XLOP v 0.25

User Manual

Jean-Côme Charpentier
February 26, 2013

Contents

Chapter 1

Overview

The xlop package is intended to make automatic arithmetic operation on arbitrary sized numbers and to display result either on display mode or in-line mode. Here is a first exemple for an overview of the syntax:

$$\begin{array}{rr} & {\scriptstyle 1} \\ & 4\ 5.0\ 5 \\ + & 7\ 8.4 \\ \hline & 1\ 2\ 3.4\ 5 \end{array}$$

source

```
\opadd{45,05}{78,4}
```

We comment this first example in order to give an idea about how use xlop. Addition is displayed "like in school": this is the default displaying. We have an alignment on dots (operand's dot and result's dot), operator symbol is put on the left and it is vertically centered between the operands, and the decimal separator is a dot even though we have specified operands with comma. Finally, note that there is a carry above the first operand.

Alignment on dot is obligatory. The other points above are deal with options. Many macros accept an optional argument which controls some aspects of displaying or computing operation. For that, we use a "keyval-like" syntax: we specify a sequence of parameter's modifications through an affectation's comma separated sequence. One affectation has one of the two possible syntax below:

```
<parameter>=<value>
<parameter>
```

the second one is a shorthand for:

```
<parameter>=true
```

In this affectation sequence, we can put space(s) after comma. But don't put space around the equal sign nor before comma: if you put space(s) here, that means that parameter name or value has a space.

So, if you want a comma as decimal separator , an operator symbol side by side with the second operand, and no carry, you have just to say:

```
    4 5,0 5
+   7 8,4
  1 2 3,4 5
```

source
```
\opadd[decimalsepsymbol={,},
  voperator=bottom,
  carryadd=false]{45.05}{78.4}
```

Note the trick which consists to put the comma between braces in the decimal separator symbol definition. In fact, if you say:

source
```
\opadd[decimalsepsymbol=,,voperator=bottom,
       carryadd=false]{45.05}{78.4}
```

xlop drives mad! It don't understand what is this sort of list!

Another important point, though it is less apparent, is that the figures are put in very precise places. Each figure is put in a box of fixed width and fixed height (user can change these values), decimal separator is put in a null-width box (by default), and the lines have a regular interspace (with or without horizontal rule). This allows exact spacing and to place what we want where we want.

```
    1 <------- carry
    4 5.0 5
+
    7 8.4
  1 2 3.4 5
```

source
```
\psset{xunit=\opcolumnwidth,
       yunit=\oplineheight}
\opadd{45.05}{78.4}
\oplput(1.5,3){carry}
\psline{->}(1,3.15)(-3.25,3.15)
```

This example uses package pstricks

We have said that xlop package is able to deal with arbitrary sized numbers. We come again about this subject and, for now, we just give an example which shows what is possible. Don't look at the code, some explanations will be given later in this manual, for now just admire the result!

source
```
\opdiv[style=text,period]{1}{49}
```

$$1 \div 49 = 0.\underline{0204081632653061224489795918367346938775 51}\ldots$$

The package xlop provides some other features. It is possible to manipulate numbers through variables. These variables can be created with an assignation or as a computation result. You can also manipulate the figures individually:

source
```
\opadd*{45.05}{78.4}{r}%
The first figure after dot of
$45.05+78.4$ is
\opgetdecimaldigit{r}{1}{d}%
$\opprint{d}$.
```

The first figure after dot of $45.05 + 78.4$ is 4.

you can make tests:

The sum 45.05 + 78.4 is greater than 100.

source

```
\opadd*{45.05}{78.4}{r}%
The sum $45.05+78.4$ is
\opcmp{r}{100}%
\ifopgt greater than
\else\ifoplt less than
\else equal to
\fi\fi
$100$.
```

you can use some operations and some functions:

gcd of 182 and 442 is 26

source

```
gcd of $182$ and $442$ is
\opgcd{182}{442}{d}$\opprint{d}$
```

you can compute complex expression in infix form:

$$\frac{2+3^2}{\gcd(22,33)} = 1$$

source

```
\opexpr{(2+3^2)/(gcd(22,33))}{r}%
$$\frac{2+3^2}{\gcd(22,33)} =
  \opprint{r}$$
```

Chapter 2

xlop Instructions

Except some macros which will be examined later, the xlop's macros can have an optional argument between squared braces in order to localy modify parameter's values. The other arguments (mandatory) are (nearly) always numbers. The two sections of this chapter describe in details what is a number for xlop and how use parameters.

2.1 In the Beginning Was the Number

2.1.1 Size

Before we see the general syntax of number, we examin the very particular xlop feature: the ability to deal with arbitrary sized number.

To be precise, the theoric maximum size of a number is $2^{31}-1$ digits. In practice, this limit can't be reached for two essential reasons. The first one is that a multiplication with two numbers with 2^{25} digits needs more than 7 000 years to be performed on the author computer! The second one is more restrictive because it is linked to TEX stack size limits. Here is a table showing a TEX compilation for a multiplication with two operands of same size, on a linux computer, pentium II 600 and 256 Mb RAM:[1]

number of digits	100	200	300	400	425	450
compilation time (s)	2	8	18	32	36	crash

The "crash" in the table is due to an overstack for hash table. On LATEX, the limit before crash will be reduced. These tests are made on a minimal file. With a typical document, this limit will be reduced too. The spool size is another limit quickly reached. To typeset this document which contain many calls to the xlop macros, the author has grown up the spool size to 250000 (125 000 was insufficient) editing the line `pool_size` in the `texmf.cnf` file. Also, the author has grown up the hash table to 1000 in the line `hash_extra`.

[1] In fact it was the author computer in 2004. The actual author computer is *more* powerfull but the author is lazy, and he has not remake the tests!

2.1.2 Syntax

Now we present the syntax using the BNF grammar. There will be human explanations later:

```
    ⟨number⟩ := {⟨sign⟩}⟨positive⟩ | ⟨name⟩
      ⟨sign⟩ := + | -
  ⟨positive⟩ := ⟨integer⟩ | ⟨sep⟩⟨integer⟩ |
                  ⟨integer⟩⟨sep⟩ | ⟨integer⟩⟨sep⟩⟨integer⟩
       ⟨sep⟩ := . | ,
   ⟨integer⟩ := ⟨digit⟩{⟨digit⟩}
      ⟨name⟩ := ⟨start⟩{character}
     ⟨start⟩ := character except ⟨sign⟩, ⟨sep⟩
                  , and ⟨digit⟩
```

The `character` symbol means nearly any character accepted by TEX. The exceptions are characters % and # which are completely prohibited. In fact, the use of active characters is risked. For instance, on LATEX, the ~ definition prohibits the use of it inside a variable name. In the other hand, the \ is always the escape char, that is, the variable name will be the name *after* all is expanded. There isn't any other restraint as the following code show it:

source

```
\newcommand\prefix{a/b}
\opadd*{2}{2}{a/b_{^c}!&$}
\opprint{\prefix_{^c}!&$}
```

4

Note particullary that `a/b_{^c}!&$` and `\prefix_{^c}!&$` produce exactly the same name... obviously if `\prefix` has the right definition! This possibility to have a name using macro could seem useless but it is not true. For instance, you can realize loops with names as `r1`, `r2`, ..., `r<n>` using the code `r\the\cpt` as variable name, where `cpt` is a counter in the TEX meaning. With LATEX, the code is more verbose with `r\number\value{cpt}` where `cpt` is now a LATEX counter. We will see an example using this syntax in the section B.2 page 41.

In practice, what does it mean all these rules? First, they means that a number writes in a decimal form can be preceded by any sequence of plus or minus signs. Obviously, if there is a odd number of minus signs, the number will be negative. Next, a decimal number admits only one decimal separator symbol which can be a dot or a comma, this one can be put anywhere in the number. Finally, a number is write in basis 10. Be carefull: these rules mean that `-a` is not valid.

The package uses some private names and it is safe to not begin a variable name with the character `@`.

2.2 xlop Parameters

Parameter assignments are local to the macro when they are indicated in the optional argument. To make global a parameter assignment, you have

to use the \opset macro. For example:

source
```
\opset{decimalsepsymbol={,}}
```

give the comma as decimal separator symbol for the whole document, at least, until another redefinition with \opset.

2.2.1 Symbols

The afterperiodsymbol parameter indicates the symbol that follows a quotient in line in a division with period search. Its default value is $\ldots$

The equalsymbol parameter indicates the symbol used for equality. Its default value is $=$. In fact, this parameter is defined with:

source
```
\opset{equalsymbol={$=$}}
```

that is, with braces in order to protect the equal sign. Without theses braces, there will be a compilation error. You have to process like that when there is an equal sign or a comma in the value.

The parameter approxsymbol indicates the symbol used for approximations. Its default value is $\approx$.

The parameter decimalsepsymbol indicates the symbol used for the decimal separator. Its default value is a dot.

Parameters addsymbol, subsymbol, mulsymbol, and divsymbol indicate the symbols used for the four arithmetic operations. The default value are $+$, $-$, $\times$ et $\div$ respectively.

2.2.2 General Displaying

The voperation parameter indicates the way a dispayed operation is put with respect to the baseline. The possible values are top, center, and bottom, the latter one is the default value.

top
$$\begin{array}{r} {\scriptstyle 1} \\ 45 \\ +\ 172 \\ \hline 217 \end{array}$$

center
$$\begin{array}{r} {\scriptstyle 1} \\ 45 \\ +\ 172 \\ \hline 217 \end{array}$$

bottom
$$\begin{array}{r} {\scriptstyle 1} \\ 45 \\ +\ 172 \\ \hline 217 \end{array}$$

source
```
top\quad
\opadd[voperation=top]{45}{172}\par
center\quad
\opadd[voperation=center]{45}{172}\par
bottom\quad
\opadd[voperation=bottom]{45}{172}
```

The voperator parameter indicates how the operator symbol is put with repect to operands. The possible values are top, center (default value), and bottom.

top
$$\begin{array}{r} {\scriptstyle 1} \\ +\ 45 \\ 172 \\ \hline 217 \end{array}$$

center
$$\begin{array}{r} {\scriptstyle 1} \\ +\ 45 \\ 172 \\ \hline 217 \end{array}$$

bottom
$$\begin{array}{r} {\scriptstyle 1} \\ +\ 45 \\ 172 \\ \hline 217 \end{array}$$

source
```
top\quad
\opadd[voperator=top]{45}{172}\par
center\quad
\opadd[voperator=center]{45}{172}\par
bottom\quad
\opadd[voperator=bottom]{45}{172}
```

The `deletezero` parameter indicates if some numbers in operation should be displayed with or whithout non-significant zeros. Exact rôle of this parameter depends of the actual operation. We will see that when we will study the different operations.

