The fcolumn package*

Edgar Olthof

Printed March 11, 2015

Abstract

In financial reports, text and currency amounts are regularly put in one table, e.g., a year balance or a profit-and-loss overview. This package provides the settings for automatically typesetting such columns, including the sum line (preceded by a rule of the correct width), using the specifier `f`.

The column specifier `f` itself is rather simple. It is the predefined version of a generic column `F`. The generic version expects three arguments: `#1` is the separator, `#2` is the decimal mark, and `#3` the coding used for grouping digits of the integer part and decimal part.

The f-column in the current version of the package is defined as `\newcolumntype{f}{F{.}{,}{3,2}}` (which is the continental European standard, implemented here) and people in the Anglo-saxon world would rather code `\newcolumntype{f}{F{,}{.}{3,2}}`. The default value for `#3` is 3,2, indicating that grouping of the integer part is by three digits and that the decimal part consists of two digits. If however, in your country or company grouping is done with a thinspace every four digits and there are three digits after the decimal mark—that happens to be a `\cdot`—, then simply specify `\newcolumntype{f}{F{\,}{\cdot}{4,3}}` in that case. By default two digits are used for the decimal part, so if you really want no decimal digits (in that case of course also skipping the decimal mark) you have to explicitly specify `x,0`.

1 Introduction

To show where and how the f-column is used, let's look at a typical financial table.

Balance sheet

properties	31 dec 2014	debts	31 dec 2014
house	200.000,00	equity capital	50.000,00
bank account	−603,23	mortgage	150.000,00
savings	28.000,00		
cash	145,85	profit	27.542,62
	227.542,62		227.542,62

*This file has version number v1.0, last revised 2015/03/07.

This table was entered as:

```
\begin{table}[htb]
\begin{tabular}{@{}lflf@{}}
\multicolumn4c{\bfseries Balance sheet}\\
\multicolumn1{@{}l}{properties}&\multicolumn1r{31 dec 2014}
&\multicolumn1l{debts}&\multicolumn1{l@{}}{31 dec 2014}\\
\hline
house &     20000000 & equity capital&  5000000\\
bank account& -60323 & mortgage &      15000000\\
savings&     2800000 &         & \leeg         \\
cash&          14585 & profit  &        2754262\\
\sumline
\check12
\hline
\end{tabular}
\end{table}
```

All the work was done by the column specifier "f" (for "finance"). In this case it constructs the `\sumline`, typesets the numbers, calculates the totals, and checks whether the two columns are in balance; if not, the user is warned via a `\message`. Of course for nice settings one should use the `booktabs` package, but that is not the point here.

This package is heavily inspired by the `dcolumn` package by David Carlisle, some constructions are more or less copied from that package.

There is at least one things to improve in this version: An empty entry in an f-column should result in an empty spot in the table. Now an empty entry is not correct and is repaired by LaTeX to be equal to 0, printing a number in the table. This can be prevented by explicitly specifying `\leeg`, like in the table above, but that should be automated.

2 The macros

`column F` `column f`

Note that the definition of the column type `f` does not use private macros (no `@`), so overriding its definition is easy for a user.

```
\newcolumntype{F}[3]{>{\b@fi{#1}{#2}{#3}}r<{\e@fi}}
\newcolumntype{f}{F{.}{,}{3,2}}
```

`\FCsc@l`

Two ⟨*count*⟩s are defined, that both start at zero: the ⟨*count*⟩ `\FCsc@l`, that keeps track at which f-column the tabular is working on and the ⟨*count*⟩ `\FCtc@l`, that records the number of f-columns that were encountered so far. Later in the package the code can be found for generating a new ⟨*count*⟩ and a new ⟨*dimen*⟩ if the number of requested f-columns is larger than currently available. This is of course the case when an f-column is used for the first time.

```
\newcount\FCsc@l \FCsc@l0
\newcount\FCtc@l \FCtc@l0
```

