685505198743151968256/761443810644964678986073374720000000000[-2
-90] (length of numerator: 129)
```

```
E(L(12[-2]))=852817067917170404743367455694175958595958834849647782275488039683039028140142
627504107752450752216393977420945142612214186817956418552109426998619906107437170238454024542
66764149586292947053094306437899694978421745516544/761443810644964678986073374720000000000[-2
180] (length of numerator: 219)
```

```
E(L(123[-3]))=85510139934748417382331557840332916647230257508795878817033811296662463224972
19142331115922733076372551695958545295492787962370297931258555334838173016993380276766882172
119374626466595481228434620385193134459556100336250033905803253996685925486649200488748729352
7193753147231854444914269381901852177272347295744/761443810644964678986073374720000000000[-22
70] (length of numerator: 309)
```

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))=246928403777284535542005925363994435551841707513610888309477702059412077893312772
8456047080539072743831005696/2160623533139083201661118734213279886660128668842038173763451722
792064291516013203517727545795543040000000000[0] (length of numerator: 108; length of denomina-
tor: 108)
```

```
E(L(1/71))=31693003114420993773601541095216519886197494696368120129960391285745860214771192
416176332877917435184310890684584136895442264511996499581982071012712604174613973484964435392
24405185605375986452422252984559104/312528225156095910823136583628983100903663468032171880552
179095599208106488310703560230149547919271401955564834180272077013758540888195207115933497192
4372055039374604062422190122738418253432587550720000000000[0] (length of numerator: 206; length
of denominator: 206)
```

```
E(L(1/712))=3003564353778406020559670408411885925389093114199308387996560136260710297841742
496819290884958041362038132421744055614153154268292413172870530372734533290558141538915173252
756941123200263645694953665349180314390511046104875297961920582057259996416578066159049290482
98946463533146662233869249/299935178105220909766968489591763105361775507557039697364359215352
224604108923285325397380419112021214124247158817340492547166400824709873409851519325042814942
240645967888744414705331478482078635497788470006171032646666387826770190191301139308374215312
810478062025966102914017525760000000000[0] (length of numerator: 288; length of denominator:
288)
```

Thus decimal numbers such as *0.123* (equivalently *123[-3]*) give less computing intensive tasks than fractions such as *1/712*: in the case of decimal numbers the (raw) denominators originate in the coefficients of the series themselves, powers of ten of the input within brackets being treated separately. And even then the numerators will grow with the size of the input in a sort of linear way, the coefficient being given by the order of series: here 10 from the log and 9 from the exp, so 90. One more digit in the input means 90 more digits in the numerator of the output: obviously we can not go on composing such partial sums of series and hope that *xint* will joyfully do all at the speed of light!

Hence, truncating the output (or better, rounding) is the only way to go if one needs a general calculus of special functions. This is why the package *xintseries* provides, besides *\xintSeries*, *\xintRationalSeries*, or *\xintPowerSeries* which compute exact sums, *\xintFxFtPowerSeries* for fixed-point computations and a (tentative naive) *\xintFloatPowerSeries*.

14.5 *\xintPowerSeries*

\xintPowerSeries{A}{B}{\coeff}{f} evaluates the sum $\sum_{n=A}^{n=B} \text{\coeff}\{n\} \cdot f^n$. The initial and final indices are given to a *\numexpr* expression. The *\coeff* macro (which, as argument to *\xintPowerSeries*

is expanded only at the time `\coeff{n}` is needed) should be defined as a one-parameter expandable command, its input will be an explicit number.

The `f` can be either a fraction directly input or a macro `\f` expanding to such a fraction. It is actually more efficient to encapsulate an explicit fraction `f` in such a macro, if it has big numerators and denominators ('big' means hundreds of digits) as it will then take less space in the processing until being (repeatedly) used.

This macro computes the exact result (one can use it also for polynomial evaluation), 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]}%
\[\log 2 \approx \sum_{n=1}^{n=20} \frac{1}{n \cdot 2^n}
=\xintFrac {\xintIrr {\xintPowerSeries {1}{20}{\coefflog}{\f}}}\]
\[\log 2 \approx \sum_{n=1}^{n=50} \frac{1}{n \cdot 2^n}
=\xintFrac {\xintIrr {\xintPowerSeries {1}{50}{\coefflog}{\f}}}\]
```

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

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

```
\setlength{\columnsep}{0pt}
\begin{multicols}{3}
\cnta 1 % previously declared count
\loop % in this loop we recompute from scratch each partial sum!
% we can afford that, as \xintPowerSeries is fast enough.
\noindent\hbox to 2em{\hfil\texttt{\the\cnta.} }%
\xintTrunc {12}
{\xintPowerSeries {1}{\cnta}{\coefflog}{\f}}\dots
\endgraf
\ifnum \cnta < 30 \advance\cnta 1 \repeat
\end{multicols}
```

1. 0.500000000000...	11. 0.693109245355...	21. 0.693147159757...
2. 0.625000000000...	12. 0.693129590407...	22. 0.693147170594...
3. 0.666666666666...	13. 0.693138980431...	23. 0.693147175777...
4. 0.682291666666...	14. 0.693143340085...	24. 0.693147178261...
5. 0.688541666666...	15. 0.693145374590...	25. 0.693147179453...
6. 0.691145833333...	16. 0.693146328265...	26. 0.693147180026...
7. 0.692261904761...	17. 0.693146777052...	27. 0.693147180302...
8. 0.692750186011...	18. 0.693146988980...	28. 0.693147180435...
9. 0.692967199900...	19. 0.693147089367...	29. 0.693147180499...
10. 0.693064856150...	20. 0.693147137051...	30. 0.693147180530...