The `style` parameter indicates the way an operation is displayed: display with `display` value (default value) or inline with `text` value. We will see when we will study division because there is many possibilities with this operation.

$45 + 172 = 217$

source
```
\opadd[style=text]{45}{172}
```

In inline operations, xlop takes care to not typeset the formula in mathematic mode in a direct way. This allow to specify what you want as in the next example, and it is also for that that you have to specify the classical values of symbols between mathematic delimiters.

42 plus 172 equal 214

source
```
\opadd[addsymbol=plus,
       equalsymbol=equal,
       style=text]{42}{172}
```

Meanwhile, xlop introduces exactly the same penalities and the same spaces as for a mathematic formula.

The `parenthesisnegative` parameter indicates how to typeset negative numbers in inline operations. The possible values are:

- none which typesets negative numbers without parenthesis;
- `all` which typesets negative numbers with parenthesis;
- `last` which typesets negative numbers with parenthesis but the first one.

$-12 + -23 = -35$
$(-12) + (-23) = (-35)$
$-12 + (-23) = -35$

source

```
\opadd[style=text,
       parenthesisnegative=none]
       {-12}{-23}\par
\opadd[style=text,
       parenthesisnegative=all]
       {-12}{-23}\par
\opadd[style=text,
       parenthesisnegative=last]
       {-12}{-23}
```

2.2.3 Dimensions

In displayed operations, figures are put in fixed size boxes. The width is given by the `lineheight` parameter and the height is given by the `lineheight` parameter. The default value of `lineheight` is `\baselineskip` that is, interline space in operation is the same (by default) as in the normal text. The default value for `columnwidth` is `2ex` because the "normal" width of figures would give bad results.

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 45{,}89 \\ +\,127{,}5 \\ \hline 173{,}39 \end{array}$$

source

```
\opadd[columnwidth=0.5em]
      {45.89}{127.5}
```

One reason for this bad result is that the decimal separator is put in a box which width is controlled by the `decimalsepwidth` parameter and the default value of this parameter is null. You can improve this presentation giving a "normal" width to the dot.

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 45.89 \\ +\,127.5 \\ \hline 173.39 \end{array}$$

source

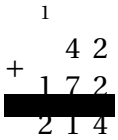
```
\opadd[columnwidth=0.5em,
       decimalsepwidth=0.27778em]
      {45.89}{127.5}
```

It is better but give a positive width to the box that contain the decimal separator is risked. It will be more difficult to place extern object and it is counter against the idea to have a fixed grid. You should avoid this in normal time.

The `columnwidth` and `lineheight` parameters correspond to the only dimensions that xlop provides as public one, that is, `\opcolumnwidth` and `\oplineheight` respectively. It is dangerous to directly modify these dimensions since a modification in a "normal" way doesn't only change the dimension value. Package xlop make these dimensions public only for reading, not for writting.

The two next parameters allow to specify width of horizontal and vertical rules stroked by xlop. We have `hrulewidth` and `vrulewidth` parameters. The default values are both `0.4pt`.

These rules are typeset with no change on grid. That is, with no space added. Therefore, with great values for thickness, the rules could run over numbers.



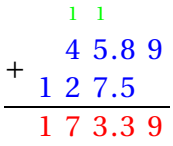
source
```
\opadd[hrulewidth=8pt]{42}{172}
```

There is also a paramater which allows to control the horizontal shift of decimal separator. It is the `decimalsepoffset` paramater with a default value of `-0.35`. This value indicates a length with the unit `\opcolumnwidth`. We will see an example at section 3.4 page 17.

2.2.4 Figure's Styles

The xlop package provides five types of numbers and associates five style paramaters:

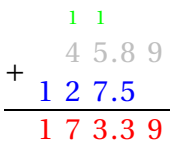
- operands with `operandstyle`;
- result with `resultstyle`;
- remainders with `remainderstyle`;
- intermediary numbers with `intermediarystyle`;
- carries with `carrystyle`.



source
```
\opadd[operandstyle=\blue,
       resultstyle=\red,
       carrystyle=\scriptsize\green]
       {45.89}{127.5}
```

Keep in mind that, in this manual, we use pstricks package.

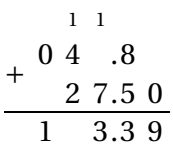
In fact, the management of these styles is even more powerfull since you can distingish different number of a same class. In one operation, you have several operands, and, possibly several remainders and several intermediary numbers. You can access to the style of these numbers adding an index to the matching style.



source
```
\opadd[operandstyle=\blue,
       operandstyle.1=\lightgray,
       resultstyle=\red,
       carrystyle=\scriptsize\green]
       {45.89}{127.5}
```

In this example, we indicate that the first operand must be typesetted with the `\lightgray` style. We don't indicate anything for the second operand, so it takes the basic style for its class. (Then with `\blue` style.)

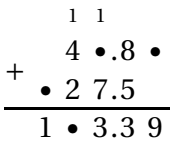
This mechanism is even more powerfull since you can write two level index for operands, carries, and intermediary numbers (one level for result and carry) in order to access to each style figure of these numbers. To simplify index, a positive index indicates the rank of a figure in the integer part (right to left order, index 1 is for the unit figure) and a negative index indicates the rank of a figure in the decimal part (left to right order, -1 is for the tenth figure).



source

```
\opadd[operandstyle.1.1=\white,
       operandstyle.1.-2=\white,
       operandstyle.2.3=\white,
       resultstyle.2=\white,
       deletezero=false]
       {045.89}{127.50}
```

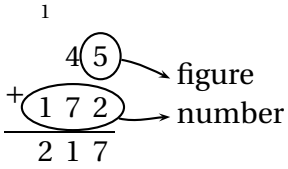
You can also use a macro with one parameter as a style.



source

```
\newcommand\hole[1]{$\bullet$}
\opadd[operandstyle.1.1=\hole,
       operandstyle.1.-2=\hole,
       operandstyle.2.3=\hole,
       resultstyle.2=\hole]
       {45.89}{127.5}
```

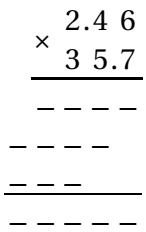
When the style is a macro with argument, this one is the figure. Here is a more complicated example using pst-node package of the pstricks bundle:



source

```
\newcommand\OPoval[3]{%
  \dimen1=#2\opcolumnwidth
  \ovalnode{#1}
    {\kern\dimen1 #3\kern\dimen1}}
\opadd[voperation=top,
  operandstyle.1.1=\OPoval{A}{0},
  operandstyle.2.2=\OPoval{C}{0.8}]
  {45}{172}\qquad
\begin{minipage}[t]{2cm}
  \pnode(0,0.2em){B}\ figure
  \ncarc{->}{A}{B}\par
  \pnode(0,0.2em){D}\ number
  \ncarc{<-}{D}{C}
\end{minipage}
```

As for figures, the decimal separator take account to number style. To access individually to the decimal separator style, you have to use `d` index, numeric indexes are for figures.



source

```
\newcommand\hole[1]{\texttt{\_}}
\opmul[intermediarystyle=\hole,
  resultstyle=\hole,
  resultstyle.d=\white]{2.46}{35.7}
```

Chapter 3

Arithmetic Operations

3.1 Addition

Addition is deal by the \opadd macro. When it is in display mode, it display only nonnegative numbers. Then, it displays a substraction when one of the operands is nonpositive.

$$\begin{array}{r} -\ 2\,4\,5 \\ \ \ 7\,2 \\ \hline -\ 1\,7\,3 \end{array}$$

source

```
\opadd{-245}{72}
```

In a general manner, the principle is to display the operation that allows to find the result as you make it “by hand”. On the contrary, the inline mode shows always an addition since we can now write nonpositive numbers.

$-245+72=-173$

source

```
\opadd[style=text]{-245}{72}
```

In addition to the general parameters discussed in the section 2.2, the macro \opadd uses parameters carryadd, lastcarry, and deletezero.

The carryadd parameter is a boolean parameter, that is, it accepts only the values true and false. By habit, when you don’t specify the value and the equal sign, that is like assignment =true. This parameter indicates if the carries must be showed or not. Its default value is true.

The lastcarry parameter is also a boolean parameter. It indicates if a carry without matching digit for the two operands must be showed or not. Its default value is false. Take care to the exact rôle of this parameter. For instance, if the second operand in the following example is 15307, the last carry would be showed for any value of the lastcarry parameter since there is a matching digit in the second operand.

$$\begin{array}{r} \ {\scriptstyle 1}\ \ {\scriptstyle 1}\ \ \ \ \\ \ 4\,8\,2\,5 \\ +\ \ 5\,3\,0\,7 \\ \hline \ 1\,0\,1\,3\,2 \end{array}$$

source

```
\opadd{4825}{5307}
```

$$\begin{array}{r} \;4\,8\,2\,5 \\ +\quad 5\,3\,0\,7 \\ \hline 1\,0\,1\,3\,2 \end{array}$$

source

```
\opadd[carryadd=false]{4825}{5307}
```

$$\begin{array}{r} {\scriptstyle 1\;1\;\;\;1} \\ \;4\,8\,2\,5 \\ +\quad 5\,3\,0\,7 \\ \hline 1\,0\,1\,3\,2 \end{array}$$

source

```
\opadd[lastcarry]{4825}{5307}
```

The `deletezero` parameter is also a boolean parameter. It indicates if non-significant zeros must be deleted or not. Its default value is `true`. When this parameter is `false`, the operands and the result has the same number of digits. For that, xlop package adds non-significant zeros. Also, the non-significant zeros of operands are not removed.

$$\begin{array}{r} {\scriptstyle \;1\;1\;1} \\ 1\,2.3\,4\,2\,7 \\ +\quad 5.2\,7\,7\,3 \\ \hline 1\,7.6\,2 \end{array}$$

$$\begin{array}{r} {\scriptstyle \;1\;1\;1} \\ 0\,1\,2.3\,4\,2\,7 \\ +\;0\,0\,5.2\,7\,7\,3 \\ \hline 0\,1\,7.6\,2\,0\,0 \end{array}$$

source

```
\opadd{012.3427}{5.2773}\par
\opadd[deletezero=false]
  {012.3427}{5.2773}
```

This parameter has exactly the same rôle for inline mode than for displayed mode.