`\geldm@cro` The macro `\geldm@cro` takes a number and by default interprets this as an amount expressed in cents (dollar cents, euro cents, centen, Pfennige, kopecks, groszy) and typesets it as the amount in entire currency units (dollars, euros, guldens, Marke, ruble, złoty) with comma as decimal separator and the dot as thousand separator. As explained, this can be changed. It uses a private boolean `\withs@p` and an accessable, i.e., without `@` boolean: `\strictaccounting`. The latter is used to typeset negative numbers between parentheses. By default it doesn't do this: a minus sign is used. It also uses two private ⟨*count*⟩s. I'm still figuring out whether this is needed; everything is done local within a column of an `\ialign`, so using `\count0`, `\count1`, etc. might also work: it would safe two ⟨*count*⟩s.

```
\newcount\g@lda
\newcount\g@ldb
\newif\ifwiths@p
\newif\ifstrictaccounting \strictaccountingfalse
```

Actually `\geldm@cro` is only a wrapper around `\g@ldm@cro`.

```
\def\geldm@cro#1#2{\withs@pfalse
 \afterassignment\g@ldm@cro\count@#2\relax{#1}}
```

`\g@ldm@cro` This macro starts by looking at the sign of `#2`: if it is negative, it prints the correct indicator (a parenthesis or a minus sign), assigns the absolute value of `#2` to `\g@lda` and goes on. Note that `\geldm@cro` and therefore `\g@ldm@cro` are used within `$`s, so it is really a minus sign that is printed, not a hyphen.

```
\def\g@ldm@cro#1\relax#2{%
 \ifnum#2<0 \ifstrictaccounting(\else-\fi\g@lda=-#2 \else\g@lda=#2 \fi
```

Calculate the entire currency units: this is the result of x/a as integer division, with $a = 10^n$ and n the part of `#1` after the separator (if any). Here the first character of `#1` is discarded, so the separator in `#1` is not strict: you could also specify `3.2` instead of `3,2` (or even `3p2`).

```
 \ifx\relax#1\relax\g@ldb=2 \else \g@ldb\@gobble#1\relax\fi
 \loop
  \ifnum\g@ldb>0 \divide\g@lda by 10 \advance\g@ldb by \m@ne
 \repeat
```

The value in `\g@lda` is then output by `\g@ldens` using the separation given.

```
 \g@ldens{\the\count@}%
```

If there is a decimal part...

```
 \ifx\relax#1\relax\g@ldb=2 \else \g@ldb\@gobble#1\relax\fi
 \ifnum\g@ldb>0\decim@lmark
```

Next the decimal part is dealt with. Now x mod a is calculated in the usual way: $x - (x/a) * a$. The minus sign necessary for this calculation is introduced in the next line by changing the comparison to `>` instead of `<`.

```
  \ifnum#2>0 \g@lda=-#2\else\g@lda=#2 \fi
  \loop
   \ifnum\g@ldb>0 \divide\g@lda by 10 \advance\g@ldb by \m@ne
  \repeat
  \ifx\relax#1\relax\g@ldb=2 \else \g@ldb\@gobble#1\relax\fi
```

```
 \loop
  \ifnum\g@ldb>0 \multiply\g@lda by 10 \advance\g@ldb by \m@ne
 \repeat
 \ifnum#2>0 \advance\g@lda by #2
 \else \advance\g@lda by -#2
 \fi
 \ifx\relax#1\relax\g@ldb=2 \else \g@ldb\@gobble#1\relax\fi
 \zerop@d{\number\g@ldb}{\number\g@lda}%
\fi
```

If the negative number is indicated by putting it between parentheses, then the closing parenthesis should stick out of the column, otherwise the alignment of this entry in the column is wrong. This is done by an `\rlap` and therefore does not influence the column width. For the last column this means that this parenthesis may even stick out of the table. I don't like this, therefore I chose to put `\strictaccountingfalse`. Change if you like.

```
\ifnum#2<0 \ifstrictaccounting\rlap{)}\fi\fi}
```