```
\def\coeffarctg #1{1/\the\numexpr\ifodd #1 -2*#1-1\else2*#1+1\fi\relax }%
```

```
% the above gives (-1)^n/(2n+1). The sign being in the denominator,
%      **** no [0] should be added ****,
% else nothing is guaranteed to work (even if it could by sheer luck)
% Notice in passing this aspect of \numexpr:
%      **** \numexpr -(1)\relax is ilegal !!! ****
\def\fr{1/25[0]}% 1/5^2
\[\mathrm{Arctg}(\frac{1}{5})\approx \frac{1}{5}\sum_{n=0}^{15} \frac{(-1)^n}{(2n+1)25^n} = \frac{165918726519122955895391793269168}{840539304153062403202056884765625}
```

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

14.6 \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...	0.499511320760604148...	-1.597091692317639401...
0.199999995263443554...	0.593980619762352217...	-12.648937932093322763...
0.299999338075041781...	0.645144282733914916...	-66.259639046914679687...
0.399974460740121112...	0.398118280111436442...	-304.768437445462801227...

14.7 \xintFxPtPowerSeries

`\xintFxPtPowerSeries{A}{B}{\coeff}{f}{D}` computes $\sum_{n=A}^{n=B} \text{coeff}\{n\} \cdot f^n$ with each term of the series truncated to `D` digits after the decimal point. As usual, `A` and `B` are completely expanded through their inclusion in a `\numexpr` expression. Regarding `D` it will be similarly be expanded each time it is used inside an `\xintTrunc`. The one-parameter macro `\coeff` is similarly expanded at the time it is used inside the computations. Idem for `f`. If `f` itself is some complicated macro it is thus better to use the variant `\xintFxPtPowerSeriesX` which expands it first and then uses the result of that expansion.

The current (1.04) implementation is: the first power `f^A` is computed exactly, then truncated. Then each successive power is obtained from the previous one by multiplication by the

exact value of f , and truncated. And $\backslash\text{coeff}\{n\}\cdot f^n$ is obtained from that by multiplying by $\backslash\text{coeff}\{n\}$ (untruncated) and then truncating. Finally the sum is computed exactly. Apart from that $\backslash\text{xintFxpTPowerSeries}$ (where *FxpT* means ‘fixed-point’) is like $\backslash\text{xintPowerSeries}$.

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 $\backslash\text{xintFxpTPowerSeries}$ does not compute f^n from scratch at each n . Perhaps in the next package release.

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

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

```
\def\coeffexp #1{1/\xintFac {#1}[0]}% 1/n!
\def\ f {-1/2[0]}% [0] for faster input parsing
\cnta 0 % previously declared \count register
\noindent\loop
$\xintFxpTPowerSeries {0}{\cnta}{\coeffexp}{\f}{20}$\\
\ifnum\cnta<19 \advance\cnta 1 \repeat\par
% One should not trust the final digits, as the potential truncation
% errors of up to  $10^{-20}$  per term accumulate and never disappear! (the
% effect is attenuated by the alternating signs in the series). We can
% confirm that the last two digits (of our evaluation of the nineteenth
% partial sum) are wrong via the evaluation with more digits:
\xintFxpTPowerSeries {0}{19}{\coeffexp}{\f}{25}= 0.6065306597126334236037992
```

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

```
\xintPowerSeries {0}{19}{\coeffexp}{\f} = \frac{38682746160036397317757}{63777066403145711616000}
= 0.606530659712633423603799152126...
```

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.

14.8 $\backslash\text{xintFxpTPowerSeriesX}$

$\backslash\text{xintFxpTPowerSeriesX}\{A\}\{B\}\{\text{coeff}\}\{f\}\{D\}$ computes, exactly as $\backslash\text{xintFxpTPowerSeries}$, the sum of $\text{coeff}\{n\}\cdot f^n$ from $n=A$ to $n=B$ with each term of the series being truncated to D digits after the decimal point. The sole difference is that $\backslash f$ is first expanded and it is the result of this which is used in the computations.

Let us illustrate this on the numerical exploration of the identity

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

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

```
\cnta 0
\def\coefflog #1{the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}% (-1)^{n-1}/n
\def\coeffalt #1{the\numexpr\ifodd#1 -1\else1\fi\relax [0]}% (-1)^n
\begin{multicols}2
\loop
```

```

\noindent \hbox to 2.5cm {\hss\texttt{D(\the\cnta/100): }}%
\xintAdd {\xintFxFtPowerSeriesX {1}{10}{\coefflog}{\the\cnta [-2]}\{5}}
{\xintFxFtPowerSeriesX {1}{10}{\coefflog}
{\xintFxFtPowerSeriesX {1}{10}{\coeffalt}{\the\cnta [-2]}\{5}}
{5}}\endgraf
\ifnum\cnta < 49 \advance\cnta 7 \repeat
\end{multicols}
D(0/100): 0/1[0]
D(7/100): 2/1[-5]
D(14/100): 2/1[-5]
D(21/100): 3/1[-5]
D(28/100): 4/1[-5]
D(35/100): 4/1[-5]
D(42/100): 9/1[-5]
D(49/100): 42/1[-5]

```

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