$2.8+1.2=4$
$02.8+1.2=04.0$

source

```
\opadd[style=text]{02.8}{1.2}\par
\opadd[style=text,
       deletezero=false]{02.8}{1.2}\par
```

3.2 Substraction

Substraction is made by `\opsub` macro. In displayed mode, the substraction shows only nonnegative numbers. For that, it shows an addition when one operand is nonpositive.

$$\begin{array}{r} {\scriptstyle 1} \\ \;2\,4\,5 \\ +\quad 7\,2 \\ \hline 3\,1\,7 \end{array}$$

source

```
\opsub{-245}{72}
```

In a general way, the principle is to display the operation which allow to find the result as you make it “by hand”. On the contrary, inline mode shows always a substraction since you can now write nonpositive numbers.

$-245-72=-317$

source
```
\opsub[style=text]{-245}{72}
```

This principle apply also when the first operand is less than the second one (positive case). In this case, we have an operand inversion.

$$\begin{array}{r} 2.45 \\ -\ 1.2 \\ \hline 1.25 \end{array}$$

source
```
\opsub{1.2}{2.45}
```

Of course, inline operation gives an exact result.

$1.2-2.45=-1.25$

source
```
\opsub[style=text]{1.2}{2.45}
```

In addition to general parameters we have seen at section 2.2, `\opsub` takes account of `carrysub`, `lastcarry`, `offsetcarry`, `deletezero`, and `behaviorsub` parameters.

The `carrysub` parameter is a boolean one which indicates if carries must be present or not. Its default value is `false`. (Remember that the default value of `carryadd` parameter is `true`.)

$$\begin{array}{r} 1\ 2\ {}^{1}3\ 1\!4 \\ -\ \ {}^{1}5\ {}^{1}6\ 7 \\ \hline 6\ 6\ 7 \end{array}$$

source
```
\opsub[carrysub]{1234}{567}
```

In the last example, you can see that there is no carry above the last digit of 1234. This is quite common (at least in France). If you want display this last carry, you have to use the `lastcarry` parameter. This parameter does not have the same behavior in substraction and in addition since here, the last carry is not displayed when the second operand does not have correspondent digit. (For addition, last carry is not displayed when *all* the operands do not have correspondent digit.)

$$\begin{array}{r} 1\ {}^{1}2\ {}^{1}3\ 1\!4 \\ -\ 1\ \ {}^{1}5\ {}^{1}6\ 7 \\ \hline 6\ 6\ 7 \end{array}$$

source
```
\opsub[carrysub,lastcarry]{1234}{567}
```

Note that, in this case, it is better to set the `deletezero` parameter to `false` in order to have a nicer result.

$$\begin{array}{r} 1\ {}^{1}2\ {}^{1}3\ 1\!4 \\ -\ {}^{1}0\ {}^{1}5\ {}^{1}6\ 7 \\ \hline 0\ 6\ 6\ 7 \end{array}$$

source
```
\opsub[carrysub,
       lastcarry,
       deletezero=false]{1234}{567}
```

Perhaps it seems to you that showing carries for substraction is a bit more dense. You can enlarge the figure box with the `opcolumnwidth` parameter. You can also indicate the carry horizontal shift using the `offsetcarry` parameter. Its default value is `-0.35`.

$$
\begin{array}{r}
{}_{1}1\,{}_{1}2.{}_{1}3\,{}_{1}4 \\
-\ {}_{1}0\,{}_{1}5.{}_{1}6\ \ 7 \\
\hline
\ \ 0\ \ 6.6\ \ 7
\end{array}
$$

source

```
\opsub[carrysub,
       lastcarry,
       deletezero=false]{12.34}{5.67}

\bigskip
\opsub[carrysub,
       lastcarry,
       columnwidth=2.5ex,
       offsetcarry=-0.4,
       decimalsepoffset=-3pt,
       deletezero=false]{12.34}{5.67}
```

$$
\begin{array}{r}
\ \ 1\ {}_{1}2.{}_{1}3\ {}_{1}4 \\
-\ {}_{1}0\ {}_{1}5.{}_{1}6\ \ 7 \\
\hline
\ \ 0\ \ 6.\ 6\ \ 7
\end{array}
$$

It is possible that a substraction with two positive numbers and with the first one less than the second one signs an user error. In this case, and only in this case, the `behaviorsub` parameter allows a call to order. The three possible values are:

- `silent` which is the default value and which gives the result;
- `warning` which gives also the result but shows the warning message:

  ```
  xlop warning.  Substraction with first operand less than second one
      See documentation for further information.
  ```

- `error` which shows the error message:

  ```
  xlop error.  See documentation for further information.
                   Type  H <return>  for immediate help.
  ! Substraction with first operand less than second one.
  ```

  and the operation is not performed.

3.3 Multiplication

The multiplication is under the control of the `\opmul` macro.

The parameters we will see below are `hfactor`, `displayintermediary`, `shiftintermediarysymbol`, and `deletezero`. We studied the other parameters in section 2.2.

The `shiftintermediarysymbol` parameter indicates what is the symbol used for showing the shifting of intermediary numbers (default value is `$\cdot$`). The `displayshiftintermediary` parameter can take value `shift` (default value) which shows this symbol only for shifting greater than one level, value `all` which shows this symbol for all the shiftings, and the value none which means that this symbol will be never showed.

source

```
\opmul[displayshiftintermediary=shift]{453}{1001205}\qquad
\opmul[displayshiftintermediary=all]{453}{1001205}\qquad
\opmul[displayshiftintermediary=none]{453}{1001205}
```

$$\begin{array}{r} 4\,5\,3 \\ \times\ 1\,0\,0\,1\,2\,0\,5 \\ \hline 2\,2\,6\,5 \\ 9\,0\,6\,\cdot \\ 4\,5\,3 \\ \hline 4\,5\,3\,\cdot\,\cdot \\ \hline 4\,5\,3\,5\,4\,5\,8\,6\,5 \end{array} \qquad \begin{array}{r} 4\,5\,3 \\ \times\ 1\,0\,0\,1\,2\,0\,5 \\ \hline 2\,2\,6\,5 \\ 9\,0\,6\,\cdot\,\cdot \\ 4\,5\,3\,\cdot\,\cdot\,\cdot \\ \hline 4\,5\,3\,\cdot\,\cdot\,\cdot\,\cdot\,\cdot\,\cdot \\ \hline 4\,5\,3\,5\,4\,5\,8\,6\,5 \end{array} \qquad \begin{array}{r} 4\,5\,3 \\ \times\ 1\,0\,0\,1\,2\,0\,5 \\ \hline 2\,2\,6\,5 \\ 9\,0\,6 \\ 4\,5\,3 \\ \hline 4\,5\,3 \\ \hline 4\,5\,3\,5\,4\,5\,8\,6\,5 \end{array}$$

In fact, null intermediary numbers are not display because of the default value none of the `displayintermediary` parameter. The value `all` shows all the intermediary numbers, even null intermediary numbers.

$$\begin{array}{r} 4\,5\,3 \\ \times\ 1\,0\,0\,1\,2\,0\,5 \\ \hline 2\,2\,6\,5 \\ 0\,0\,0 \\ 9\,0\,6 \\ 4\,5\,3 \\ 0\,0\,0 \\ 0\,0\,0 \\ 4\,5\,3 \\ \hline 4\,5\,3\,5\,4\,5\,8\,6\,5 \end{array}$$

source

```
\opmul[displayintermediary=all]
      {453}{1001205}
```

Note that null intermediary numbers are displayed with the same width than the first factor width.

The `displayintermediary` parameter accepts the value `nonzero` which means the same than the `none` value except when second factor has only one digit.

source

```
\opmul{3.14159}{4}\qquad
\opmul[displayintermediary=nonzero]{3.14159}{4}
```

$$\begin{array}{r} 3.1\,4\,1\,5\,9 \\ \times\ 4 \\ \hline 1\,2.5\,6\,6\,3\,6 \end{array} \qquad \begin{array}{r} 3.1\,4\,1\,5\,9 \\ \times\ 4 \\ \hline 1\,2\,5\,6\,6\,3\,6 \\ \hline 1\,2.5\,6\,6\,3\,6 \end{array}$$

Finally, parameter `displayintermediary` accepts the value `None` which don't display any intermediary numbers in all cases.

source

```
\opmul[displayintermediary=None]{453}{1001205}
```

$$\begin{array}{r} 4\,5\,3 \\ \times\ 1\,0\,0\,1\,2\,0\,5 \\ \hline 4\,5\,3\,5\,4\,5\,8\,6\,5 \end{array}$$

The `hfactor` parameter indicates how align operands. The default value, `right`, gives a raggedleft alignment. The `decimal` value gives an alignment on dot.

source
```
\opmul{3.1416}{12.8}\qquad\opmul[hfactor=decimal]{3.1416}{12.8}
```

$$\begin{array}{r} 3.1\,4\,1\,6 \\ \times\quad 1\,2.8 \\ \hline 2\,5\,1\,3\,2\,8 \\ 6\,2\,8\,3\,2 \\ 3\,1\,4\,1\,6 \\ \hline 4\,0.2\,1\,2\,4\,8 \end{array} \qquad \begin{array}{r} 3.1\,4\,1\,6 \\ \times\quad 1\,2.8 \\ \hline 2\,5\,1\,3\,2\,8 \\ 6\,2\,8\,3\,2 \\ 3\,1\,4\,1\,6 \\ \hline 4\,0.2\,1\,2\,4\,8 \end{array}$$

For displayed multiplication, the `deletezero` parameter is only for operands. The result keeps its non-significant zeros since there are necessary in order to make a correct dot shifting when we work "by hand".

source
```
\opmul[deletezero=false]{01.44}{25}\qquad
\opmul{01.44}{25}
```

$$\begin{array}{r} 0\,1.4\,4 \\ \times\quad 2\,5 \\ \hline 0\,7\,2\,0 \\ 0\,2\,8\,8 \\ \hline 0\,3\,6.0\,0 \end{array} \qquad \begin{array}{r} 1.4\,4 \\ \times\quad 2\,5 \\ \hline 7\,2\,0 \\ 2\,8\,8 \\ \hline 3\,6.0\,0 \end{array}$$

In the other hand, this parameter has its usual behaviour in inline multiplication.

source
```
\opmul[deletezero=false,style=text]{01.44}{25}\qquad
\opmul[style=text]{01.44}{25}
```

$01.44 \times 25 = 036.00 \qquad 1.44 \times 25 = 36$

3.4 Division

The xlop package deals with "normal" division via `\opdiv` macro and with euclidean division via `\opidiv` macro. Division is a very complex operation so it is not strange that there are many parameters to control it.

