The Package widetable*

Claudio Beccari†

September 30, 2009

Contents

Abstract

This package allows to typeset tables of specified width, provided they fit in one page. Instead of introducing an infinite stretching glue, which has an unsymmetrical effect in standard LaTeX, here the `\tabcolsep` dimension is computed so as to have the table come out with the proper width.

1 Legalese

*Version number v.1.0; last revision 2009/07/21.

†`claudio dot beccari at gmail dot com`

2 Introduction

It is well known that when the standard environment `tabular*` is opened with a specified width, it is necessary to introduce in the delimiter declaration `@{...}` of (possibly) the first cell of the model row a declaration such as

```
\extraclosep{\fill}
```

in addition to other possible printable delimiters, such as vertical lines, and other fixed spacing commands. The effect is that the extra stretchable glue operates only on the left of each cell *after* (to the *right* of) the cell that received the declaration; the first cell will never get larger in spite of the presence of this glue.

Another package, `tabularX`, normally distributed by the LaTeX 3 Team with every version of the TeX system distribution, allows to create expandable cells, provided they contain only text. These expandable cells are identified with the column identifier `X`; this identifier defines a paragraph-like cell, the width of which gets determined after some runs of the typesetter on the same source tabular material, so as to find out the correct width of the textual columns.

The approach here is a little bit different: the cell contents need not be textual and no cell width is determined in one or more runs of the typesetter; instead the inter column glue is determined so as to fill every cell on both sides with the proper space. The macros contained in this package are insensitive to the particular kind of cell descriptors and to the presence of multiple `\multicolumn` commands. It proved to work properly also if the `array` package extensions are used.

On the other hand, as well as for `tabularX`, it needs to typeset the table three times; the first two times with standard values for the inter column glue `\tabcolsep`, in order to find the exact parameters of the linear dependence of the table width from the value of that glue; then executes some computations so as to extrapolate the final correct value of `\tabcolsep`, and on the third run it eventually typesets the table with the specified width.

The time increase needed for these three table typesettings are in general rather negligible, nevertheless if a specific document contained many dozens of such tables, the compilation time might become observable.

It might be noticed that in order to perform the necessary computations a fractional division algorithm had to be implemented; A specific LaTeX run that loads several different packages might then contain several fractional division macros, besides those already contained in the kernel. But unfortunately each of these macros has been designed for a specific purpose and a specific interface.

3 Usage

This package issues an error message only in case the environment includes other unhided environment; this is explained in the Implementation section. Here it is assumed that the table is first typeset to its natural width; should it appear too small, and should it be typeset at a larger width, for example by filling the total

`\linewidth` available at that specific point, then and only then the `tabular` environment is changed to `widetable`, Should the initial table be moderately larger than the `\linewidth`, than it might be shrinked to `\linewidth` with `widetable`, provided there are enough columns, and therefore delimiters, to be reduced in size. Of course it's impossible to typeset any table with any negative value of `\tabcolsep`; or better, it is possible, but the result in general is very messy.

In other words `widetable` should be used as a second resort, so as to correct some typesetting features not considered aesthetically acceptable.

The syntax for the use of the environment `widetable` is the same as that of the `tabular*` environment; the only difference is the name. Therefore one has to specify:

`\begin{widetable}{`⟨*width*⟩`}{`⟨*column descriptors*⟩`}`
⟨*line of cells*⟩`\\`
⟨*line of cells*⟩`\\`
...
⟨*line of cells*⟩`\\`
⟨*line of cells*⟩`\\`
`\end{widetable}`

4 The method

The principle on which this little package is based is the following; suppose a certain table is typeset with an inter column glue $t_0 = 0$ and that its width turns out to be l_0; suppose the same tabular material is typeset again with an inter column glue $t_1 > 0$ so that the table table gets as large as $l_1 > l_0$. Then, if the table has to be as wide as l the inter column glue must equal the value

$$t = \frac{l - l_0}{l_1 - l_0} \cdot t_1$$

Therefore we need to run the typesetting of the same tabular material with the two values of the inter column glue set to zero and to t_1, respectively, so as to find the widths l_0 and l_1. Afterwards it has to determine the correct final value t, and typeset once again the same tabular material for the last time. Of course the first two runs must put their results into suitable boxes so as to avoid outputting them into the output file, while at the same time allowing to record the width of the enclosing boxes.