```

\begin{multicols}2
\loop
\noindent \hbox to 2.5cm {\hss\texttt{D(\the\cnta/100): }}%
\dt{\xintRound{4}
{\xintAdd {\xintFxFtPowerSeriesX {1}{15}{\coefflog}{\the\cnta [-2]}\{6}}
{\xintFxFtPowerSeriesX {1}{15}{\coefflog}
{\xintRound {4}{\xintFxFtPowerSeriesX {1}{15}{\coeffalt}
{\the\cnta [-2]}\{6}}}}
{6}}}%
}\endgraf
\ifnum\cnta < 49 \advance\cnta 7 \repeat
\end{multicols}
D(0/100): 0
D(7/100): 0.0000
D(14/100): 0.0000
D(21/100): -0.0001
D(28/100): -0.0001
D(35/100): -0.0001
D(42/100): -0.0000
D(49/100): -0.0001

```

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

14.9 `\xintFloatPowerSeries`

`\xintFloatPowerSeries[P]{A}{B}{\coeff}{f}` computes $\sum_{n=A}^{n=B} \text{coeff}\{n\} \cdot f^n$ with a floating point precision given by the optional parameter `P` or by the current setting of `\xintDigits`.

In the current, preliminary, version, no attempt has been made to try to guarantee to the final result the precision `P`. Rather, `P` is used for all intermediate floating point evaluations. So rounding errors will make some of the last printed digits invalid. The operations done are first the evaluation of f^A using `\xintFloatPow`, then each successive power is obtained from this first one by multiplication by `f` using `\xintFloatMul`, then again with `\xintFloatMul` this is multiplied with `\coeff{n}`, and the sum is done adding one term at a time with `\xintFloatAdd`. To sum up, this is just the naive transformation of `\xintFxFtPowerSeries` from fixed point to floating point.

```

\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}%
\xintFloatPowerSeries [8]{1}{30}{\coefflog}{-1/2[0]}

```

-6.9314718e-1

14.10 `\xintFloatPowerSeriesX`

`\xintFloatPowerSeriesX[P]{A}{B}{\coeff}{f}` is like `\xintFloatPowerSeries` with the difference that `f` is expanded once and for all at the start of the computation, thus allowing efficient chaining of such series evaluations.

```
\def\coeffexp #1{1/\xintFac {#1}[0]}% 1/n! (exact, not float)
\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}%
\xintFloatPowerSeriesX [8]{0}{30}{\coeffexp}
{\xintFloatPowerSeries [8]{1}{30}{\coefflog}{-1/2[0]}}
5.0000001e-1
```

14.11 Computing $\log 2$ and π

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

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

```

{\xintMul {2}{\xintFxFtPowerSeries {1}{#2}{\coefflog}{\xa}{#3}}}
{\xintMul {5}{\xintFxFtPowerSeries {1}{#1}{\coefflog}{\xb}{#3}}}%
}%
}%
\noindent $\log 2 \approx \text{LogTwo } \{60\} \text{ dots} \endgraf
\noindent \phantom{\$ \log 2} \{ \} \approx \{ \} \text{ printnumber} \{ \text{LogTwo } \{65\} \} \text{ dots} \endgraf
\noindent \phantom{\$ \log 2} \{ \} \approx \{ \} \text{ printnumber} \{ \text{LogTwo } \{70\} \} \text{ dots} \endgraf
\log 2 \approx 0.693147180559945309417232121458176568075500134360255254120484...
\approx 0.69314718055994530941723212145817656807550013436025525412068000711...
\approx 0.6931471805599453094172321214581765680755001343602552541206800094933723...

```

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

```

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

```

Let us turn now to π , computed with the Machin formula. Again the numbers of terms to keep in the two `arctg` series were roughly estimated, and some experimentations showed that removing the last three digits was enough (at least for $D=0-100$ range). And the algorithm does print the correct digits when used with $D=1000$ (to be convinced of that one needs to run it for $D=1000$ and again, say for $D=1010$.) A theoretical analysis could help confirm that this algorithm always gets better than 10^{-D} precision, but again, strings of 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\else 1\fi\relax/%
  \the\numexpr 2*#1+1\relax [0]}%
%\def\coeffarctg #1{\romannumeral0\xintmon{#1}/\the\numexpr 2*#1+1\relax}%
\def\xa {1/25[0]}% 1/5^2, the [0] for faster parsing
\def\xb {1/57121[0]}% 1/239^2, the [0] for faster parsing
\def\Machin #1{% #1 may be a count register, \Machin {\mycount} is allowed
  \romannumeral0\expandafter\MachinA \expandafter
  % number of terms for arctg(1/5):
  {\the\numexpr (#1+3)*5/7\expandafter}\expandafter
  % number of terms for arctg(1/239):
  {\the\numexpr (#1+3)*10/45\expandafter}\expandafter
  % do the computations with 3 additional digits:
  {\the\numexpr #1+3\expandafter}\expandafter
  % allow #1 to be a count register:
  {\the\numexpr #1\relax}}%
\def\MachinA #1#2#3#4%
{\xinttrunc {#4}
{\xintSub

```

```

{\xintMul {16/5}{\xintFxFtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}}
{\xintMul{4/239}{\xintFxFtPowerSeries {0}{#2}{\coeffarctg}{\xb}{#3}}}%
}}%
\begin{framed}
\[\pi = \text{Machin }{60}\text{dots }\]
\end{framed}

```

$$\pi = 3.141592653589793238462643383279502884197169399375105820974944\dots$$

Here is a variant `\MachinBis`, which evaluates the partial sums exactly using `\xintPowerSeries`, before their final truncation. No need for a ‘+3’ then.

```

\def\MachinBis #1{% #1 may be a count register,
% the final result will be truncated to #1 digits post decimal point
\romannumeral0\expandafter\MachinBisA \expandafter
% number of terms for arctg(1/5):
{\the\numexpr #1*5/7\expandafter}\expandafter
% number of terms for arctg(1/239):
{\the\numexpr #1*10/45\expandafter}\expandafter
% allow #1 to be a count register:
{\the\numexpr #1\relax }}%
\def\MachinBisA #1#2#3%
{\xinttrunc {#3} %
{\xintSub
{\xintMul {16/5}{\xintPowerSeries {0}{#1}{\coeffarctg}{\xa}}}
{\xintMul{4/239}{\xintPowerSeries {0}{#2}{\coeffarctg}{\xb}}}%
}}%