`\g@ldens` Here the whole currency units are dealt with. The macro `\g@ldens` is used recursively, therefore the double braces; this allows to use `\count0` locally. Tail recursion is not possible here (at least I don't know of a way), but that is not very important, as the largest number $(2^{31} - 1)$ will only cause a threefold recursion. This also implies that the largest amount this package can deal with is 2.147.483.647 (using `x.0`). For most people this is probably more than enough if the currency is euros or dollars. If not, then it is more likely that you are spending your time on the beach than studying this package. And otherwise make clear that you use a currency unit of k$.

```
\def\g@ldens#1{{\g@ldb=\g@lda \count0=#1
```

First divide by 10^n, where n is `#1`.

```
 \loop
  \ifnum\count0>0
  \divide\g@lda by 10
  \advance\count0 by \m@ne
 \repeat
```

Here is the recursive part,

```
 \ifnum\g@lda>0 \g@ldens{#1}\fi
```

and then reconstruct the rest of the number.

```
 \count0=#1
 \loop
  \ifnum\count0>0
  \multiply\g@lda by 10
  \advance\count0 by \m@ne
 \repeat
 \g@lda=-\g@lda
 \advance\g@lda by \g@ldb \du@zendprint{#1}}}
```

`\du@zendprint` The macro `\du@zendprint` takes care for correctly printing the separator and possible trailing zeros.

```
\def\du@zendprint#1{\ifwiths@p\sep@rator\zerop@d{#1}{\number\g@lda}%
 \else\zerop@d{1}{\number\g@lda}\fi
 \global\withs@ptrue}
```

`\zerop@d` The macro `\zerop@d` uses at least `#1` digits for printing the number `#2`, padding with zeros when necessary. Initially this was a nice macro using tail recursion until it was found that the running time of that macro was proportional to n^2, where n is roughly the number of zeros to be padded. The worst case situation is when printing "0". The running time of the current version is only linear in n.

It is done within an extra pair of braces, so that `\count0` and `\count1` can be used without disturbing their values in other macros.

```
\def\zerop@d#1#2{{\count0=1 \count1=#2
```

First determine the number of digits of `#2` (expressed in the decimal system). This number is in `\count0` and is at least 1.

```
 \loop
  \divide \count1 by 10
  \ifnum\count1>0 \advance\count0 by 1
 \repeat
```

The number of zeros to be padded is $\max(0, \texttt{\#1-\textbackslash count0})$ (the second argument can be negative), so a simple loop suffices.

```
 \loop
  \ifnum\count0<#1 0\advance\count0 by 1
 \repeat
 \number#2}}
```

`\zetg@ld` This macro takes care for several things: it increases the total for a given f-column, it records the largest width of the entries in that column and it typesets `#1` via `\geldm@cro`.

```
\def\zetg@ld#1#2{\global\advance\csname
 FCtot@\romannumeral\FCsc@l\endcsname by #1
 \setbox0=\hbox{$\geldm@cro{#1}{#2}$}%
 \ifdim\wd0>\csname FCwd@\romannumeral\FCsc@l\endcsname
  \global\csname FCwd@\romannumeral\FCsc@l\endcsname=\wd0
 \fi\unhbox0}
```

`\check` The macro `\check` provides a way to the user to check that the appropriate columns are balanced (as it should in a balance). Arguments `#1` and `#2` are the column numbers to compare. It is the responsibility of the user to provide the correct numbers here, otherwise bogus output is generated. But the output is only to screen and the transcript file; it doesn't change the appearance of your document, so you can safely ignore the result then, repair and go on.

```
\def\check#1#2{\noalign{\ifnum\csname FCtot@\romannumeral#1\endcsname=
 \csname FCtot@\romannumeral#2\endcsname\else
 \message{Columns #1 and #2 do not balance!}\fi}}
```

