The fcolumn package[*]

Edgar Olthof
`edgar <dot> olthof <at> inter <dot> nl <dot> net`

Printed July 25, 2023

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 and checking such columns, including the sum line (preceded by a rule of the correct width), using the specifier `f`.

1 Introduction

The package `fcolumn` provides the macros for an extra tabular specifier that makes creating financial tables easy. The column specifier `f` itself is rather simple; it is the predefined version of a generic column `F`. The generic version expects four arguments: 1) grouping character of the integer part on output, 2) decimal mark used on output, 3) compact additional information on input/output characteristics, and 4) anything, but primarily used for providing formatting information, see below.

The f-column in the current version of the package is defined for the continental European standard: `\newcolumntype{f}{F.,{3,2}{}}`. This means that a number like 12345,67 will be typeset as 12.345,67. People in the Anglo-Saxon world would rather code `\newcolumntype{f}{F,.{3,2}{}}` for the same input, yielding 12,345.67 as output for the number given above. The default value for `#3` is `3,2`, indicating that grouping of the integer part is by three digits, that a comma is used in the TEX-source to indicate the decimal mark, and that the decimal part consists of two digits. However, if in your country or company grouping is done with a thinspace every four digits, that the decimal mark in the source should be the character `p`, and there are three digits after the decimal mark—that happens to be a `\cdot`—, then simply specify `\newcolumntype{f}{F{\,}{\cdot}{4p3}{}}` in that case. The input could be 123456p78 then, yielding 12 3456·780 as output.

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`. If you want no grouping character, specify `0,x`.

[*]This file has version number v1.4.2, last revised 2023/07/25.

As the fourth parameter you can insert anything just before the typesetting of an amount in a column takes place. Its purpose is to add additional formatting information, e.g., `\color{blue}` to have the contents of a column coloured blue, but it can be misused, so use with care. And it can't do all!

Table 1: Example 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 package requires and loads the `array` package [1]. To show where and how the `F`-column is used, let's look at some typical financial information as shown in Table 1 and how this is entered in LaTeX (Table 2). All the work was done by

Table 2: Verbatim version of Example Table 1.

```
\begin{table}[htb]
\caption{Example Table.} \label{tab:ex1}
\begin{tabular}{@{}lflf@{}}
\multicolumn4{@{}c@{}}{\bfseries Balance sheet}\\
\toprule
properties & \leeg{31 dec 2014} & debts & \leeg{31 dec 2014}\\
\midrule
house        & 200000    & equity capital &  50000    \\
bank account &   -603,23 & mortgage       & 150000    \\
savings      &  28000                                \\
cash         &    145,85 & profit         &  27542,62 \\
\sumline
\bottomrule
\end{tabular}
\end{table}
```

the column specifier `f` (for "finance"). In this case it constructs the `\sumline`, typesets the numbers, calculates the totals, determines the widths of the sumrules, and checks whether the two columns are in balance; if not, the user is warned via a `\PackageWarning`. Of course for nice settings the `booktabs` package [2] was used, but that is not the point here.

This package is heavily inspired by the `dcolumn` package by David Carlisle [3]; some constructions are more or less copied from that package. From version 1.3 onwards it incorporates the idea of Christian Hoff of providing additional (formatting) information per column. A rather contrived example is given in Table 3, combining colour and fonts. How this is entered in LaTeX is shown in Table 4. The font changing commands like `\mathsf` and `\mathbf` act on an argument, hence

Table 3: Example Table with column formatting.