5 The long division algorithm

The only simple equation the algorithm must compute consists in evaluating the difference of of two lengths (a computation that is perfectly feasible with the available simple TEX primitive commands); a fractional division (not feasible with any TEX primitive command), and finally into multiplying this division result by

the only non zero inter column glue (another simple task to be done with TEX primitive commands).

I have tried several algorithms for computing the fractional result of the division of two lengths; unfortunately no one guarantees a minimum of precision with any sized operands; overflows and similar "accidents" are very common. Iterative algorithms are difficult to initialize; scaling the operands give a good chance of getting acceptable results, but in one way or another I always found some drawbacks. Therefore I decided to program the so called "long division" algorithm preceded by a number of tests in order to avoid spending time in vanishing results, and, even more important, to waste time in getting overflows that cause an abnormal termination of the typesetting program.

Of course there is no limit to a better solution; nevertheless the one I implemented never crashed in any real world situation I tested.

6 Acknowledgements

I must deeply thank Enrico Gregorio for the revision of this package macros and for his wise suggestions about the correct programming style. If some glitch still remains in the programming style, that is just my fault.

7 Implementation

the first thing to do is to globally define a certain number of TEX dimensions and counters; these dimension and counter registers are selected among the first even numbered ones, as our Grand Wizard suggested in the TEXbook.

Actually I'd prefer to define such registers within the group of the division algorithm, so as not to mess up anything that might be used by other macros, but I accepted the suggestion of Enrico Gregorio, about the programming style and I left these register definitions in a global position, instead of a local group position.

Another point that initially I had solved in a different way was to use register numbers over the value 255, the maximum that good old TEX could handle. Now the typesetting/interpreter program pdftex embeds all the extensions introduced with the former ε-TEX program; now the numbering of the registers can go up to $2^{15} - 1$, and there is enough choice for any numbering. But it may be argued that LATEX users do not upgrade their software so often, while there are some situations where the use of obsolete versions must be still preferred (I can't imagine any, but they assure me that there are some). Therefore Enrico correctly suggests to use the scratch even numbered registers (Knuth's suggestion, although Knuth excluded the counter registers from this statement, being the first 10 counters reserved for complicated page numbering applications).

```
\dimendef\wt@Numer=2
\dimendef\wt@Denom=4
\countdef\wt@Num=2
\countdef\wt@Den=4
\countdef\wt@I=6
```

```
\def\wt@segno{}
```

We then start the definition of the division algorithm; the name of the macro and the separators of the delimited arguments are in Italian, thus minimizing the risk of colliding with macros of other packages. "dividi...per...in..." means "divide...by..., to..."; the first "..." represent the dividend, the second "dots" represent the divisor (both are lengths), while the third "..." represent the quotient (a signed fractional decimal number).

The first operations performed on the operands are to copy them into named dimension registers; the named registers make the programming a little easier, in the sense that the chosen names have a meaning and their contents should conform to that meaning.

Then the sings of the register operands are checked and pssibly changed so as to work with positive values; the overall result sign is memorized into a named macro (all macros are named, but here the name conforms to its contents "segno" maps to "sign"). Afterwards the zero value of the denominator is tested; if the test is true the result assigned to the internal quotient macro `\wt@Q` is the signed dimensional "infiniy", that in TEX and LATEX is equal to $2^{30}-1$ scaled points; this value is assigned by the format to the kernel dimension register `\maxdimen`, so we need just use this name, instead of assigning strange numerical values; the only thing we must pay attention to is to strip the "pt" information from this "infinite" dimension, since the quotient must be a dimensionless signed fractional decimal number.

Otherwise we load the operands in similarly named counter registers, effectively transferring the dimension integer number of scaled points to integer variables over which we continue our operations.

We compute by the primitive TEX integer division command the integer part of the quotient and we assing its expanded decimal value, followed by a decimal point, to the temporary internal quotient. Getting back to dimensions, we compute the remainder of the numerator minus the quotient times the denominator in terms of lengths. We locally set the number of iterations `\wt@I` to six, and then we call the iterative algorithm of the long division within a `\@whilenum...\do` cycle.