Pay attention that the xlop package v. 0.25 is unable to deal with "english" division. In this package version, the division is the "french" one, which is more or less used as it in some other countries. The xlop package v. 0.3 will allow "enlish" division (and many more feautures).

3.4.1 End Control

In the following text, term *step* means the set of process which allow to get one digit for the quotient. This number of steps is (not only) under the control of `maxdivstep`, `safedivstep`, and `period` parameters. It is only partially true because a classical division will stop automatically when a remainder will be zero, whatever the values of these three parameters and a euclidean division will stop with an integer quotient without attention for these three parameters.

```
2 5                    | 7
  4 0                  | 3.5 7 1 4 2 8 5 7 1
    5 0                |
      1 0              |
        3 0            |
          2 0          |
            6 0        |
              4 0      |
                5 0    |
                  1 0  |
                    3  |
```

source

```
\opdiv{25}{7}
```

```
2 5 | 7
  4 | 3
```

source

```
\opidiv{25}{7}
```

The first example stops because of the value of `maxdivstep` which is 10 by default. Pay attention that the maximum step number could cause strange result when it is too small.

```
1 2 4 8 | 3
  0 4   | 4 1
    1   |
```

source

```
\opdiv[maxdivstep=2]{1248}{3}
```

Clearly, the last result is false. In the other hand, xlop package did what we have ask, that is, obtain two digits (maximum) for the quotient.

The inline mode differ with zero remainder or not and with the type of division (classical or euclidean).

$3.14 \div 2 = 1.57$
$3.14 \div 3 \approx 1.046666666$
$314 = 2 \times 157$
$314 = 3 \times 104 + 2$

source

```
\opdiv[style=text]{3.14}{2}\par
\opdiv[style=text]{3.14}{3}\par
\opidiv[style=text]{314}{2}\par
\opidiv[style=text]{314}{3}
```

Note the use of `equalsymbol` or `approxsymbol` parameter according to the case. Note also that xlop displays results with floor, not with round. We will see how obtain a round in section 4.5.

For inline mode of `\opdiv`, xlop take account of `maxdivstep`. It means that we can obtain results very false with too small values of this parametrer and, unlike with display mode division, inline mode don't allow to understand what is wrong.

$1248 \div 3 \approx 41$

source
```
\opdiv[maxdivstep=2,style=text]
      {1248}{3}
```

In addition, if the last remainder is zero, we obtain a must:

$1208 \div 3 = 4$

source
```
\opdiv[maxdivstep=1,style=text]
      {1208}{3}
```

because there is no approximation at all!

A classical division can stop with period detection. For that, you have just to give the value `true` for the `period` parameter.

$$\begin{array}{r|l} 1\,0\,0 & 3 \\ \cline{2-2} \quad 1\,0 & 3\ 3.3 \\ \quad\ \ 1\,0 & \\ \quad\ \ \ 1 & \end{array}$$

source
```
\opdiv[period]{100}{3}
```

To avoid comparizons between each remainder with all previous remainder, xlop calculates immediatly the period length. That allows to process only one comparizon for each step, then to have a much more efficient process.[1] Unfortunately, these calculations are made with numbers that are directly accesible to TEX. As consequence, you can't use operand with absolute value greater than $\left\lfloor \frac{2^{31}-1}{10} \right\rfloor = 214748364$.

In order to avoid too long calculations, xlop don't process beyond the value of `safedivstep` parameter in division with period. Its default value is 50. However, xlop package show this problem. For example, if you ask for such a division with the code:

```
\opdiv[period]{1}{289}
```

you obtain the warning message:

```
xlop warning.  Period of division is too big (272 > safedivstep).
    Division will stop before reach it.
    See documentation for further information.
```

which indicates that this division period is 272 and that it can be achieved because of the `safedivstep` value.

The inline mode for division with period has some particularities.

[1] Thanks to Olivier Viennet about mathematic precisions that allows to implement these calculations.

$150 \div 7 = 21.\underline{428571}\ldots$

source
```
\opdiv[period,style=text]{150}{7}
```

We obtain an equality rather than an approximation, there is a rule under the period, and there is ellipsis after the period. All these components can be configured. The equality symbol is given by the `equalsymbol` parameter (default value is `{$=$}`). The rule thickness is given by the `hrulewidth` parameter (default value is `0.4pt`). The vertical offset of this rule is given by `vruleperiod` parameter (default value is `-0.2`) which indicates a vertical offset taking `\oplineheight` as unit. The ellipsis are given by the parameter `afterperiodsymbol` (default value `$\ldots$`).

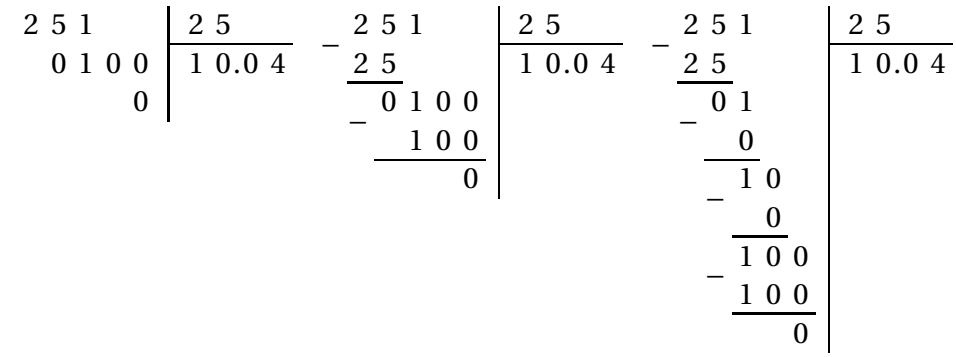
$150 \div 7 \approx 21.\overline{428571}$

source
```
\opdiv[period,style=text,
       equalsymbol=$\approx$,
       hrulewidth=0.2pt,
       vruleperiod=0.7,
       afterperiodsymbol=]
      {150}{7}
```

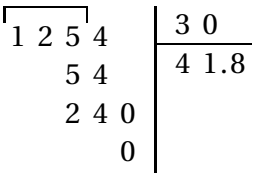
3.4.2 Other Features

Displayed divisions can include successive substractions which allow remainder calculations. For xlop, the numbers which are substracted are intermediary numbers, so the different ways to represent substractions use `displayintermediary` parameter see for multiplication. The default value, valeur none, don't display any substraction; the value `all` displays all the substractions, and the value `nonzero` displays substractions with non-zero numbers

source
```
\opdiv[displayintermediary=none,voperation=top]
      {251}{25}\quad
\opdiv[displayintermediary=nonzero,voperation=top]
      {251}{25}\quad
\opdiv[displayintermediary=all,voperation=top]
      {251}{25}
```



When we write a display division, we can draw a “bridge” over the part of dividend which is taken in count for the first step of calculation. The xlop package allow to draw this symbol thanks to the boolean parameter `dividendbridge` (default value is `false`).

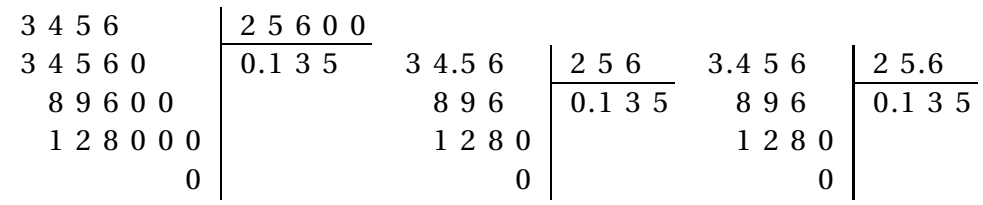


source
```
\opdiv[dividendbridge]{1254}{30}
```

3.4.3 Non Integer Numbers and Negative Numbers

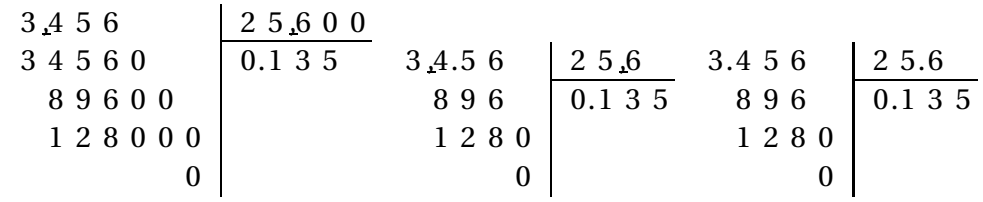
The `shiftdecimalsep` parameter governs non integer operands aspect/ Its default value is `both` which indicates that decimal separator is shifted in order to obtain integer divisor and integer dividend. The value `divisor` indicates that there is the shifting that allows an integer divisor. The value none indicates that there isn't any shifting.

source
```
\opdiv[shiftdecimalsep=both]{3.456}{25.6}\quad
\opdiv[shiftdecimalsep=divisor]{3.456}{25.6}\quad
\opdiv[shiftdecimalsep=none]{3.456}{25.6}
```



Parameter `strikedecimalsepsymbol` gives the symbol used to show the old place of decimal separator when this one is shifted. The default value is empty, that is, there isn't any symbol. This explain why you don't see anything on previous examples.

source
```
\opset{strikedecimalsepsymbol={\rlap{,}\rule[-1pt]{3pt}{0.4pt}}}
\opdiv[shiftdecimalsep=both]{3.456}{25.6}\quad
\opdiv[shiftdecimalsep=divisor]{3.456}{25.6}\quad
\opdiv[shiftdecimalsep=none]{3.456}{25.6}
```



When there is a non empty symbol for the striked decimal separator, it is possible to have non-significant zeros in operands.