`\b@fi` The macro `\b@fi` provides the beginning of the financial column. It will be inserted in the column to capture the number entered by the user. The `\let` is only local to the column and is necessary because `\ignorespaces` is automatically inserted by the preamble-generating macro `\@mkpream`. Its effect should be annihilated, otherwise the assignment to `\bedr@g` goes wrong. The plain LaTeX command `\@empty` is used for that. The separator and decimal mark are within a math environment, so you can indeed specify `\,` instead of `\thinspace`, but there is an extra brace around, so it doesn't affect the spacing between the digits (trick copied from `dcolumn`).

```
\newcount\bedr@g
\def\b@fi#1#2#3{\def\sep@rator{{#1}}\def\decim@lmark{{#2}}%
 \let\ignorespaces=\@empty \def\sp@l{#3}\global\advance\FCsc@l by \@ne
```

The value specified by the user is then captured by `\bedr@g`.

```
 \bedr@g=}
```

`\e@fi` Once captured, the rest is handled by `\e@fi`. The number is then fed to `\zetg@ld` for the current column.

```
\def\e@fi{\relax \zetg@ld{\number\bedr@g}{\sp@l}}
```

Here are adaptations to existing macros.

`\@xarraycr` Among all its tasks, now `\\` also resets the column number. This is a transparant change, i.e., loading this package without using the "f" column specifier does not change the effect of package `array.sty` in any way. This resetting has to be done after the `\cr`, so it is stored in a `\noalign`.

```
\def\@xarraycr{\@ifnextchar [%
 \@argarraycr {\ifnum 0=`{}\fi \cr\noalign{\global\FCsc@l=0 }}}
```

`\@array` The definition of `\@array` had to be extended slightly because it should also include `\@mksumline` (acting on the same `#2` as `\@mkpream` gets). Also this change is transparant: it only adds functionality and if you don't use that, you won't notice the difference.

```
\def\@array[#1]#2{%
 \@tempdima \ht \strutbox
 \advance \@tempdima by\extrarowheight
 \setbox \@arstrutbox \hbox{\vrule
            \@height \arraystretch \@tempdima
            \@depth \arraystretch \dp \strutbox
            \@width \z@}%
 \begingroup
 \@mkpream{#2}%
 \xdef\@preamble{\noexpand \ialign \@halignto
                 \bgroup \@arstrut \@preamble
                         \tabskip \z@ \cr}%
```

The combination `\endgroup` followed by `\begingroup` seems redundant, but that is not the case: the `\endgroup` restores everything that was not `\global`. With

the following `\begingroup` it is ensured that `\@mksumline` experiences the same settings as `\@mkpream` did.

```
\endgroup
\begingroup
\@mksumline{#2}%
\endgroup
```

As a side product of `\@mksumline` also the ⟨*count*⟩s for the totals and ⟨*dimen*⟩s for the widths of the colums are created. The columns should start fresh, i.e., totals are 0 and widths are 0 pt.

```
\res@tsumline
```

From here on it is just the old definition of `array.sty`.

```
\@arrayleft
\if #1t\vtop \else \if#1b\vbox \else \vcenter \fi \fi
\bgroup
\let \@sharp ##\let \protect \relax
\lineskip \z@
\baselineskip \z@
\m@th
\let\\\@arraycr \let\tabularnewline\\\let\par\@empty \@preamble}
```

Because `\@array` was changed here and it is this version that should be used, `\@@array` should be `\let` equal to `\@array` again.

```
\let\@@array=\@array
```

`\@mksumline` The construction of the sumline is much easier than that of the preamble for several reasons. It may be safely assumed that the preamble specifier is grammatically correct because it has already been screened by `\@mkpream`. Furthermore most entries will simply add nothing to `\sumline`, e.g., `@`, `!`, and `|` can be fully ignored. Ampersands are only inserted by `c`, `l`, `r`, `p`, `m`, and `b`. So a specifier like `@{}lflf@{}` will yield the sumline `&\a&&\a\\`, where `\a` is a macro that prints the desired result of the column. Had the specifier been `l|f||@{ }l|f`, then the same sumline must be constructed: all difficulties are already picked up and solved in the creation of the preamble.