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

require braces, but these are already provided internally for this purpose. For that reason this type of commands must be given last, without braces (and if you don't specify a font changing command, these extra internal braces are just redundant). The fourth argument to the new columntype may consist of two parts, separated

Table 4: Almost verbatim version of Example Table 3.

```
\newcolumntype{q}[1]{F.,{3,2}{#1}}
\begin{table}[htb]
\caption{Example Table with column formatting.} \label{tab:ex3}
\begin{tabular}{@{}lq{\color{blue}\mathsf,\mathbf}lq{\color{magenta}}@{}}
\multicolumn4{@{}c@{}}{\bfseries Balance sheet}\\
...
... (Same financial contents as in Table 1.)
...
\end{tabular}
\end{table}
```

by a comma. In that case, the part to the left of the comma is applied to the data entered by the user and the right part, if non-empty, is the replacement formatting for the result. The example in Table 3 shows this: the bold font is only used in the `\sumline` and there is no colour specification, so that's back to the default (black). If you want formatting for the whole column, like magenta in the case of the last column of Table 3, leave out the comma. Font size changes, e.g., "`\huge`" in "`\huge\mathbf`" as parameter to column type `q` are ignored by LaTeX, since the formatting information is used in math environment, which has its own way of handling this. This isn't bad, as size changes in one column, without overall changes to the table look terrible. If you want something huge, make a `\huge` table.

Table 5: Table showing compatibility of fcolumn and longtable.

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 1	7.000,00		
savings 2	7.000,00		

(Table continues on next page)

Table 5: *(continued from previous page)*

properties	31 dec 2014	debts	31 dec 2014
savings 3	7.000,00		
savings 4	7.000,00		
cash	145,85	profit	27.542,62
	227.542,62		227.542,62

As is demonstrated in the preceding page break, the package `fcolumn` now also works with `longtable` [4], provided `longtable` is loaded before `fcolumn`; it checks for that. The raw formatting of the multipage table is shown in Table 6. For more

Table 6: Almost verbatim version of Example Table 5.

```
\begin{longtable}[l]{@{}lflf@{}}
\caption{\label{tab:ex5}Table showing compatibility of fcolumn and
  longtable.}\\
\multicolumn4{@{}c@{}}{\bfseries Balance sheet}\\
\toprule
properties & \leeg{31 dec 2014} & debts & \leeg{31 dec 2014}\\
\midrule
\endfirsthead
\caption[]{\textit{(continued from previous page)\/}}\\
\toprule
properties & \leeg{31 dec 2014} & debts & \leeg{31 dec 2014}\\
\midrule
\endhead
\bottomrule
\multicolumn4{@{}r@{}}{\small\textit{(Table continues on next page)\/}}\\
\endfoot
\bottomrule
\endlastfoot
house        & 200000    & equity capital &  50000    \\
...
... (Somewhat altered financial contents compared to
...  Table 1 to demonstrate the page break.)
...
\sumline
\end{longtable}
```

information on how to handle `\endhead` and its ilk, see the documentation of `longtable` [4]. Not shown here are the new `fcolumn` formatting possibilities (like new fonts and/or colours), but it has been checked they do work in combination with `longtable`. `fcolumn` also works happily together with `supertabular` [5], so you can choose which package you want if you need this. There is no race condition with `supertabular`: you can load it before or after `fcolumn`.

Unlike in the package `dcolumn` [3], the alignment on the decimal mark in `fcolumn` is achieved by right aligning numbers that all have the same number of digits to the right of this mark, in combination with the fact that all decimal glyphs have the same width*. This width is font specific, so changing fonts within a table may ruin the alignment, see e.g., the first `F`-column in Table 3.

2 Commands

The user only needs to know six commands or constructions. These six are given here.

`F` In the tabular the column specifier `F` can be given with arguments, or the predefined version `f`, where the four arguments of `F` are `{.}`, `{,}`, `{3,2}`, and `{}`. If you want `g` to be your own definition like the curious one given in Section 1, then specify `\newcolumntype{g}{F{\,}{\cdot}{4p3}{}}` prior to using `g` in a tabular.

Entries in an `F`-column are, from that moment on, treated as numbers unless explicitly escaped by `\leeg`, see below. The numbers are typeset according to the template the user gives with his/her `F`-column. The “middle” character of `#3` is an important switch: it does more than just setting the input decimal mark. By default the input grouping character is the dot, except when the dot is specified as input decimal mark; in that case the comma is acting as input grouping character. With this convention continental Europe and the Anglo-Saxon part of the world are served. This is hard coded, but using input grouping markers is optional anyway.