At the exit of this cycle the internal quotient `\wt@Q` contains all the digits of the integer and the fractional part of the result.

Now comes the interesting part: we are within a group and we must "throw" the quotient outside the group, but hopefully we would not like to leave something behind; we then define an expanded macro that contains the unexpanded `\endgroup` so that when we execute that macro, it is this very action that closes the group and at the same time, in spite of having been started within it, it keeps being executed bringing outside the definition of the external quotient that will be executed outside the group with the expanded value of the internal quotient. When `\x` is finished it does not exist any more as well as any value that was assigned or defined within the group.

```
\def\dividi#1\per#2\in#3{%
  \begingroup
  \wt@Numer #1\relax \wt@Denom #2\relax
```

```
  \ifdim\wt@Denom<\z@ \wt@Denom -\wt@Denom \wt@Numer -\wt@Numer\fi
  \ifdim\wt@Numer<\z@ \def\wt@segno{-}\wt@Numer -\wt@Numer\fi
  \ifdim\wt@Denom=\z@
    \edef\wt@Q{\ifdim\wt@Numer<\z@-\fi\strip@pt\maxdimen}%
  \else
    \wt@Num=\wt@Numer \wt@Den=\wt@Denom \divide\wt@Num\wt@Den
    \edef\wt@Q{\number\wt@Num.}%
    \advance\wt@Numer -\wt@Q\wt@Denom \wt@I=6
    \@whilenum \wt@I>\z@ \do{\wt@dividiDec\advance\wt@I\m@ne}%
  \fi
  \edef\x{\noexpand\endgroup\noexpand\def\noexpand#3{\wt@segno\wt@Q}}
  \x
}
```

The cycle for the long division consists in multiplying the remainder in `\wt@Numer` by ten, then reassigning the dimension value to the numerator integer counter so as to determine a new digit of the quotient in `\wt@q`. After this, this digit is appended by means of an expanded definition of the internal quotient, but it is used also for determining the new remainder in the `\wt@Numer` dimension register. Since the iteration is performed six times, six fractional digits are determined by this procedure, probably one digit too many, but its better one too many than the opposite.

```
\def\wt@dividiDec{%
  \wt@Numer=10\wt@Numer \wt@Num=\wt@Numer \divide\wt@Num\wt@Den
  \edef\wt@q{\number\wt@Num}\edef\wt@Q{\wt@Q\wt@q}%
  \advance\wt@Numer -\wt@q\wt@Denom}
```

Now we define the dimension register that is to contain the desired table width. We further define the start of the tabular typesetting that will be useful in a while. Actually the table preamble is being saved into a macro, so that when the ⟨*width*⟩ and the ⟨*column descriptors*⟩ are given to the opening environment statement, these saved quantities can be used again and again.

```
\newdimen\wt@width
\def\wt@starttabular{\expandafter\tabular\expandafter{\wt@preamble}}
```

The environment opening as well as the environment closing are defined by means of low level commands. Due to the syntax of the opening command that requires two compulsory arguments, these are saved in the recently defined dimension register and to a macro respectively; anotehr macro `\wt@getTable` is used to get the body of the table; the `\end{widetable}` statement represents the ending delimiter of the table contents.

```
\def\widetable#1#2{%
  \def\@tempC{widetable}\setlength{\wt@width}{#1}%
  \def\wt@preamble{#2}\wt@getTable}
```

A new boolean, `wt@scartare`, is defined; this boolean variable will be set true in order to detect if the table body is is not well formed, with `\begin` and `\end` statements tha don't match, and the like; actually the `widetable` environment can contain other environmente, even another `widetable` environment, but the external one should not be upset by the internal ones. In order to achieve this result,

it is necessary that any embedded environment is hidden int a group delimited by a pair of matching braces.

```
\newif\ifwt@scartare\wt@scartarefalse
```

The closing statement will acttually do the greatest part of the job. First of all if the above mentioned boolean variable is true, it skips everyting and it does not set any table; but if the boolean variable is false, the table body is well formed and it can do the job as described in the previous sections. It first sets `\tabcolsep` to zero and sets the resulting table in box zero; the lower level `tabular` with the information saved into `\wt@startabular` and the body of the table contained into the token register zero.