$0{\times}0\ 3.4\ 5\ 6 \mid 2{\times}5\ 6$
$\quad 8\ 9\ 6 \mid 0.0\ 1\ 3\ 5$
$\quad 1\ 2\ 8\ 0$
$\quad\quad 0$

source

```
\opdiv[shiftdecimalsep=divisor,
       strikedecimalsepsymbol=%
         \hspace{-3pt}\tiny$\times$]
      {0.03456}{2.56}
```

We have already seen that `\opidiv` macro gives integer quotient. This is true even with non integer operands. It is somewhere strange to perform an euclidian division with non integer operands. The `\opidiv` macro will be strict about the presentation. Parameters `maxdivstep`, `safedivstep`, and `period` haven't any effect, as for `shiftdecimalsep` parameter since operands are changed to integer ones.

$3\ 4{\times}5\ 7 \mid 7{\times}0\ 0$
$\quad 6\ 5\ 7 \mid 4$

source

```
\opidiv[strikedecimalsepsymbol=%
        \hspace{-3pt}\tiny$\times$]
        {34.57}{7}
```

When operands are negative, the inline `\opidiv` numbers is different from the displayed `\opidiv` ones. Remainder will be between zero (include) and absolute value of divisor (exclude).

$124 \div 7 \approx 17.71428571$
$124 = 7 \times 17 + 5$
$124 = -7 \times -17 + 5$
$-124 = 7 \times -18 + 2$
$-124 = -7 \times 18 + 2$

source

```
\opdiv[style=text]{124}{7}\par
\opidiv[style=text]{124}{7}\par
\opidiv[style=text]{124}{-7}\par
\opidiv[style=text]{-124}{7}\par
\opidiv[style=text]{-124}{-7}
```

This condition for remainder is valid even with non integer divisor.

$1.24 = 0.7 \times 1 + 0.54$
$1.24 = -0.7 \times -1 + 0.54$
$-1.24 = 0.7 \times -2 + 0.16$
$-1.24 = -0.7 \times 2 + 0.16$

source

```
\opidiv[style=text]{1.24}{0.7}\par
\opidiv[style=text]{1.24}{-0.7}\par
\opidiv[style=text]{-1.24}{0.7}\par
\opidiv[style=text]{-1.24}{-0.7}
```

Chapter 4

Other Commands

4.1 Starred Macros

The five macros seen in previous chapter have a starred version. These starred macros perform the calculation and don't display anything. Result is record in a variable given as argument.

Since these commands don't display anything, parameters don't make sens and aren't allowed for `\opadd*`, `\opsub*`, `\opmul*`, and `\opidiv*`. In the other hand, parameters `maxdivestep`, `safedivstep`, and `period` influence calculations, then `\opdiv*` macro accepts an optional argument to take account of them.

source

```
\opmul*{2}{2}{a}%
\opmul*{a}{a}{a}\opmul*{a}{a}{a}%
\opadd[style=text]{a}{1}
```

$256 + 1 = 257$

For macros `\opdiv` and `\opidiv`, there are two extra arguments to record quotient and final remainder.

source

```
\opdiv*[maxdivstep=1]{-88}{16}{q}{r}%
\opmul*{q}{16}{bq}%
\opmul[style=text]{16}{q}\par
\opadd[style=text]{bq}{r}
```

$16 \times -5 = -80$
$-80 + -8 = -88$

4.2 Input-Output

The `\opcopy` macro copies its first argument into its second one. Then, the first argument is a number write in decimal form or *via* a variable, whereas the second one is a variable name.

The `\opprint` macro displays its argument. The following example uses the counter `\time` which indicates numbers of minutes since midnight.

It is 13 hours 55 minutes

source
```
\opidiv*{\the\time}{60}{h}{m}%
It is \opprint{h}~hours
\opprint{m}~minutes
```

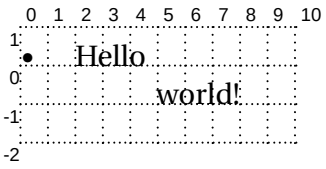
We will see at section 4.4 how to improve this example with tests.

The \opdisplay macro also displays a number but here, each figure is in a box. The width of this box is given by columnwidth and the height of this box is given by lineheight. Style is specified by the first argument. This macro accepts an optional argument in order to give a specific style for individual figures.

1 2 **9**.1 **9** 2

source
```
\opdisplay[resultstyle.1=\bfseries,
           resultstyle.-2=\bfseries]
          {resultstyle}{129.192}
```

Macros \oplput and \oprput allow to put anything anywhere. The syntax of both of them is different from the other ones of xlop since the place is indicated with coordinates between parenthesis. The coordinates use \opcolumnwidth and \oplineheight as units. Then user is able to build his own "operations".



source
```
\psset{xunit=\opcolumnwidth,
  yunit=\oplineheight}%
\psgrid[subgriddiv=1,gridlabels=7pt,
        griddots=5](0,1)(10,-2)
\oplput(2,0){Hello}
\oprput(8,-1){world!}
$\bullet$
```

On example above, note that these macros don't move the reference point. As a precaution, they kill the trailing space and then, there is no need to protect the end of line with a %.

Macros \ophline and \opvline complete the previous ones to give all the tools the user needs to build its own operations. \ophline allows to draw a horizontal rule; its length is given by the parameter after coordinates. \opvline does the same for vertical rules. Remember that parameters hrulewidth and vrulewidth indicate the thickness of these rules.

source
```
\par\vspace{2\oplineheight}
\oplput(1,2){O}\oplput(2,2){N}\oplput(3,2){E}
\oplput(0,1.5){$+$}
\oplput(1,1){O}\oplput(2,1){N}\oplput(3,1){E}
\ophline(0,0.8){4}
\oplput(1,0){T}\oplput(2,0){W}\oplput(3,0){O}
```

$$\begin{array}{r} \mathrm{ONE} \\ +\ \mathrm{ONE} \\ \hline \mathrm{TWO} \end{array}$$

Macro \opexport allow to export a number in a macro. It's an extra to version 0.23 which is very usefull to exchange datas between xlop and the outside world. The first argument is a number in the xlop sense, that is, either a number write with figures, or a variable name. The number is translated in a form directly acceptable for TEX and hold in the second argument which should be a macro name. However, note that decimal separator will be the one specified by decimalsepsymbol (without its possible braces).

source

```
\opmul*{5}{3.141592654}{F}
\opexport{F}{\fivepi}
\texttt{\meaning\fivepi}
```

macro:->15.70796327

We can use this macro to typeset numbers calculated by xlop in an array with a decimal alignment, or to initialize a counter or a length (don't forget the unit in the last case).

4.3 Figures of Numbers

Macros \opwidth, \opintegerwidth, and \opdecimalwidth indicate number of digits of the whole number, of its integer part, of its decimal part respectively. The first argument is the examined number and the second one indicates the variable where result will be record.

source

```
\opcopy{123456.1234}{a}%
\opwidth{a}{na}%
\opintegerwidth{a}{ia}%
\opdecimalwidth{a}{da}%
\opprint{a} is written with
\opprint{na} figures (\opprint{ia} in
the integer part and \opprint{da} in
the decimal part).
```

123456.1234 is written with 10 figures (6 in the integer part and 4 in the decimal part).

Macro \opunzero delete all the non-significant zeros of the number passed as argument.

source

```
\opcopy{00150.00250}{a}%
Before : \opprint{a}\par
\opunzero{a}%
After : \opprint{a}
```

Before : 00150.00250
After : 150.0025

Macros \integer and \opdecimal give the integer part and the decimal part of a number respectively. First argument is the number to process, and the second one is the variable name which hold the result.

source

```
\opcopy{-37.69911}{a}%
\opinteger{a}{ia}%
\opdecimal{a}{da}%
Integer part: \opprint{ia}\par
Decimal part: \opprint{da}
```

Integer part: 37
Decimal part: 69911

Six macros allow to write or read a figure of a number. You can read or read a figure according to its place in the whole number, or in the integer part, or in the decimal part. Figures for whole number and for decimal part are numbered from right to left, figures for integer part are numbered from left to right. For instance, with the number 1234.56789, the second figure is 8, the second figure of the integer part is 3, and the second figure of the decimal part is 6. It is now easy to guess the rôle of the six next macros:

- `opgetdigit` ;
- `opsetdigit` ;
- `opgetintegerdigit` ;
- `opsetintegerdigit` ;
- `opgetdecimaldigit` ;
- `opsetdecimaldigit` ;

Syntax is the same for these macros. The first argument is the processed number (reading or writting), the second one is the index of te figure, and the third one is the variable name which hold the result (figure read or changed number). If index is out of the range, the reading macros give 0 as result and writing macros extend the number in order to reach this index (for that, zero will be created in new slots).

4.4 Comparisons

When you want complex macros, often you need to realize tests. For that, xlop gives the macro `\opcmp`. The two arguments are numbers and this macro setup the tests `\ifopgt`, `\ifopge`, `\ifople`, `\ifoplt`, `\ifopeq`, and `\ifopneq` to indicate that first operand is greater, greater or equal, less or equal, less, equal, or different to the second operand respectively.

For technical reasons, xlop give global definitions for the six tests above. Then, they are not protected by groups. Since these tests are used by many xlop macros, you must *always* use tests `\ifop...` immediately after `\opcmp`, or, at least, before any use of a xlop macro. Otherwise, there will be bugs hard to fix!

Let's resume the hour display macro see at section 4.2. But now, we check if argument is between 0 (include) and 1440 (exclude), then we process tests in order to know if "hour" is plural or not, as for "minute".

source

```
\newcommand\hour[1]{%
  \opcmp{#1}{0}\ifopge
  \opcmp{#1}{1440}\ifoplt
    \opidiv*{#1}{60}{h}{m}%
    \opprint{h} hour%
```

```
    \opcmp{h}{1}\ifopgt
      s%
    \fi
    \opcmp{m}{0}\ifopneq
      \space\opprint{m} minute%
      \opcmp{m}{1}\ifopgt
        s%
      \fi
    \fi
  \fi\fi
}
\hour{60} -- \hour{1080} -- \hour{1081} -- \hour{1082}
```

1 hour – 18 hours – 18 hours 1 minute – 18 hours 2 minutes

4.5 Advanced Operations

The macros left to be examined are either internal macros and which it will be a shame to keep private , or macro asked for users.