`\sumline` The numbers in an `F`-column are typeset as a financial amount, but the real benefit comes with the `\sumline`. It does three things:

1) It calculates and shows the total of the column so far and the maximum width encountered so far, including the width of the total;
2) It generates a rule with width calculated in the first item;
3) It checks the columns that are supposed to balance whether or not they actually do. If so, nothing happens. Otherwise a `\PackageWarning` is given that column i and j do not balance, where i and j are the relevant columns. This is only done if the total number of `F`-columns is even, e.g., if there are six `F`-columns, then 1 is checked against 4, 2 against 5, and 3 against 6. If the number of `F`-columns is odd then anything could be possible in that table and nothing is assumed about structure within the table. This behaviour can be overridden, see below.

By default the vertical separation between the rule and the total is 2 pt, but this can be changed by the optional argument to `\sumline`. Give, e.g., `\sumline[10pt]`, in case you want this spacing to be 10 pt. And you may even give two options, like in `\sumline[10pt][5pt]`, in which the second option is the extra space below the summary row. In fact that second option is parsed to `\\`, that is implicit in `\sumline`.

`\resetsumline` Suppose you want to typeset one tabular with the profit-and-loss of many projects individually. In case the layout of those tabulars is similar it were nice if all

* No common font is known in which this is not the case, but exceptions may exist.

columns were aligned. This can be done by making it one big tabular with a fresh start for each project. The macro `\resetsumline` is used for that: it resets all totals and all column widths, see for example Table 7. Note that the rules

Table 7: Example: multiple projects.

Project 1

expense	actual	budget	income	actual	budget
food	450,20	500,00	tickets	1.200,00	1.000,00
drinks	547,50	400,00			
music	180,00	100,00			
profit	22,30				
	1.200,00	1.000,00		1.200,00	1.000,00

Project 2

expense	actual	budget	income	actual	budget
food	250,00	300,00	tickets	400,00	450,00
drinks	100,00	80,00			
music	80,00	70,00	loss	30,00	
	430,00	450,00		430,00	450,00

in the first and third `F`-columns of Project 1 cover 1.200,00 whereas in Project 2 those rules are narrower since they only cover 430,00; still the columns are aligned (similar story for `F`-columns two and four). The verbatim way of setting up Table 7 is given in Table 8.

`\leeg` If an `F`-column should be empty then simply leave it empty. If however it should not be empty but the entry should be treated as text—even if it is a number—, this can be done with `\leeg`. It expects an argument and this argument is typeset in the column. The common case is where `p.m.` (*pro memoria*) is entered. In contrast to v1.1.2 of this package, now even an empty `F`-column followed by `\\` is allowed.

`\checkfcolumns` The automatic column balance check can also be done manually. If `F`-columns 1 and 4 should balance and you want them to be checked, then simply say `\checkfcolumns14`. With more than nine `F`-columns you may be forced to say something like `\checkfcolumns{10}{12}`. If `\checkfcolumns` is used, the automatic check is disabled. Multiple `\checkfcolumns`s are supported; if `F`-columns 1, 2, and 3 should balance, you specify `\checkfcolumns12` and `\checkfcolumns23`. There is no explicit command to disable all checking, but `\checkfcolumns11` obviously does the trick at the expense of some trivial calculations.

`\ifstrict@ccounting` `\ifminusr@d` In the rare occasion that a negative number occurs in a financial table, the sign of that number can be an explicit minus sign, i.e., '$-$' or the number is coloured red, or it is typeset between parentheses, and there may be even other ways. By default (for aesthetic reasons) `fcolumn` typesets it with a minus sign, but