```

Let us use this variant for a loop showing the build-up of digits:

```

\begin{multicols}{2}
\cnta 0 % previously declared \count register
\loop \noindent
\centeredline{\dt{\MachinBis{\cnta}}}%
\ifnum\cnta < 30
\advance\cnta 1 \repeat
\end{multicols}

```

	3.141592653589793
3.	3.1415926535897932
3.1	3.14159265358979323
3.14	3.141592653589793238
3.141	3.1415926535897932384
3.1415	3.14159265358979323846
3.14159	3.141592653589793238462
3.141592	3.1415926535897932384626
3.1415926	3.14159265358979323846264
3.14159265	3.141592653589793238462643
3.141592653	3.1415926535897932384626433
3.1415926535	3.14159265358979323846264338
3.14159265358	3.141592653589793238462643383
3.141592653589	3.1415926535897932384626433832
3.1415926535897	3.14159265358979323846264338327
3.14159265358979	3.141592653589793238462643383279

You want more digits and have some time? compile this copy of the `\Machin` with `etex` (or `pdftex`):

```

% Compile with e-TeX extensions enabled (etex, pdftex, ...)
\input xintfrac.sty
\input xintseries.sty

```

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

\def\xa {1/25[0]}%
\def\xb {1/57121[0]}%
\def\Machin #1{%
  \romannumeral0\expandafter\MachinA \expandafter
  {\the\numexpr (#1+3)*5/7\expandafter}\expandafter
  {\the\numexpr (#1+3)*10/45\expandafter}\expandafter
  {\the\numexpr #1+3\expandafter}\expandafter
  {\the\numexpr #1\relax }}%
\def\MachinA #1#2#3#4%
{\xinttrunc {#4}
{\xintSub
{\xintMul {16/5}{\xintFxFtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}}
{\xintMul {4/239}{\xintFxFtPowerSeries {0}{#2}{\coeffarctg}{\xb}{#3}}}%
}}%
\pdfresettimer
\oodef\Z {\Machin {1000}}
\oodef\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 16 seconds last time I tried.⁶⁷ As mentioned in the introduction, the file `pi.tex` by D. ROEGEL shows that orders of magnitude faster computations are possible within \TeX , but recall our constraints of complete expandability and be merciful, please.

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

But, for the computation of a single constant, we are really interested in the exact decimal expansion, so we truncate and compute more terms until the earlier result gets validated. Finally if we do want the rounding we can always do it on a value computed with $D+1$ truncation.

15 Commands of the *xintcf* package

.1	Package overview.....	126	.5	<code>\xintGctoGCx</code>	132
.2	<code>\xintCFrac</code>	131	.6	<code>\xintFtoC</code>	132
.3	<code>\xintGCFrac</code>	131	.7	<code>\xintFtoCs</code>	132
.4	<code>\xintGGCFrac</code>	131	.8	<code>\xintFtoCx</code>	132

⁶⁷ With 1.09i and earlier *xint* releases, this used to be 42 seconds; the 1.09j division is much faster with small denominators as occurs here with `\xa=1/25`, and I believe this to be the main explanation for the speed gain.

.9	<code>\xintFtoGC</code>	133	.20	<code>\xintCntoF</code>	136
.10	<code>\xintFGtoC</code>	133	.21	<code>\xintGCntoF</code>	136
.11	<code>\xintFtoCC</code>	133	.22	<code>\xintCntoCs</code>	136
.12	<code>\xintCstoF</code>	133	.23	<code>\xintCntoGC</code>	137
.13	<code>\xintCtoF</code>	134	.24	<code>\xintGCntoGC</code>	137
.14	<code>\xintGctoF</code>	134	.25	<code>\xintCstoGC</code>	137
.15	<code>\xintCstoCv</code>	135	.26	<code>\xintiCstoF, \xintiGctoF, \xintiCstoCv,</code> <code>\xintiGctoCv</code>	138
.16	<code>\xintCtoCv</code>	135	.27	<code>\xintGctoGC</code>	138
.17	<code>\xintGctoCv</code>	135	.28	Euler's number e	138
.18	<code>\xintFtoCv</code>	135			
.19	<code>\xintFtoCCv</code>	136			

This package was first 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 brings some new macros: `\xintFtoC`, `\xintCtoF`, `\xintCtoCv`, dealing with sequences of braced partial quotients rather than comma separated ones, `\xintFGtoC` which is to produce “guaranteed” coefficients of some real number known approximately, and `\xintGGCFrac` for displaying arbitrary material as a continued fraction; also, some changes to existing macros: `\xintFtoCs` and `\xintCntoCs` insert spaces after the commas, `\xintCstoF` and `\xintCstoCv` authorize spaces in the input also before the commas.

This section contains:

1. an [overview](#) of the package functionalities,
2. a description of each one of the package macros,
3. further illustration of their use via the study of the [convergents of \$e\$](#) .

15.1 Package overview

The package computes partial quotients and convergents of a fraction, or conversely start from coefficients and obtain the corresponding fraction; three macros `\xintCFrac`, `\xintGCFrac` and `\xintGGCFrac` are for typesetting (the first two assume that the coefficients are numeric quantities acceptable by the *xintfrac* `\xintFrac` macro, the last one will display arbitrary material), the others can be nested (if applicable) or see their outputs further processed by other macros from the *xint* bundle, particularly the macros of *xinttools* dealing with sequences of braced items or comma separated lists.

A *simple* continued fraction has coefficients $[c_0, c_1, \dots, c_N]$ (usually called partial quotients, but I dislike this entrenched terminology), where c_0 is a positive or negative integer and the others are positive integers.