In reality the sumline must be constructed from the expanded form of the specifier, so `@{}lf@{}` will expand as `@{}l>{\b@fi#1#2#2}r<{\e@fi}@{}`. The rules for constructing the sumline are now very simple:

- add an ampersand when `c`, `l`, `r`, `p`, `m`, or `b` is found, unless it is the first one (this is the same as in the preamble);
- add a `\a` when `<{\e@fi}` is found;
- ignore everything else.

To discriminate, a special version of `\testpach` could be written, but that is not necessary: `\testpach` can do all the work, although much of it will be discarded. Here speed is sacrificed for space and this can be afforded because the creation of the sumline is only done once per `\tabular`.

The start is copied from `\@mkpream`.

```
\def\@mksumline#1{\gdef\sumline{}
 \@lastchclass 4 \@firstamptrue
```

At first the column number is reset and the actual code for what was called `\a` above is made inactive.

```
\global\FCsc@l=\z@
\let\prr@sult=\relax
```

Then `\@mkpream` is picked up again.

```
\@temptokena{#1}
\@tempswatrue
\@whilesw\if@tempswa\fi{\@tempswafalse\the\NC@list}%
\count@\m@ne
\let\the@toks\relax
\prepnext@tok
```

Next is the loop over all tokens in the expanded form of the specifier. The change with respect to `\@mkpream` is that the body of the loop is now only dealing with classes 0, 2, and 10. What to do in those cases is of course different from what to do when constructing the preamble, so special definitions are created, see below.

```
\expandafter \@tfor \expandafter \@nextchar
 \expandafter :\expandafter =\the\@temptokena \do
 {\@testpach
 \ifcase \@chclass \@classfz
 \or \or \@classfii \or
 \or \or \or \or \or \or \or \@classfx \fi
 \@lastchclass\@chclass}%
```

And the macro is finished by appending the `\\` to the sumline.

```
\xdef\sumline{\sumline\noexpand\\}}
```

`\@addtosumline` Macro `\@addtosumline`, as its name already suggests, adds something to the sumline, like its counterpart `\@addtopreamble` did to the preamble.

```
\def\@addtosumline#1{\xdef\sumline{\sumline #1}}
```

`\@classfx` Class 10 for the sumline creation is a stripped down version of `\@classx`: add an ampersand unless it is the first. It deals with the specifiers `b`, `m`, `p`, `c`, `l`, and `r`.

```
\def\@classfx{\if@firstamp \@firstampfalse \else \@addtosumline &\fi}
```

`\@classfz` Class 0 is applicable for specifiers `c`, `l`, and `r` and if the arguments of `p`, `m`, or `b` are given. The latter three cases, with `\@chnum` is 0, 1, or 2 should be ignored and the first three cases are now similar to class 10.

```
\def\@classfz{\ifnum\@chnum<\thr@@ \@classfx\fi}
```

`\@classfii` Here comes the nice and nasty part. Class 2 is applicable if a `<` is specified. This is tested by checking `\@lastchclass`, which should be equal to 8. Then it is checked that the argument to `<` is indeed `\e@fi`. This check is rather clumsy but this was the first way, after many attempts, that worked. It is necessary because the usage of `<` is not restricted to `\e@fi`: the user may have specified other TEX-code using `<`.

```
\def\@classfii{\ifnum\@lastchclass=8
 \edef\t@stm{\expandafter\string\@nextchar}
```

```
\edef\t@stn{\string\e@fi}
\ifx\t@stm\t@stn
```

If both tests yield `true` then add the macro to typeset everything.

```
\@addtosumline{\prr@sult}
```

But we're not done yet: in the following lines of code the appropriate ⟨*count*⟩s and ⟨*dimen*⟩s are created, if necessary.

```
\global\advance\FCsc@l by \@ne
\ifnum\FCsc@l>\FCtc@l
```