Table 8: Verbatim version of Table 7.

```
\begin{table}[htb]
\caption{Example: multiple projects.} \label{tab:ex7}
\begin{tabular}{@{}lfflff@{}}
\multicolumn6{@{}c@{}}{\bfseries Project~1}\\
expense & \leeg{actual} & \leeg{budget} &
income  & \leeg{actual} & \leeg{budget} \\
\midrule
food   &  450,20 & 500    & tickets & 1200    & 1000    \\
drinks &  547,5  & 400                                   \\
music  &  180    & 100                                   \\
profit &   22,3                                          \\
\sumline[2pt][10pt]
\resetsumline
\multicolumn6{@{}c@{}}{\bfseries Project~2}\\
\toprule
expense & \leeg{actual} & \leeg{budget} &
income  & \leeg{actual} & \leeg{budget} \\
\midrule
food   &  250    & 300    & tickets &  400    &  450    \\
drinks &  100    &  80                                   \\
music  &   80    &  70    & loss    &   30              \\
\sumline
\bottomrule
\end{tabular}
\end{table}
```

strict accounting prescribes that the number should be put between parentheses. The latter can be accomplished by setting `\strict@ccountingtrue`, but since this contains a non-letter, it is also possible to invoke `fcolumn` with the option `strict`, i.e., `\usepackage[strict]{fcolumn}`, which sets this flag. If you want the negative numbers to be coloured red, use the option `red`, i.e., `\usepackage[red]{fcolumn}`, but note that `strict` and `red` are mutually exclusive: if both are used, `red` is reset because `strict` is strict.

3 The macros

Here follows the actual code.

3.1 Options

`option red`
`option strict`

There are two options. If `red` is set, negative numbers are displayed in red. If `strict` is set, strict accounting rules are used in display and this overrules a possibly set `red`. Depending on the two booleans associated with these options the `color` package [6] will be loaded or not.

```
\newif\ifminusr@d \minusr@dfalse
\newif\ifstrict@ccounting \strict@ccountingfalse
\DeclareOption{red}{\minusr@dtrue}
```

```
\DeclareOption{strict}{\strict@ccountingtrue}
\ProcessOptions \ifminusr@d \ifstrict@ccounting \minusr@dfalse
\PackageWarningNoLine{fcolumn}{Option ‘red’ is reset due to use
 of ‘strict’}\else\usepackage{color}\fi\fi
```

3.2 Definitions

column F
column f

The column specifier `F` is the generic one, and `f` is the default (continental European) one for easy use. 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}[4]{>{\b@fi{#1}{#2}{#3}{#4}}r<{\e@fi}}
\newcolumntype{f}{F.,{3,2}{}}
```

\FCsc@l
\FCtc@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@l=0 \newcount\FCtc@l \FCtc@l=0
```

\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, centimes, kopecks, groszy) and typesets it as the amount in entire currency units (dollars, euros, guldens, Marke, francs, rubles, złoty) with comma as decimal mark and the dot as grouping character (thousand separator if the first part of `#1` is 3). As explained, this can be changed. It uses three private booleans: `\ifwiths@p`, `\ifstrict@ccounting`, and `\ifminusr@d`. The latter two are used to typeset negative numbers in a special way. By default it doesn't do this: a minus sign is used.

```
\newif\ifwiths@p
```

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

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

\g@ldm@cro

After setting the environment for formatting, 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 `\count2` and goes on. Note that `\geldm@cro` and therefore `\g@ldm@cro` are always used within `$`s, so it is really a minus sign that is printed, not a hyphen. All calculations are done with `\count0`, `\count1`, etc. i.e., without `F`-column-specific ⟨*count*⟩s because it is all done locally. Leaving the tabular environment will restore their values. This is also true for the effect of `\FCform@t`, so that formatting information is local to this column. The reason for inserting the `{` between `\FCform@t` and `\ifnum` (and the accompanying `}` just before finishing this definition) is to facilitate the possible use of `\mathbf` or any other font changing command as the last item in `\FCform@t`.

When `\FCform@t` includes a colour change, this is overruled for negative numbers when option `red` is set. For that reason it is unwise to use a red colour for the whole column in combination with the option `red`: sign information is obscured then.

```
\def\g@ldm@cro#1\relax#2{\ifstrict@ccounting\def\bm@nus{(}
 \def\em@nus{)}\else\def\em@nus{}\ifminusr@d\def\bm@nus{\color{red}}
 \else\def\bm@nus{-}\fi\fi\FCform@t{\ifnum#2<0 \bm@nus\count2=-#2
 \else\count2=#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`).