Typesetting is usually done via the *amsmath* macro `\cfrac`:

```
\[ c_0 + \cfrac{1}{c_1 + \cfrac{1}{c_2 + \cfrac{1}{c_3 + \cfrac{1}{\ddots}}}} \]
```

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

Here is a concrete example:

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

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

But it is the command `\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.⁶⁸ The *centered* continued fraction is an example:

```
\[ \xintFrac {915286/188421}=\xintGCFrac {5+-1/7+1/39+-1/53+-1/13}
=\xintCFrac {915286/188421}\]
```

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

The command `\xintGCFrac`, contrarily to `\xintCFrac`, does not compute anything, it just typesets starting from a generalized continued fraction in inline format, which in this example was input literally. We also used `\xintCFrac` for comparison of the two types of continued fractions.

To let \TeX compute the centered continued fraction of *f* there is `\xintFtoCC`:

```
\[ \xintFrac {915286/188421}\to\xintFtoCC {915286/188421}\]
```

$$\frac{915286}{188421} \rightarrow 5 + -1/7 + 1/39 + -1/53 + -1/13$$

The package macros are expandable and may be nested (naturally `\xintCFrac` and `\xintGCFrac` must be at the top level, as they deal with typesetting).

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

$$5 - \frac{1}{7 + \frac{1}{39 - \frac{1}{53 - \frac{1}{13}}}}$$

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*-expandable.

```
\[ \xintFrac{\xintGctoF {1+-1/57+\xintPow {-3}{7}/\xintiQuo {132}{25}}
= \xintGCFrac {1+-1/57+\xintPow {-3}{7}/\xintiQuo {132}{25}}\]
```

⁶⁸ *xintcfrac* may be used with indeterminates, for basic conversions from one inline format to another, but not for actual computations. See `\xintGGCFrac`.

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

To compute the actual fraction one has `\xintGctoF`:

$$\frac{1907}{1902}$$

For non-numeric input there is `\xintGGCFrac`.

$$a_0 + \frac{b_0}{a_1 + \frac{b_1}{a_2 + \frac{b_2}{\ddots + \frac{b_{n-1}}{a_{n-1} + \frac{b_n}{a_n}}}}}$$

For regular continued fractions, there is a simpler comma separated format:

$$\frac{-7, 6, 19, 1, 33}{1084483/398959} \rightarrow \frac{-28077}{4108} = -7 + \frac{1}{6 + \frac{1}{19 + \frac{1}{1 + \frac{1}{33}}}}$$

The command `\xintFtoCs` produces from a fraction *f* the comma separated list of its coefficients.

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

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.

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

This allows under Plain \TeX with `amstex` to obtain the same effect as with \LaTeX with `amsmath` and `\xintCFrac`:

$$\frac{2721}{1001} = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{4 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{6 + \cfrac{1}{2}}}}}}}}$$

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

$$\frac{2721}{1001} = \xintFtoGC {2721/1001}$$

`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`:

$$\frac{2721}{1001} = \xintFtoCC {2721/1001}$$

`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 2.1) macro which we use in this document to print long numbers can also be useful on long continued fractions.

$$\frac{35037018906350720204351049}{244241737886197404558180}$$

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

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

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

```
\xintGCtoF {143+1/2+...+-1/6}=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 commands of *xintcfac* such as `\xintFtoCv`, `\xintFtoCCv`, and others which compute such convergents, return them as a list of braced items, with no separator (as does `\xintFtoC` for the partial quotients). Here is an example:

```
\[\xintFrac{915286/188421}\to
\xintListWithSep{,}{\xintApply\xintFrac{\xintFtoCv{915286/188421}}}\
\frac{915286}{188421} \rightarrow 4, 5, \frac{34}{7}, \frac{1297}{267}, \frac{1331}{274}, \frac{69178}{14241}, \frac{70509}{14515}, \frac{915286}{188421}
```

```
\[\xintFrac{915286/188421}\to
\xintListWithSep{,}{\xintApply\xintFrac{\xintFtoCCv{915286/188421}}}\
\frac{915286}{188421} \rightarrow 5, \frac{34}{7}, \frac{1331}{274}, \frac{70509}{14515}, \frac{915286}{188421}
```

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

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

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

Next, we use the following code:

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

It produces:

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

The macro `\xintCtoF` allows to specify the coefficients as a function given by a one-parameter macro. The produced values do not have to be integers.

```
\def\cn #1{\xintiiPow {2}{#1}}% 2^n
\[\xintFrac{\xintCtoF {6}{\cn}}=\xintCFrac [1]{\xintCtoF {6}{\cn}}\]
```

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

Notice the use of the optional argument [1] to `\xintCFrac`. Other possibilities are [r] and (default) [c].