Apparently the number of requested columns is larger than the currently available number, so a new one should be created. What is checked here is its existence: if it exists, it is not created by `fcolumn` so it may be something other than a ⟨*count*⟩, in which things go haywire. Therefore it is assumed that only `fcolumn` is allowed to generate a ⟨*count*⟩ called `\FCtot@<some romannumeral>`.

```
  \expandafter\ifx\csname FCtot@\romannumeral\FCsc@l\endcsname\relax
   \expandafter\newcount\csname FCtot@\romannumeral\FCsc@l\endcsname
  \else
   \message{\noexpand\FCtot@\romannumeral\FCsc@l already exists.}
  \fi
```

And the same is applicable for the ⟨*dimen*⟩: it is assumed that only `fcolumn` is allowed to generate a ⟨*dimen*⟩ called `\FCwd@<some romannumeral>`.

```
  \expandafter\ifx\csname FCwd@\romannumeral\FCsc@l\endcsname\relax
   \expandafter\newdimen\csname FCwd@\romannumeral\FCsc@l\endcsname
```

If the creation was successful, the ⟨*count*⟩ `\FCtc@l` should be increased.

```
   \global\FCtc@l=\FCsc@l
  \else
   \message{\noexpand\FCwd@\romannumeral\FCsc@l already exists.}
  \fi
 \fi
\fi}
```

Once created they are not initialised here because that is done later in one go.

`\leeg` The macro `\prr@sult` (see below) actually puts the information together. It starts with switching off the default behaviour of the f-column by providing the number 0 to comfort the assignment in `\b@fi`, although any number would be good: it is discarded anyway. This is achieved by `\let`ting `\e@fi` to be `\relax`. This `\let` is only local to the current column. Since the user may from time to time also need an empty entry in the table, this definition is without at-sign.

```
\def\leeg{0\relax \let\e@fi=\relax}
```

A better solution were to check for a number. If present then do the default action, otherwise generate an empty entry.

`\prr@sult` And here it is: the end result. First annihilate the effect of `\e@fi`.

```
\def\prr@sult{\leeg
```

Then the information for the last line is computed. It is not sufficient to calculate the width of the result (in points) to use that as the width of the rule separating the individual entries and the result. It may be that the sum is larger (in points) than any of the entries, e.g., when the result of $6 + 6$ is typeset. The width of the rule should be equal to the width of `\hbox{$12$}` then. On the other hand the width of the rule when summing 24 and -24 should be that of `\hbox{$-24$}` (or `\hbox{$(24$}`, see above), not the width of the result `\hbox{$0$}`. Therefore the maximum of all entry widths, including the result, was calculated.

```
\setbox0=\hbox{$\geldm@cro{\number\csname
FCtot@\romannumeral\FCsc@l\endcsname}{\sp@l}$}%
\ifdim\wd0>\csname FCwd@\romannumeral\FCsc@l\endcsname
  \global\csname FCwd@\romannumeral\FCsc@l\endcsname=\wd0
\fi
\vbox{\hbox to \csname
 FCwd@\romannumeral\FCsc@l\endcsname{\hrulefill}\vskip2pt
 \hbox to \csname
 FCwd@\romannumeral\FCsc@l\endcsname{\hfil\unhbox0}}}
```

`\res@tsumline` Since all changes to the totals and widths of the columns are global, they have to be reset actively at the start of a tabular or array. That is an action by itself, but it may occur more often, on request of the user, therefore a special macro is defined. The count `\FCsc@l` can be used as a temporary register because its actual value is not important anymore.

```
\def\res@tsumline{\FCsc@l\FCtc@l \loop\ifnum\FCsc@l>0
   \global\csname FCtot@\romannumeral\FCsc@l\endcsname=0
   \global\csname FCwd@\romannumeral\FCsc@l\endcsname=0pt
   \advance\FCsc@l by \m@ne \repeat}
```

`\resetsumline` To reset a sumline within a table, it should be done within a `\noalign`.

```
\def\resetsumline{\noalign{\res@tsumline}}
```

That's it!