```
\count4=\ifx\relax#1\relax 2 \else\@gobble#1\relax\fi
\count3=0
\loop\ifnum\count3<\count4
  \divide\count2 by 10 \advance\count3 by \@ne
\repeat
```

Note that `\count3` now equals `\count4`: this going up-and-down will be used more often, it saves several assignments. The value in `\count2` is then output by `\g@ldens` using the separation given (and stored in `\count@`).

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

If there is a decimal part...

```
\ifnum\count3>0 {\decim@lmark}
```

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

```
  \ifnum#2>0 \count2=-#2\else\count2=#2 \fi
  \loop\ifnum\count3>0
    \divide\count2 by 10 \advance\count3 by \m@ne
  \repeat
```

The value of `\count3` is now 0, so counting up again.

```
  \loop\ifnum\count3<\count4
    \multiply\count2 by 10 \advance\count3 by \@ne
  \repeat
  \ifnum#2>0 \advance\count2 by #2
  \else \advance\count2 by -#2
  \fi
  \zerop@d{\number\count3}{\number\count2}%
\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 `\strict@ccountingfalse`. Change if you like, by setting the option `strict`.

If overflow was detected, an exclamation mark is output to the right of the value that caused this. This of course ruins the appearance of the table, but in this case that serves a clear goal: there's something wrong and you should know.

```
\ifx\FCs@gn\m@ne\ifnum#2<0 \rlap{\em@nus~!}
 \else\rlap{\phantom{\em@nus}~!}\fi
\else\ifnum#2<0 \rlap{\em@nus}\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. This also implies that tail recursion is not possible here, but that is not very important, as the largest number (which is $2^{31}-1$) will only cause a threefold recursion using the default `3,2` (ninefold when using `1,0`, but who does that?). The largest amount this package can deal with is therefore 2.147.483.647 (using `3,0`). For most people this is probably more than enough if the currency is euros or dollars. And otherwise clearly state that you use a currency unit of k€ (or even M€ for the very rich).

There is no obvious interpretation of `#1` being zero or negative, therefore this is used as an indicator that no grouping character should be used.

```
\def\g@ldens#1{{\count3=\count2 \count0=#1
```

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

```
\ifnum\count0<1 \count0=3 \fi
\loop \ifnum\count0>0 \divide\count2 by 10 \advance\count0 by \m@ne
\repeat
```

Here is the recursive part,

```
\ifnum\count2>0 \g@ldens{#1}\fi
```

and then reconstruct the rest of the number.

```
\count0=#1
\ifnum\count0<1 \count0=3 \fi
\loop \ifnum\count0>0 \multiply\count2 by 10 \advance\count0 by \m@ne
\repeat
\count2=-\count2
\advance\count2 by \count3 \du@zendprint{#1}}}
```

`\du@zendprint` The macro `\du@zendprint` takes care for correctly printing the separator and possible trailing zeros. The former, however, is only done if `#1` is larger than zero.

```
\def\du@zendprint#1{\ifwiths@p\ifnum#1>0 {\sep@rator}\fi
 \zerop@d{#1}{\number\count2}%
\else\zerop@d1{\number\count2}\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. Note: `#1` being zero or negative is a flag that it should be interpreted as 3. A bit ugly, but it works, since the related code knows about this.

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 \@ne
\repeat
```

If #1 is positive, the number of zeros to be padded is max(0, #1-`\count0`) (the second argument can be negative), so a simple loop suffices. If it is zero or negative, this is a signal that it should be interpreted as 3 (and no separator will be output).

```
\ifnum#1>0
 \loop \ifnum\count0<#1\relax 0\advance\count0 by \@ne
 \repeat
\else
 \advance\count0 by -3
 \loop \ifnum\count0<0 0\advance\count0 by \@ne
 \repeat
\fi\number#2}}
```

`\zetg@ld` This macro takes care for several things: it increases the subtotal for a given F-column, it checks whether or not that subtotal has overflown, it records the largest width of the entries in that column and it typesets #1 via `\geldm@cro`.