Internal macros are `\opgcd` which gives gcd of two numbers and macro `\opdivperiod` which gives the period length of quotient of two numbers. For efficiency reason, these macros don't use xlop number, they rather use numbers directly understand by TEX. There are two consequences: the numbers can't be greater than 2147483647 for `\opgcd`; it can't be greater than 214748364 for `\opdivperiod`. A warning is displayed for an overflow. Result is put in the third parameter.

There is also some checks on the two first parameters: a gcd must not have null argument; length of period can't be processed with null quotient. Futhermore, if an argument is a non integer number, only the integer part will be take account.

source

```
\opcopy{5376}{a}%
\opcopy{2304}{b}%
\opgcd{a}{b}{gcd(ab)}%
$\gcd(\opprint{a},\opprint{b}) =
  \opprint{gcd(ab)}$
```

$\gcd(5376,2304) = 768$

You can play and find long period of divisions. Without going into mathematical details, square of prime numbers are good choices. For instance with $257^2 = 66049$ you obtain:

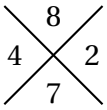
source

```
\opdivperiod{1}{66049}{p}%
$\frac{1}{66049}$ has a period
of length $\opprint{p}$.
```

$\frac{1}{66049}$ has a period of length 65792.

With macros `\opcastingoutnines` and `\opcastingoutelevens` you can build casting out of nines and casting out of elevens. xlop don't typeset

directly these “operations” since they need diagonal rules, and then, need some particular packages. In fact, macro \opcastingoutnines calculates the sum modulo 9 of first argument digits and put the result in second argument. Macro \opcastingoutelevens calculates the sum modulo 11 of the even rank digits of first argument, calculates the sum moldulo 11 of the odd rank digits of first argument, and calculates the difference of these two sums.



source

```
\newcommand\castingoutnines[3]{%
  \opcastingoutnines{#1}{cna}%
  \opcastingoutnines{#2}{cnb}%
  \opmul*{cna}{cnb}{cna*cnb}
  \opcastingoutnines{cna*cnb}{cna*cnb}%
  \opcastingoutnines{#3}{cn(a*b)}%
  \begin{pspicture}(-3.5ex,-3.5ex)%
                    (3.5ex,3.5ex)
    \psline(-3.5ex,-3.5ex)(3.5ex,3.5ex)
    \psline(-3.5ex,3.5ex)(3.5ex,-3.5ex)
    \rput(-2.75ex,0){\opprint{cna}}
    \rput(2.75ex,0){\opprint{cnb}}
    \rput(0,2.75ex){\opprint{cna*cnb}}
    \rput(0,-2.75ex){\opprint{cn(a*b)}}
  \end{pspicture}
}
\castingoutnines{157}{317}{49669}
```

In passing, this example shows that $157 \times 317 \neq 49669$! The right operation is $157 \times 317 = 49769$.

The two next macros are very simple. We have \opneg which calculates the opposite of its first argument and store it in the variable indicated by the second argument. We have also \opabs which does the same with absolute value.

Macro \oppower calculates integer powers of numbers. This macro has three parameters. The third one store the first argument to the power of the second argument. When the first argument is zero: if the second argument is zero, result is 1; if the second argument is positive, result is 0; if the second argument is negative, there is an error. There isn't any limitation on first parameter. This leads to some problems, for instance:

source

```
\opcopy{0.8}{a}\opcopy{-17}{n}%
\oppower{a}{n}{r}%
$\opprint{a}^{\opprint{n}} = \opprint{r}$
```

$$0.8^{-17} = 44.408920985006261616945266723632812 5$$

With 0.7 rather than 0.8, problem is worse:

source

```
\opcopy{0.7}{a}\opcopy{-8}{n}%
\oppower{a}{n}{r}%
\opdecimalwidth{r}{dr}
$\opprint{a}^{\opprint{n}}$ has \opprint{dr}
figures after dot.
```

0.7^{-8} has 72 figures after dot.

In fact, when exponent is negative, *first* xlop calulates inverse of the number and *after that*, it calculates the power with opposite of the exponent. In this example, if we had left -17 rather than -8, then there will be a capacity overflow capacity of TEX.

Three macros allow a control about precision. They allow to approximate a number giving the rank of the approximation. There are `\opfloor`, `\opceil`, and `\oproun`d. They need three parameters which are (in order): start number, rank of approximation, variable name to store the result.

Rank is an integer value giving number of digits after decimal separator which must be present. If this rank is negative, approximation will be done before the decimal separator. If rank is positive and indicates more digits than decimal part has, then zeros will be added. If rank is negative and indicates more digits than integer part has, then approximation will be locked in order to give the first digit of the number at least.

Here is a summary table which allow to understand how these macros work.

`\op...{3838.3838}{n}{r}`			
n	**floor**	**ceil**	**round**
6	3838.383800	3838.383800	3838.383800
4	3838.3838	3838.3838	3838.3838
3	3838.383	3838.384	3838.384
0	3838	3839	3838
−1	3830	3840	3840
−2	3800	3900	3800
−6	3000	4000	4000

`\op...{-3838.3838}{n}{r}`			
n	**floor**	**ceil**	**round**
6	−3838.383800	−3838.383800	−3838.383800
4	−3838.3838	−3838.3838	−3838.3838
3	−3838.384	−3838.383	−3838.384
0	−3839	−3838	−3838
−1	−3840	−3830	−3840
−2	−3900	−3800	−3800
−6	−4000	−3000	−4000

The very last macro we have to study is \opexpr. It calculates a complex expression. This macro needs two parameters: the first one is the expression in infix form (the natural one for human), the second one is the variable name where the result is stored.

Initially, expression must have been polish one (for instance, notation used on old HP calculator, or PostScript language), but another work with Christophe Jorssen has given the actual form for expression in xlop, more pleasant for users.

Formulas accept usual arithmetic operators +, −, ∗, and /. They accept also : operator for euclidian division, and ^ for power. The − operator has both rôle of substraction and unary operator for opposite. The + has also these rôles, here the unary operator do... nothing! Operands are written in decimal form or *via* variable name. However, \opexpr introduces a restriction about variable name since variable names must be different to function names recognized by \opexpr. Accessible functions are:

- abs(a);
- ceil(a,i);
- decimal(a);
- floor(a,i);
- gcd(a,b);
- integer(a);
- mod(a,b) gives result of a modulo b;
- rest(a,b) gives remainder of a divide by b (difference between remainider and modulo is the same as between non euclidian division and euclidian division);
- round(a,i).

where functions that aren't listed above ask the matching macros. (function xxx calls macro \opxxx) For functions ceil, floor, and round, the number i indicates rank for approximation.

Macro \opexpr accept optional argument since it can realize division which can be controlled by maxdivstep,safedivstep, and period parameters. Our first example is quite basic:

source

```
\opexpr{3--gcd(15*17,25*27)*2}{r}%
$3--\gcd(15\times17,25\times27)\times2 = \opprint{r}$
```

$$3--\gcd(15\times17,25\times27)\times2=33$$

Here is another example that shows that datas can come from a macro:

source

```
\newcommand\try{2}%
\opexpr{\try+1/
  (\try+1/
    (\try+1/
      (\try+1/
        (\try+1/
          (\try))))){r}
Continued fraction of base $u_n=2$ equal \opprint{r} at rank~5.
```

Continued fraction of base $u_n = 2$ equal 2.414285714 at rank 5.

Appendix A

Short Summary

A.1 Compilation times

Compilation times was measured on a computer with processor Pentium II 600 MHz, RAM 256 MB, on linux system (Debian woody).[1]. The principle is to do a minimal file `.tex`. The general canvas is:

```
\input xlop
\count255=0
\loop
\ifnum\count255<1000
  <operation to test>
 \advance\count255 by1
\repeat
\bye
```

Compilation time with `<operation to test>` empty was substract from the others test. Only the user time was take account. Results are given in millisecond and should be read with great precautions.

Next table gives operation times in milliseconds. Operands used had decimal notation but some trails with variable has shown that times was very closed.

First line indicates the numbers of digits for both operands. Operands were build like this:

- A = 1 et B = 9 for one digit;
- A = 12 et B = 98 for two digits;
- A = 123 et B = 987 for three digits;
- A = 12345 et B = 98765 for five digits;
- A = 1234567890 et B = 9876543210 for ten digits;

[1] In fact, these measures was done in 2004, when the 0.2 version was released. Author is somewhere lasy and he doesn't measure with his new computer (more efficient)!

- A = 12345678901234567890 et B = 98765432109876543210 for twenty digits;

Here is results, some comments follow:

	1	2	3	5	10	20
`\opadd*{A}{B}{r}`	1.1	1.4	1.6	2.1	3.3	5.8
`\opadd*{B}{A}{r}`	1.1	1.4	1.6	2.1	3.3	5.8
`\opsub*{A}{B}{r}`	1.7	2.1	2.4	3.0	4.8	8.3
`\opsub*{B}{A}{r}`	1.5	1.7	2.0	2.6	4.0	7.0
`\opmul*{A}{B}{r}`	4.6	6.3	8.2	12.8	29.9	87.0
`\opmul*{B}{A}{r}`	5.0	6.6	8.5	13.2	30.3	87.8
`\opdiv*{A}{B}{q}{r}`	46.4	53.8	53.8	64.3	85.8	124.7
`\opdiv*{B}{A}{q}{r}`	12.4	48.9	55.7	58.6	72.8	111.0
`\opdiv*[maxdivstep=5]{A}{B}{q}{r}`	26.8	30.0	32.6	37.6	49.5	73.5
`\opdiv*[maxdivstep=5]{B}{A}{q}{r}`	12.4	29.1	32.6	35.2	43.3	67.9
`\opidiv*{A}{B}{q}{r}`	10.8	12.2	13.5	16.0	22.3	35.5
`\opidiv*{B}{A}{q}{r}`	11.6	13.0	14.2	16.6	23.0	36.7
`\opidiv*{A}{2}{q}{r}`	10.7	12.0	15.3	22.3	42.9	83.0

It is normal that inversion of operands don't have sensible influence for addition. Then, it could be strange that there is influence for substraction. In fact, when the second operand is bigger than the second one, there is additional process (double inversion, operation on the sign of the result).