Then it sets `\tabcolsep` to 1 cm (arbitrarily chosen) and typesets again the table into box two. The width of box zero is l_0 and that of box two is l_1; these are the lengths needed by the equation that evaluates the final typesetting glue. The arbitrary constant of 1 cm is t_1, and the specified width l is the dimension saved into `\wt@width`. The subtractions are operated directly on the dimension registers `\wt@width` (the numerator) and on the auxiliary register `\@tempdimenb`; the `\dividi` command is executed in order to get the quotient in `\@tempA`, and the final definitive value of `\tabcolsep` is eventually computed. The table is finally typeset without using boxes, while the contents of box zero and two are restored upon exiting the environment to any value they might have contained before entering `widetable`.

```
\def\endwidetable{%
  \ifwt@scartare
    \noindent\null
  \else
    \tabcolsep=\z@
    \setbox\z@=\hbox{\wt@starttabular\the\toks@\endtabular}%
    \tabcolsep=1cm\relax
    \setbox\tw@=\hbox{\wt@starttabular\the\toks@\endtabular}%
    \advance\wt@width-\wd\z@
    \@tempdimb=\wd\tw@
    \advance\@tempdimb-\wd\z@
    \dividi\wt@width\per\@tempdimb\in\@tempA
    \tabcolsep=\@tempA\tabcolsep
    \wt@starttabular\the\toks@\endtabular
  \fi
  \ignorespacesafterend
}
```

Of course other actions must be performed before executing the closing environment statement. We need a macro `wt@finetabella` that is equivalent to the ending environment statement.

```
\def\wt@finetabella{\end{widetable}}%
```

We finally can define the all important macro that gets the table body; it requires two delimited arguments: in `#1` the table body and, after the `\end` command, the closing environment name will be set in `#2`. The environment name is assigned to the macro `\@tempB`, which is checked against the correct name

`widetable` saved in the macro `\@tempC` by the opening command. If the names match, then the table body is assigned to the token register zero, to be used later on by the typesetting macros. But if the names don't match, then something went wrong and a package message is issued to explain what happened and how the program will manage the situation.

Specifically the names may not match if a cell contained another environment and its whole `\begin{...}...\end{...}` was not closed within a pair of matched braces. If an enclosed environment is hidden within a group, the delimited macro `\wt@getTable` will ignore such embedded environment, otherwise it will get a non matching name and messy things might happen. Besides warning about this fact, the body of the table, at least what has been read by the macro, will be discarded and substituted with a box containing a message; therefore a table will be typeset, but not the desired one. The remaining part of the body remains in the input stream and might cause, presumably, strange errors, such as `&` characters used outside a tabular or array environment. We must take care of this so that the typesetting procedure does not crash.

```
\def\wt@getTable#1\end#2{\def\@tempB{#2}%
  \ifx\@tempB\@tempC
    \toks@={#1}%
    \expandafter\wt@finetabella
  \else
    \PackageWarning{widetable}{%
      The table contains environment `\@tempB' %
      \MessageBreak
      not enclosed in braces. This is expressly forbidden!%
      \MessageBreak
      The table is not typeset  and is substituted%
      \MessageBreak
      with a framed box}
    \noindent\framebox[\wt@width]{The table was not typeset because
    it contains a visible \texttt{\char`\\end} in one or more cells.}\par
    \expandafter\wt@finishTable
  \fi
}
```

In order to avoid a complete mess, we have to iteratively gobble the rest of the input stream until a valid `\end{widetable}` is encountered; Actually the following macro will do a nice job in general, but it is not infallible if the input stream is really composed in a very bad way. In facts it calls itself again and again, always gobbling it arguments, until a valid terminating environment name matches the name `widetable`.

```
\def\wt@finishTable#1\end#2{%
  \def\@tempB{#2}%
  \ifx\@tempB\@tempC
    \wt@scartaretrue\expandafter\wt@finetabella
  \else
    \expandafter\wt@finishTable
  \fi
```

```
}
\endinput
```