```
\def\zetg@ld#1#2{\count0=#2\relax \let\FCs@gn=\@ne
```

First it checks whether there is a risk of overflow in this step. If A and B are two TEX-registers and B is to be added to A, overflow cannot occur if 1) one is (or both are) zero or 2) if A and B have different signs; otherwise be careful. Note that TEX does not check for overflow when performing an `\advance` (done in section 1238 of Ref. [7]), in contrast to `\multiply`, see section 105.

```
\ifnum\count0<0
  \ifnum\csname FCtot@\romannumeral\FCsc@l\endcsname<0
    \let\FCs@gn=\m@ne
  \fi
\fi
\ifnum\count0>0
  \ifnum\csname FCtot@\romannumeral\FCsc@l\endcsname>0
    \let\FCs@gn=\m@ne
  \fi
\fi
\global\advance\csname FCtot@\romannumeral\FCsc@l\endcsname by \count0
\ifx\FCs@gn\m@ne
```

They have the same sign, hence risk of overflow. Record the sign of `\count0` (and of the original total of this column; we just established they were the same) in `\FCs@gn`. Table 9 shows what can go wrong if the numbers are too large: in the left F-column the sumline is incorrect and the number that caused the overflow is indicated by an exclamation mark. In the middle F-column, overflow occurs twice and because this is once positive, once negative here, cancellation of errors occurs and the sumline is correct in the end. Nevertheless, it is advised to swap the two items that caused the overflow, as shown in the right F-column.

Table 9: Examples on overflow.

Projects

income	31 dec 2014	31 dec 2015	31 dec 2016
item 1	20.000.000,00	20.000.000,00	20.000.000,00
item 2	10.000.000,00 !	2.000.000,00 !	−1.500.000,00
item 3	5.000.000,00	−1.500.000,00 !	2.000.000,00
	−7.949.672,96	20.500.000,00	20.500.000,00

Check that the sign of the updated column total is still correct. If so, `\FCs@gn` is reset (to `\@ne`) at the end of this chunk.

```
  \ifnum\count0>0 \let\FCs@gn\@ne \fi
  \count0=\csname FCtot@\romannumeral\FCsc@l\endcsname
  \ifnum\FCs@gn<0 \count0=-\count0 \fi
  \ifnum\count0<0
    \let\FCs@gn=\m@ne
    \PackageError{fcolumn}{Register overflow}{Overflow occurred in
    fcolumn \number\FCsc@l\space near or at line \the\inputlineno.
    You can\MessageBreak press <enter> now and I'll proceed, but check
    your table.\MessageBreak The offending entry is indicated with an
    exclamation mark\MessageBreak in the output.}%
  \else\let\FCs@gn=\@ne
  \fi
\fi
```

The value of `\FCs@gn` is used in `\geldm@cro` below.

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

The ⟨*count*⟩s `\FC@l` and `\FC@r` capture the parts to the left and to the right of the decimal mark, respectively.

```
\newcount\FC@l \newcount\FC@r
```

Some auxiliary definitions for capturing compacted information.

```
\def\setucc@de#1#2\relax{\uccode`\~=`#1 }
\def\assignform@t#1,#2,#3\assignform@t{\def\FCform@t{#1}%
  \def\FCform@tt{#2}\ifx\FCform@tt\@empty \def\FCform@tt{#1}\fi}
```

`\m@thcodeswitch` As will be shown below, once the first digit, or sign, or decimal mark, or grouping character is scanned, the decimal digits should loose their activeness. That is done here for the digits in a rather blunt way, by putting their `\mathcode`s to zero if `#1` equals 0, since the actual `\mathcode` is not important—as long as it is not `"8000`—because the digits are not used for typesetting. (And even if they were; it's inside `\box0`, whose contents will be discarded.) When the `$` in `\e@fi` is encountered, the digits get back their original `\mathcode`s so that the actual typesetting in `\zetg@ld` is correct again. With a non-zero `#1` the activeness is switched on.