It is normal that division time is greater than the multiplication one. It could be abnormal that division seems catch up! In fact, the multiplication complexity grows quickly with the operand length. In the other hand, division complexity is stopped by `maxdivstep` parameter. It is clear on example where there is only five steps.

Some results seems odd. For instance `\opdiv*{9}{1}{q}{r}` is very fast. These is due to the one digit quotient. `\opdiv*{123}{987}{q}{r}`, even more odd, is rather fast. Here, explanation is quite subtle: this is due to many zeros in the quotient.

When operands have comparable length, euclidian division is much faster than non euclidian one. This is because quotient has few digits (only one for all the numbers `A` and `B`). The last line of the table is more relevant for this operation time.

All these remarks are written to put the emphasis on the difficulty to evaluate the compilation time: it depends on too many parameters. On the other hand, this table give a pretty good idea of what can be expected.

A.2 Macros List

Macro	Description
\opabs{n}{N}	N stores the absolute value of n.
\opadd[P]{n1}{n2}	Displays result of n1+n2.
\opadd*{n1}{n2}{N}	Calcules n1+n2 and put result in N.
\opcastingoutelevens{n}{N}	Calcules difference (modulo 11) of sum of rank odd digits and sum of rank even digits of n and put the result in N.
\opcastingoutnines{n}{N}.	Calcules sum modulo 9 of digits of n and put result in N.
\opceil{n}{T}{N}	Places in N the approximation (ceiling) of n to rank T.
\opcmp{n1}{n2}	Compares numbers n1 and n2 and setup the tests \ifopeq, \ifopneq, \ifopgt, \ifopge, \ifople et \ifoplt.
\opcopy{n}{N}	Copy number n in N.
\opdecimal{n}{N}	Copy decimal part (positive integer number) of n in N.
\opdecimalwidth{n}{N}	N stores the width of decimal part of number n.
\opdisplay[P]{S}{n}	Display number n width style S puting each figure in a box which has a width of \opcolumnwidth and a height of \oplineheight.
\opdiv[P]{n1}{n2}	Display result of n1/n2.
\opdiv*[P]{n1}{n2}{N1}{N2}	Calculates n1/n2, put the quotient in N1 and the remainder in N2.
\opdivperiod{T1}{T2}{N}	Calculates length of period of T1 divide by T2 and put the result in N.
\opexport[P]{n}\cmd	Copy number n in macro \cmd.
\opexpr[P]{F}{N}	Evaluates formula F and put the final result in N.
\opfloor{n}{T}{N}	Put in N the apprimation (floor) of n at rank T.
\opgcd{T1}{T2}{N}	Calculates gcd of T1 and T2 and put result in N.
\opgetdecimaldigit{n}{T}{N}	Build the number N with the only digit in slot T of decimal part of n.
\opgetdigit{n}{T}{N}	Build the number N with the only digit in slot T of number n.
... to be continued ...	

Macro	Description
\opgetintegerdigit{n}{T}{N}	Build the number N width the only digit in slot T of integer part of n.
\ophline(T1,T2){T3}	Draw a horizontal rule of length T3, of thickness hrulewidth, and which begin at (T1,T2) in relation to reference point.
\opidiv[P]{n1}{n2}	Display the result of n1/n2. (euclidian division, that is, with integer division)
\opidiv*{n1}{n2}{N1}{N2}	Calculates n1/n2 (euclidian division), put quotient (integer) in N1 and remainder (between 0 (include) and |n2| (exclude)) in N2.
\opinteger{n}{N}	Copy integer part (positive integer number) of n in N.
\opintegerwidth{n}{N}	Number N stores the width of integer part of number n.
\oplput(T1,T2){<object>}	Put <object> to the right of the point with coordinates (T1,T2) in relation to reference point.
\opmul[P]{n1}{n2}	Display result of n1*n2.
\opmul*{n1}{n2}{N}	Calculates n1*n2 and put the result in N.
\opneg{n}{N}	Number N stores opposite of n.
\oppower{n}{T}{N}	Calculates n to the power of T and put the result in N.
\opprint{n}	Display number n in a direct way.
\opround{n}{T}{N}	Put in N the approximation of n at rank T.
\oprput(T1,T2){<object>}	Put <object> to the left of the point with coordinates (T1,T2) in relation to reference point.
\opset{L}	Allocates globally xlop parameters given in the list L.
\opsetdecimaldigit{n}{T}{N}	Modify the digit of rank T in decimal part of N in order to have the value n for this digit.
\opsetdigit{n}{T}{N}	Modify the digit of rank T of N in order to have the value n for this digit.
\opsetintegerdigit{n}{T}{N}	Modify the digit of rank T in integer part of N in order to have the value n for this digit.
\opsub[P]{n1}{n2}	Display result of n1−n2.
... to be continued ...	

Macro	Description
\opsub*{n1}{n2}{N}	Calculates n1−n2 and put the result in N.
\opunzero{N}	Delete non-significant zeros of N.
\opvline(T1,T2){T3}	Draw a vertical ruleof length T3, of thickness hrulewidth and which begin at (T1,T2) in relation to reference point.
\opwidth{n}{N}	Number N stores number of digits of number n.

In this table, parameters:

- n and ni (where i is an index) indicate that parameter must be a number given in decimal form or a variable name;
- N and Ni (where i is an index) indicate that parameter must be a number given in decimal form or a variable name;
- [P] indicates that the macro accept an optional parameter which allow to modify parameter of xlop;
- T and Ti (where i is an index) indicate that parameter must be a number given in decimal form or a variable name but must be less than numbers acceptable by TEX, that is, $-2147483648 \leq T \leq 2147483647$.

A.3 Parameter list

Parameter	Default	Signification
afterperiodsymbol	$\ldots$	Symbol used after a period of a division.
approxsymbol	$\approx$	Symbol used as approximation relation in inline operations.
equalsymbol	{$=$}	Symbol used as equality relation in inline operations.
addsymbol	$+$	Symbol used as addition operator.
subsymbol	$-$	Symbol used as substraction operator.
mulsymbol	$\times$	Symbol used as multiplication operator.
divsymbol	$\div$	Symbol used as multiplication operator for inline operations.
decimalsepsymbol	.	Symbol used as decimal separator.
strikedecimalsepsymbol		Symbol used as decimal separator moved in dividend and divisor for display division.
shiftintermediarysymbol	$\cdot$	Symbol used to show intermediary numbers shifting for display multiplication.
... to be continued ...		

Parameter	Default	Signification
displayshiftintermediary	shift	Indicates that the shifting character for multiplications will be displayed only for additional shifting (value shift), for all the shifting (value all), or never (value none).
voperation	bottom	Vertical alignement for displayed operation. The value bottom indicates that the bottom of operation will be aligned with baseline. The value top indicates that the top of operation will be aligned with baseline. The value center indicates that operation will be verticaly centred with baseline.
voperator	center	Vertical alignement for operators in displayed operations. The value top put operator at the level of first operand. The value bottom put operator at the level of second operand. The value center put operator between operands.
hfactor	decimal	Sort of operands alignement for displayed operation. The value decimal indicates an alignement on decimal separator. The value right indicates a flushright alignement.
vruleperiod	-0.2	Vertical position of rule which indicates period of quotient for inline division.
dividendbridge	false	Indicates if there is a “bridge” above dividend.
shiftdecimalsep	both	Indicates how shift decimal separator into operands for a displayed division. The value both indicates that shifting are made on both divisor and dividend in order to make integer numbers. The value divisor indicates that the shifting must give an integer divisor. The value none indicates that there is no shifting.
maxdivstep	10	Maximal number of steps in division.
safedivstep	50	Maximal number of steps in division when there is a period to reach.
period	false	Indicates if division must be stoped when a whole period is reached.
deletezero	true	Indicates that non-significant zeros are displayed (false) or deleted (true).
… to be continued …		

Parameter	Default	Signification
carryadd	true	Indicates that carries are displayed (true) for displayed additions.
carrysub	false	Indicates that carries are displayed (true) for displayed substractions.
offsetcarry	-0.35	Horizontal offset for carries into displayed substractions.
style	display	Indicates tha operation are inline (text) or displayed (display).
displayintermediary	nonzero	Indicates that all intermediary results are displayed (all), only non null ones are displayed (nonzero), or any intermediary result isn't displayed into displayed multiplications and divisions.
lastcarry	false	Indicates that carry with no figure just below it must be displayed (true), or not (false).
parenthesisnegative	none	Behavior to display negative numbers in inline operations. The value none displays them without parenthesis. The value all displays them always with parenthesis. The value last display parenthesis except for first operand of an expression.
columnwidth	2ex	With of box for one figure.
lineheight	\baselineskip	Height of box for one figure.
decimalsepwidth	0pt	Width of box that hold the decimal separator.
decimalsepoffset	0pt	Horizontal offset for decimal separator.
hrulewidth	0.4pt	Thickness of horizontal rules.
vrulewidth	0.4pt	Thickness of vertical rules.
behaviorsub	silent	xlop behavior for an "impossible" substraction, that is, a substraction with two positive operands, the second greater than the first one. The value silent does operation swapping the two operands in a slient way. With the value warning, there are also a swapping but xlop gives a warning. The value error display an error message and operation isn't processed.
… to be continued …		

Parameter	Default	Signification
country	french	Indicates the displayed operation behavior depending of contry. Package xlop put forward only french, american, and russian but these different ways to display operations aren't encoded in version 0.25.
operandstyle		Style for operands.
resultstyle		Style for results.
remainderstyle		Style for remainders.
intermediarystyle		Style for intermediary results (intermediary numbers in multiplication and number to substract in division when successive substractions are displayed).
carrystyle	\scriptsize	Style for carries. The default value when compilation are made without LaTeX is \sevenrm.

Appendix B

Tricks

B.1 xlop vs. calc and fp

You could believe that xlop can replace package such calc and fp. In fact, that is not so simple. Obviously xlop can do complex calculations, on arbitrary long numbers but, unlike calc, it don't allow to process directly dimensions. Comparison with fp is somewhere more realistic but remember that xlop can make memory usage too high.