```
\def\cn #1{\xintPow {2}{-#1}}%
```

$$\begin{aligned} & \backslash[\backslash\text{xintFrac}\{\backslash\text{xintCntoF } \{6\}\{\backslash\text{cn}\}\}=\backslash\text{xintGCFrac } [r]\{\backslash\text{xintCntoGC } \{6\}\{\backslash\text{cn}\}\}= \\ & \quad [\backslash\text{xintFtoCs } \{\backslash\text{xintCntoF } \{6\}\{\backslash\text{cn}\}\}\backslash] \\ & \frac{3159019}{2465449} = 1 + \frac{1}{\frac{1}{2} + \frac{1}{\frac{1}{4} + \frac{1}{\frac{1}{8} + \frac{1}{\frac{1}{16} + \frac{1}{\frac{1}{32} + \frac{1}{\frac{1}{64}}}}}}} = [1, 3, 1, 1, 4, 14, 1, 1, 1, 1, 79, 2, 1, 1, 2] \end{aligned}$$

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

$$\begin{aligned} & \backslash\text{def}\backslash\text{an } \#1\{\backslash\text{the}\backslash\text{numexpr } 2*\#1+1\backslash\text{relax } \}% \\ & \backslash\text{def}\backslash\text{bn } \#1\{\backslash\text{the}\backslash\text{numexpr } (\#1+1)*(\#1+1)\backslash\text{relax } \}% \\ & \backslash[\backslash\text{xintFrac}\{\backslash\text{xintDiv } \{4\}\{\backslash\text{xintGCntoF } \{5\}\{\backslash\text{an}\}\{\backslash\text{bn}\}\}\} = \\ & \quad \backslash\text{cfrac}\{4\}\{\backslash\text{xintGCFrac}\{\backslash\text{xintGCntoGC } \{5\}\{\backslash\text{an}\}\{\backslash\text{bn}\}\}\} = \\ & \quad \backslash\text{xintTrunc } \{10\}\{\backslash\text{xintDiv } \{4\}\{\backslash\text{xintGCntoF } \{5\}\{\backslash\text{an}\}\{\backslash\text{bn}\}\}\}\backslash\text{dots}\backslash] \\ & \frac{92736}{29520} = \frac{4}{1 + \frac{1}{3 + \frac{4}{5 + \frac{9}{7 + \frac{16}{9 + \frac{25}{11}}}}}} = 3.1414634146\dots \end{aligned}$$

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

$$\begin{aligned} & \backslash[\backslash\text{xintFrac}\{\backslash\text{xintCstoF}\{3,7,15,1,292,1,1\}\}= \\ & \quad \backslash\text{xintGCFrac}\{3+1/7+1/15+1/1+1/292+1/1+1/1\}= \\ & \quad \backslash\text{xintTrunc}\{10\}\{\backslash\text{xintCstoF}\{3,7,15,1,292,1,1\}\}\backslash\text{dots}\backslash] \\ & \frac{208341}{66317} = 3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1 + \frac{1}{292 + \frac{1}{1 + \frac{1}{1}}}}}} = 3.1415926534\dots \end{aligned}$$

When studying the continued fraction of some real number, there is always some doubt about how many terms are valid, when computed starting from some approximation. If $f \leq x \leq g$ and f, g both have the same first K partial quotients, then x also has the same first K quotients and convergents. The macro `\xintFGtoC` outputs as a sequence of braced items the common partial quotients of its two arguments. We can thus use it to produce a sure list of valid convergents of π for example, starting from some proven lower and upper bound:

```

 $\pi$ \to [\backslash\text{xintListWithSep}\{, \}
\{\backslash\text{xintFGtoC } \{3.14159265358979323\}\{3.14159265358979324\}\}, \backslash\text{dots}\}
\backslash\text{noindent}\pi\to\backslash\text{xintListWithSep}\{,\backslash\text{allowbreak}\};
\{\backslash\text{xintApply}\{\backslash\text{xintFrac}\}

```

```
{\xintCtoCv{\xintFGtoC {3.14159265358979323}{3.14159265358979324}}}, \dots$

$$\pi \rightarrow [3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, \dots]$$


$$\pi \rightarrow 3, \frac{22}{7}, \frac{333}{106}, \frac{355}{113}, \frac{103993}{33102}, \frac{104348}{33215}, \frac{208341}{66317}, \frac{312689}{99532}, \frac{833719}{265381}, \frac{1146408}{364913}, \frac{4272943}{1360120}, \frac{5419351}{1725033}, \frac{80143857}{25510582}, \frac{165707065}{52746197}, \frac{245850922}{78256779}, \frac{411557987}{131002976}, \dots$$

```

15.2 `\xintCFrac`

Frac
f

`\xintCFrac{f}` is a math-mode only, \TeX with `amsmath` only, macro which first computes then displays with the help of `\cfrac` the simple continued fraction corresponding to the given fraction. It admits an optional argument which may be `[l]`, `[r]` or (the default) `[c]` to specify the location of the one's in the numerators of the sub-fractions. Each coefficient is typeset using the `\xintFrac` macro from the `xintfrac` package. This macro is *f*-expandable in the sense that it prepares expandably the whole expression with the multiple `\cfrac`'s, but it is not completely expandable naturally as `\cfrac` isn't.

15.3 `\xintGCFrac`

f `\xintGCFrac{a+b/c+d/e+f/g+h/...+x/y}` uses similarly `\cfrac` to prepare the typesetting with the `amsmath` `\cfrac` (\TeX) of a generalized continued fraction given in inline format (or as macro which will *f*-expand to it). It admits the same optional argument as `\xintCFrac`. Plain \TeX with `amstex` users, see `\xintGctoGCx`.

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

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

This is mostly a typesetting macro, although it does provoke the expansion of the coefficients. See `\xintGctoF` if you are impatient to see this specific fraction computed.

It admits an optional argument within square brackets which may be either `[l]`, `[c]` or `[r]`. Default is `[c]` (numerators are centered).

Numerators and denominators are made arguments to the `\xintFrac` macro. This allows them to be themselves fractions or anything *f*-expandable giving numbers or fractions, but also means however that they can not be arbitrary material, they can not contain color changing commands for example. One of the reasons is that `\xintGCFrac` tries to determine the signs of the numerators and chooses accordingly to use + or -.

15.4 `\xintGGCFrac`

f `\xintGGCFrac{a+b/c+d/e+f/g+h/...+x/y}` is a clone of `\xintGCFrac`, hence again \TeX specific with package `amsmath`. It does not assume the coefficients to be numbers as understood by `xintfrac`. The macro can be used for displaying arbitrary content as a continued fraction with `\cfrac`, using only plus signs though. Note though that it will first *f*-expand its argument, which may be thus be one of the `xintfrac` macros producing a (general) continued fraction in inline format, see `\xintFtoCx` for an example. If this expansion is not wished, it is enough to start the argument with a space.