```
\def\m@thcodeswitch#1{\count0=10 \loop\ifnum\count0>0
\advance\count0 by \m@ne\mathcode\expandafter‘\the\count0=
\ifnum#1=0 0 \else "8000 \fi\repeat}
```

`\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 separator and decimal mark are within a math environment, so you can indeed specify `\,` instead of `\thinspace`, but there are extra braces around them in the code where they are used, so it doesn't affect the spacing between the digits (trick copied from `dcolumn`, Ref. [3]).

```
\def\b@fi#1#2#3#4{\sep@xt#1\sep@xt\def\decim@lmark{#2}\def\sp@l{#3}%
\assignform@t#4,,\assignform@t\global\advance\FCsc@l by \@ne
\global\FC@l=0 \global\FC@r=1
```

The value specified by the user is then captured by `\FC@l` and this is done in a special way: `\FC@l` is assigned globally within `\box0`. Why? To use it as scribbling paper to examine what the user entered, without dumping it into the horizontal list.

There are four parts to an `F`-column entry, all parts optional, making 16 combinations. The sequence is (in the Backus–Naur notation of Ref. [8]): ⟨sign⟩ ⟨integer constant⟩ ⟨decimal mark⟩ ⟨integer constant⟩. Here ⟨sign⟩ is a plus or minus character with category code 12, ⟨integer constant⟩ is a sequence of zero or more (decimal) ⟨digit⟩s, and ⟨decimal mark⟩ is the middel part of `#3`, i.e., the comma in `3,2` or the period in `3.2`. If the ⟨decimal mark⟩ is absent with no space characters between the two ⟨integer constant⟩ terms, these merge, making four redundant entries. One of the combinations is ⟨empty⟩, a sequence of characters outside the ones that are made active, e.g., plain text or nothing at all: this is the only combination that doesn't put anything in an `F`-column—and was the most difficult part to handle.

The minus sign must be captured separately, because in an entry like `-0,07` the 7 cents are negative, but this cannot be seen from the part to the left of the decimal mark, since -0 is 0 in TeX (in fact in most computer languages, but not in MIX [9]), so `\ifnum-0<0` yields `false`. `\FCs@gn` is a general purpose flag. Its first use is to capture the sign.

```
\let\FCs@gn=\@ne\relax \setbox0\hbox\bgroup$
```

Do the scan inside a box and inside math mode. Start with defining all characters that may appear as the first one in an `F`-column as active; digits first. This needs a complicated `\edef` with accompanying `\noexpand` because the starting digit for `\FC@l` must be packed inside each definition.

```
\count@=10 \loop\ifnum\count@>0 \advance\count@ by \m@ne
  \uccode‘\~=\expandafter‘\the\count@ \uppercase{\edef~}{\noexpand
  \m@thcodeswitch0 \global\FC@l=\the\count@}\repeat
```

For the input decimal mark something extra is needed: if it is the first character in an `F`-column (like in `,07`), it should also restore the `\mathcode`s of the digits. Checking whether or not it is the first is easy, since in that case the `\mathcodes` of the decimal digits are still `"8000`. The assignment to `\FC@r` starts with 1 (with no space following), so that appended digits get captured correctly, even if they start with 0; postprocessing of `\FC@r` is done in `\e@fi`. The input decimal mark switches itself off as active character, so at most one input decimal mark is allowed (N.B.: this makes sense).

```
\afterassignment\setucc@de\count@#3\relax
\uppercase{\def~}{\ifnum\mathcode`\0="8000 \m@thcodeswitch0 \fi
\afterassignment\deactdecm@rk\count@#3\relax \global\FC@r=1}%
```

The input grouping character effectively expands to “nothing, i.e., ignore” in a complicated way: it ignores the character and resumes scanning the number. The test prior to that action is needed if the grouping character is the first character encountered in the `F`-column. Which part to continue with depends on whether or not an input decimal mark was encountered; that can be checked by looking at its `\mathcode`.

The input grouping character is the dot “`.`”, except when that character was already chosen as input decimal mark. In that case, the grouping character will be the comma. This is easy to check because the `\uccode` of `` `\~ `` is still preserved.