If you want to process calculations on length, you can use that a dimen register allocation to a counter gives a number which correspond to this length with unit sp.

source

```
\newcommand\getsize[2]{%
  \dimen0=#1\relax
  \count255=\dimen0
  \opcopy{\the\count255}{#2}}
\getsize{1pt}{r}$1\,\mathrm{pt}=\opprint{r}\,\mathrm{sp}$\quad
\getsize{1pc}{r}$1\,\mathrm{pc}=\opprint{r}\,\mathrm{sp}$\quad
\getsize{1in}{r}$1\,\mathrm{in}=\opprint{r}\,\mathrm{sp}$\quad
\getsize{1bp}{r}$1\,\mathrm{bp}=\opprint{r}\,\mathrm{sp}$\quad
\getsize{1cm}{r}$1\,\mathrm{cm}=\opprint{r}\,\mathrm{sp}$\quad
\getsize{1mm}{r}$1\,\mathrm{mm}=\opprint{r}\,\mathrm{sp}$\quad
\getsize{1dd}{r}$1\,\mathrm{dd}=\opprint{r}\,\mathrm{sp}$\quad
\getsize{1cc}{r}$1\,\mathrm{cc}=\opprint{r}\,\mathrm{sp}$\quad
\getsize{1sp}{r}$1\,\mathrm{sp}=\opprint{r}\,\mathrm{sp}$\quad
```

1 pt = 65536 sp 1 pc = 786432 sp 1 in = 4736286 sp 1 bp = 65781 sp
1 cm = 1864679 sp 1 mm = 186467 sp 1 dd = 70124 sp 1 cc = 841489 sp
1 sp = 1 sp

However, don't forget that the xlop main goal is to *display* operations.

With this `\getsize` macro, it is possible to realise calculations on length.

Surface of spread is 106.65 cm^2

source

```
\newcommand\getsize[2]{%
  \dimen0=#1\relax
  \count255=\dimen0
  \opcopy{\the\count255}{#2}}
\getsize{1cm}{u}%
\getsize{\textwidth}{w}%
\getsize{\textheight}{h}%
\opexpr{w*h/u^2}{S}%
\opround{S}{2}{S}%
Surface of spread is
\opprint{S}\,$\mathrm{cm}^2$
```

B.2 Complex Operations

Use of xlop macros with loop of TEX allow to create operations as you want. Here, we give only two examples. The first one can express a number as a product of prime factors, the second one is a general calculation for continued fraction.

source

```
\newcount\primeindex
\newcount\tryindex
\newif\ifprime
\newif\ifagain
\newcommand\getprime[1]{%
  \opcopy{2}{P0}%
  \opcopy{3}{P1}%
  \opcopy{5}{try}
  \primeindex=2
  \loop
    \ifnum\primeindex<#1\relax
    \testprimality
    \ifprime
      \opcopy{try}{P\the\primeindex}%
      \advance\primeindex by1
    \fi
    \opadd*{try}{2}{try}%
    \ifnum\primeindex<#1\relax
      \testprimality
      \ifprime
        \opcopy{try}{P\the\primeindex}%
        \advance\primeindex by1
      \fi
      \opadd*{try}{4}{try}%
    \fi
  \repeat
}
\newcommand\testprimality{%
  \begingroup
```

```
    \againtrue
    \global\primetrue
    \tryindex=0
    \loop
      \opidiv*{try}{P\the\tryindex}{q}{r}%
      \opcmp{r}{0}%
      \ifopeq \global\primefalse \againfalse \fi
      \opcmp{q}{P\the\tryindex}%
      \ifoplt \againfalse \fi
      \advance\tryindex by1
    \ifagain
    \repeat
  \endgroup
}
```

With this code, we can create a prime numbers list (here the 20 first ones).

source

```
\getprime{20}%
\opprint{P0}, \opprint{P1}, \ldots,
\opprint{P9}, \ldots \opprint{P19}.
```

2, 3, ..., 29, ...71.

Note that this code is very bad: it is very slow and don't give anything against native TEX operations. It is only a educational example. Note also that the tricks to put loop into loop with macro `\testprimality` inside a group. xlop operations give global results.

Once you have your prime numbers "table", you can use it to write a number as product of prime number.

source

```
\newcommand\primedecomp[2][nil]{%
  \begingroup
    \opset{#1}%
    \opcopy{#2}{NbtoDecompose}%
    \opabs{NbtoDecompose}{NbtoDecompose}%
    \opinteger{NbtoDecompose}{NbtoDecompose}%
    \opcmp{NbtoDecompose}{0}%
    \ifopeq
      I refuse to factorize zero.
    \else
      \setbox1=\hbox{\opdisplay{operandstyle.1}%
          {NbtoDecompose}}%
      {\setbox2=\box2{}}%
      \count255=1
      \primeindex=0
      \loop
      \opcmp{NbtoDecompose}{1}\ifopneq
        \opidiv*{NbtoDecompose}{P\the\primeindex}{q}{r}%
        \opcmp{0}{r}\ifopeq
```

```
          \ifvoid2
            \setbox2=\hbox{%
              \opdisplay{intermediarystyle.\the\count255}%
                {P\the\primeindex}}%
          \else
            \setbox2=\vtop{%
              \hbox{\box2}
              \hbox{%
                \opdisplay{intermediarystyle.\the\count255}%
                  {P\the\primeindex}}}
          \fi
          \opcopy{q}{NbtoDecompose}%
          \advance\count255 by1
          \setbox1=\vtop{%
            \hbox{\box1}
            \hbox{%
              \opdisplay{operandstyle.\the\count255}%
                {NbtoDecompose}}
          }%
        \else
          \advance\primeindex by1
        \fi
      \repeat
      \hbox{\box1
        \kern0.5\opcolumnwidth
        \opvline(0,0.75){\the\count255.25}
        \kern0.5\opcolumnwidth
        \box2}%
    \fi
  \endgroup
}

\getprime{20}%
\primedecomp[operandstyle.2=\red,
             intermediarystyle.2=\red]{252}
```

2 5 2	2
1 2 6	2
6 3	3
2 1	3
7	7
1	

Note the use of group for the whole macro in order to protect xlop parameter modifications. Note also that void parameter aren't allowed. It's not a bug, it's a feature. Author thinks that a user who write brackets without anything between these brackets is going to make a mistake. To obviate this

prohibition, there is the particular parameter `nil` which has exactly this rôle.

Finally, note the trick `{\setbox2=\box2}` to obtain a void box register, and final manipulations to show the vertical rule in a easy-to-read way.

The second example allow to calculates a continued fraction like:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cdots}}}$$

giving the sequence $a_0, a_1, a_2, a_3, \ldots$ to the macro. This example gives fractions corresponding to gold number, and square root for 2 and 3.

source

```
\begingroup
\long\gdef\continuedfraction#1#2{%
  \let\@mirror\relax
  \@for\op@Nb:=#1\do
  {%
    \ifx\@mirror\relax
      \edef\@mirror{\op@Nb}%
    \else
      \edef\@mirror{\op@Nb,\@mirror}%
    \fi
  }%
  \let\Op@result\relax
  \@for\op@Nb:=\@mirror\do
  {%
    \ifx\Op@result\relax
      \opcopy{\op@Nb}{result}%
    \else
      \opexpr{\op@Nb+1/result}{result}%
    \fi
  }%
  \opcopy{result}{#2}%
}
\endgroup
\continuedfraction{1,1,1,1,1,1,1,1,1,1,1,1}{r}\opprint{r}\quad
\continuedfraction{1,2,2,2,2,2,2,2,2,2,2,2}{r}\opprint{r}\quad
\continuedfraction{1,1,2,1,2,1,2,1,2,1,2,1}{r}\opprint{r}
```

1.618055555 1.414213564 1.732051282

It does no harm just this once, we use LaTeX commands for the loop.

B.3 Direct Access to Number

When a number is saved in a xlop variable, it is possible to process with it in many different ways. However, in certain situations, you would creat you own macro or use external macro giving such numbers as parameter.

Giving directly `\opprint{var}` is ineffective since this macro is a complex a gives side effect. It is necessary to access directly to this number. When a variable hold a number, xlop creates a macro `\Op@var` which contain this number. Note the uppercase “O” and the lowercase “p”. The at sign is here to do this definition a private one, that is, you have to enclose it with `\makeatletter` and `\makeatother` to access it (or `\catcode @=11` in TEX).

source

```
\opcopy{1234}{a}\opcopy{56}{b}%
\opmul*{a}{b}{r}%
\makeatletter
\newcolumntype{.}{D{.}{.}{-1}}
\begin{tabular}{l.}
           & \Op@a \\
  $\times$ & \Op@b \\
  $=$      & \Op@r
\end{tabular}
\makeatother
```

$$\begin{array}{l.} & 1234 \\ \times & 56 \\ = & 69104 \end{array}$$

Note that this way of doing don't work when decimal separator is between braces since macro `\opprint{var}` contain such braces. In this case, the simplest is to use `\opexport` macro (see page 25).

Appendix C

Future Versions

Version of xlop package is 0.25 which is only a debuging version of version 0.2, which is itself a correcting version of version 0.1 (first public release). The next release will be version 0.3 and its "stable" version will be version 0.4.

The features of version 0.3 aren't definitively fixed but there are some points planned:

- international version for posées;
- opérations from 2 to 36 basis;
- additional high level functions with roots (`\oproot` for any roots and `\opsqrt` for square root), exponential function, logarithm, trigonometric functions (direct, inverse, hyperbolic);
- macro to have a formated writing, that is, write a number where length of decimal part and integer part are given (if these widths are not the ones of the number, there will be overflow or filling); this macro was present in version 0.1 and allow to display numbers decimal aligned, right aligned, or left aligned;
- macro for addition with more than two operands;
- parameter for scientific or engineer notation;
- macro to allow to write a multi-line number and/or with thousand separator;
- carries for multiplications;
- make public the successive remainders of a division;
- negative values of `maxdivstep` and `safedivstep` parameters will take acount of decimal digit of quotient.

For all requests or bug report, the author will be grateful to you to contact him at:

Jean-Come.Charpentier@wanadoo.fr

placing the word “xlop” in the subject in order to help my spam killer.

It would be nice to have a hacker manual which explain in details the source. This tool could be usefull in order to improve xlop. Unfortunately, the current code has more than 4000 lines and the work to do that may well be too long.

Appendix D

Index