```
\[\xintGGCFrac {1+q/1+q^2/1+q^3/1+q^4/1+q^5/\ddots}\]
```

$$1 + \frac{q}{1 + \frac{q^2}{1 + \frac{q^3}{1 + \frac{q^4}{1 + \frac{q^5}{\ddots}}}}}$$

15.5 `\xintGtoGCx`

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

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

The following can be used by Plain \TeX +`amstex` users to obtain an output similar as the ones produced by `\xintGCFrac` and `\xintGGCFrac`:

```

 $\xintGtoGCx {+}{\cfrac}{\{\}\{a+b/\dots\}}{\endcfrac}$ 
 $\xintGtoGCx {+}{\cfrac{\xintFwOver}{\{\}\{\xintFwOver\}}}{a+b/\dots}{\endcfrac}$ 

```

15.6 `\xintFtoC`

`\xintFtoC{f}` computes the coefficients of the simple continued fraction of `f` and returns them as a list (sequence) of braced items.

```
\oodef\test{\xintFtoC{-5262046/89233}}\texttt{\meaning\test}
```

macro:->{-59}{33}{27}{100}

15.7 `\xintFtoCs`

`\xintFtoCs{f}` returns the comma separated list of the coefficients of the simple continued fraction of `f`. Notice that starting with 1.09m a space follows each comma (mainly for usage in text mode, as in math mode spaces are produced in the typeset output by \TeX itself).

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

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

15.8 `\xintFtoCx`

`\xintFtoCx{sep}{f}` returns the list of the coefficients of the simple continued fraction of `f` separated with the help of `sep`, which may be anything (and is kept unexpanded). For example, with Plain \TeX and `amstex`,

```
 $\xintFtoCx {+}{\cfrac1{\{\}\{-5262046/89233\}}}{\endcfrac}$ 
```

will display the continued fraction using `\cfrac`. Each coefficient is inside a brace pair `{ }`, allowing a macro to end the separator and fetch it as argument, for example, again with Plain \TeX and `amstex`:

```

\def\highlight #1{\ifnum #1>200 \textcolor{red}{#1}\else #1\fi}
 $\xintFtoCx {+}{\cfrac1{\{\}\{\highlight{104348/33215}\}}}{\endcfrac}$ 

```

Due to the different and extremely cumbersome syntax of `\cfrac` under \TeX it proves a bit tortuous to obtain there the same effect. Actually, it is partly for this purpose that 1.09m added `\xintGGCFrac`. We thus use `\xintFtoCx` with a suitable separator, and then the whole thing as argument to `\xintGGCFrac`:

```

\def\highlight #1{\ifnum #1>200 \fcolorbox{blue}{white}{\boldmath\color{red}$#1$}%
\else #1\fi}
\[ \xintGGCFrac {\xintFtoCx {+1/\highlight}{208341/66317}}\]

```

$$3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1 + \frac{1}{\boxed{292} + \frac{1}{2}}}}}$$

15.9 \xintFtoGC

`\frac{f}{f}` ★ `\xintFtoGC{f}` does the same as `\xintFtoCx{+1/}{f}`. Its output may thus be used in the package macros expecting such an ‘inline format’.

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

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

15.10 \xintFGtoC

`\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`.

`\oodef\test{\xintFGtoC{-5262046/89233}{-5314647/90125}}\texttt{\meaning\test}`

macro: $\rightarrow \{-59\}\{33\}\{27\}$

$$\backslash\mathrm{oodef}\backslash\mathrm{test}\{\backslash\mathrm{xintFGtoC}\{3.141592653\}\{3.141592654\}\}\backslash\mathrm{texttt}\{\backslash\mathrm{meaning}\backslash\mathrm{test}\}$$

```
macro: ->{3}{7}{15}{1}
```

```
\oodef\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

```
\oodef\test{\xintFGtoC{1.41421356237309}{1.4142135623731}}\meaning\test
```

```
macro: ->{1}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}
```

15.11 \xintFtoCC

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

566827/208524=\xintFtoCC {566827/208524}

$$566827/208524=3+-1/4+-1/2+1/5+-1/2+1/7+-1/2+1/9+-1/2+1/11$$
$$\left[\frac{566827}{208524}\right] = \text{GCfrac}\{\text{FtoCC}\{566827/208524\}\}$$

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

15.12 \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. Starting with release 1.09m spaces before commas are allowed and trimmed automatically (spaces after commas were already silently handled in earlier releases).

$$\backslash[\backslash xintGCFrac \{-1+1/3+1/-5+1/7+1/-9+1/11+1/-13\}=$$
$$\text{\xintSignedFrac}\{\text{\xintCstoF}\{-1,3,-5,7,-9,11,-13\}\}=\text{\xintSignedFrac}\{\text{\xintGctoF}\{-1,3,-5,7,-9,11,-13\}\}$$

```
{-1+1/3+1/-5+1/7+1/-9+1/11+1/-13}}\]
```

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

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

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

A generalized continued fraction may produce a reducible fraction (`\xintCstoF` tries its best not to accumulate in a silly way superfluous factors but will not do simplifications which would be obvious to a human, like simplification by 3 in the result above).

15.13 `\xintCtoF`

f ★ `\xintCtoF{a}{b}{c}...{z}` computes the fraction corresponding to the coefficients, which may be fractions or even macros.

New with 1.09m `\xintCtoF {\xintApply {\xintiiPow 3}{\xintSeq {1}{5}}}`

```
14946960/4805083
```

```
\[ \xintFrac{14946960/4805083}=\xintCFrac {14946960/4805083}\]
```

$$\frac{14946960}{4805083} = 3 + \frac{1}{9 + \frac{1}{27 + \frac{1}{81 + \frac{1}{243}}}}$$

In the example above the power of 3 was already pre-computed via the expansion done by `\xintApply`, but if we try with `\xintApply { \xintiiPow 3}` where the space will stop this expansion, we can check that `\xintCtoF` will itself provoke the needed coefficient expansion.

15.14 `\xintGtoF`

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

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

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

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

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

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.

15.15 `\xintCstoCv`

f ★ `\xintCstoCv{a,b,c,d,...,z}` returns the sequence of the corresponding convergents, each one within braces.

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
\[\xintListWithSep{\to}{\xintApply\xintFrac{\xintCstoCv {\xintPow
{-.3}{-5},7.3/4.57,\xintCstoF{3/4,9,-1/3}}}}\]
      -100000      -72888949      -2700356878
      243          177390          6567804
```

15.16 `\xintCtoCv`

f ★ `\xintCtoCv{{a}{b}{c}...{z}}` returns the sequence of the corresponding convergents, each one within braces.

New with 1.09m `\oodef\test{\xintCtoCv {1111111111}}\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}`

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

15.18 `\xintFtoCv`

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

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

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

15.19 `\xintFtoCCv`

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

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

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

15.20 `\xintCntoF`

$\frac{\text{num}}{x}$ ★ `\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).

15.21 `\xintGCntoF`

$\frac{\text{num}}{x}$ ★ `\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}} =
```

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

There is also `\xintGCntoGC` to get the ‘inline format’ continued fraction.

15.22 `\xintCntoCs`

$\frac{\text{num}}{x}$ ★ `\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

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

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

15.23 `\xintCtoGC`

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

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

```
macro:->{1/\the \numexpr 1+0*0\relax }+1/{-2/\the \numexpr 1+1*1\relax }+1/{3/\the \numexpr
1+2*2\relax }+1/{-4/\the \numexpr 1+3*3\relax }+1/{5/\the \numexpr 1+4*4\relax }+1/{-6/\the
\numexpr 1+5*5\relax }
```

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

15.24 `\xintGCtoGC`

`\xintGCtoGC{N}{\macroA}{\macroB}` evaluates the coefficients and then returns the corresponding $\{a_0\}+\{b_0\}/\{a_1\}+\{b_1\}/\{a_2\}+\dots+\{b(N-1)\}/\{a_N\}$ inline generalized fraction. N is given to a `\numexpr`. 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}%
$\xintGCtoGC {5}{\an}{\bn}=\xintGCFrac {\xintGCtoGC {5}{\an}{\bn}} =
\displaystyle\xintFrac {\xintGCtoF {5}{\an}{\bn}}$\par
```

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

15.25 `\xintCstoGC`

`\xintCstoGC{a,b,...,z}` transforms a comma separated list (or something expanding to such a list) into an ‘inline format’ continued fraction $\{a\}+1/\{b\}+1/\dots+1/\{z\}$. The coefficients are just

copied and put within braces, without expansion. The output can then be used in `\xintGCFrac` for example.

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

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

15.26 `\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.

15.27 `\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 withing braces.

```
\oodef\x {\xintGctoGC {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/%
\xintFac {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.

15.28 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\dt{the\cnta.} }%
$\xintTrunc {30}{\xintAdd {1[0]}\{#1}\dots=}
\xintFrac{\xintAdd {1[0]}\{#1}\}$}%
\xintListWithSep{\vtop to 6pt}{\vbox to 12pt}{\par}
{\xintApply\mymacro{\xintiCstoCv{\xintCntoCs {35}\{cn}}}}
```

- [illegible]

$$34. 2.718281828459045235360287471355\dots = \frac{1085079390005041}{399178399621704}$$

$$35. 2.718281828459045235360287471352\dots = \frac{2124008553358849}{781379079653017}$$

$$36. 2.718281828459045235360287471352\dots = \frac{52061284670617417}{19152276311294112}$$

One can with no problem compute much bigger convergents. Let's get the 200th convergent. It turns out to have the same first 268 digits after the decimal point as $e-1$. Higher convergents get more and more digits in proportion to their index: the 500th convergent already gets 799 digits correct! To allow speedy compilation of the source of this document when the need arises, I limit here to the 200th convergent.

```
\oodef\z {\xintCnToF {199}{\cn}}%
\begin{group}\par\indent 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\end{group}
Numerator = 568964038871896267597523892315807875293889017667917446057232024547192296961118
23017524386017499531081773136701241708609749634329382906
Denominator = 331123817669737619306256360816356753365468823729314438156205615463246659728581
86546133769206314891601955061457059255337661142645217223
Expansion = 1.7182818284590452353602874713526624977572470936999595749669676277240766303535
475945713821785251664274274663919320030599218174135966290435729003342952605956
307381323286279434907632338298807531952510190115738341879307021540891499348841
675092447614606680822648001684774118...
```

One can also use a centered continued fraction: we get more digits but there are also more computations as the numerators may be either 1 or -1.

This documentation has been compiled without the source code, which is available in the separate file:

[sourcexint.pdf](#),

which should be among the candidates proposed by `texdoc --list xint`. To produce a single file including both the user documentation and the source code, run `tex xint.dtx` to generate `xint.tex` (if not already available), then edit `xint.tex` to set the `\NoSourceCode` toggle to 0, then run thrice `latex` on `xint.tex` and finally `dvipdfmx` on `xint.dvi`. Alternatively, run `pdflatex` either directly on `xint.dtx`, or on `xint.tex` with `\NoSourceCode` set to 0.