```
\ifnum\uccode`\~=`. \uccode`\~=`,\relax\else \uccode`\~=`.\relax\fi
\uppercase{\def~}{\ifnum\mathcode`\0="8000 \m@thcodeswitch0 \fi
\afterassignment\d@cm\count@#3\relax
```

The `\expandafter` below is necessary because the global assignment should act after the `\fi`.

```
\ifnum\count@=\mathcode`- \expandafter\global\FC@l=\the\FC@l
\else \expandafter\global\FC@r=\the\FC@r\fi}%
```

The signs are relatively simple: record the sign, restore `\mathcode`s if needed (it should be: a minus sign between digits screws up everything), and start scanning the number.

```
\uccode`\~=`+\relax \uppercase{\def~}{\ifnum\mathcode`\0="8000
\m@thcodeswitch0 \fi\global\FC@l=0}
\uccode`\~=`-\relax \uppercase{\def~}{\ifnum\mathcode`\0="8000
\m@thcodeswitch0 \fi\global\let\FCs@gn\m@ne \global\FC@l=0}%
```

Now actually activate all these codes.

```
\mathcode`-="8000 \mathcode`+="8000 \mathcode`.="8000
```

These three remain active until the `$` in `\e@fi` is encountered. The following ones will, except in the ⟨empty⟩ case, have their activeness turned off at some time.

```
\m@thcodeswitch1 \afterassignment\actdecm@rk\count@#3\relax}
```

`\sep@xt` `\actdecm@rk` `\deactdecm@rk` `\d@cm` Again a few small macros to do important work. At first `\sep@xt` to extract the first character of `#1`, which in most cases will be the only character. Then `\actdecm@rk` and `\deactdecm@rk` to activate and deactive respectively the decimal mark. Finally, `\d@cm` captures a `\mathcode` for further investigation.

```
\def\sep@xt#1#2\sep@xt{\def\sep@rator{#1}}
\def\actdecm@rk#1#2\relax{\ifx#1.\relax \mathcode`,="8000
 \else \mathcode`#1="8000 \fi}
\def\deactdecm@rk#1#2\relax{\mathcode`#1=0 }%
\def\d@cm#1#2{\count@=\mathcode`#1 }
```

`\e@fi` If the digits are still active then either nothing, or only characters that did not deactivate the digits were entered. In both cases the output should be ⟨empty⟩. To flag this situation outside the group that started with the opening `$` of `\b@fi`, `\FC@r` is set globally to a negative value. This doesn't harm, because it didn't contain relevant information anyway. Outside the group, the sign of `\FC@r` can then be tested. This is a slight misuse of this `\count`, but now it's documented. In effect, `\FC@r` can only be −1, 1, or at least 10, so the comparison `\ifnum\FC@r>0` does not miss 0.

```
\def\e@fi{\ifnum\mathcode`\0="8000 \global\FC@r=\m@ne\fi$\egroup
\ifnum\FC@r>0
```

If there was no decimal mark or if there was a decimal mark but no decimal part, `\FC@r` will still be 1, which doesn't parse well with `\secd@xt`, so a zero is appended, i.e., yielding `10`.

```
  \ifnum\FC@r=1 \FC@r=10 \fi
```

Next is a loop for bringing the decimal part in the correct way to the integer part. This loop is performed the number of decimal digits to be printed (the 2 in `3,2` of the default setting). The higher this number is, e.g., 6 when `3,6` is chosen as `#3`—but who does that?—, the lower the maximum initial value for `\FC@l` can be, before a low-level TeX arithmetic overflow will occur, so don't overdo it.

```
\afterassignment\i@ts\count@\sp@l
  \loop\ifnum\count0>0 \multiply\FC@l by 10 \expandafter\secd@xt
  \number\FC@r\secd@xt \advance\count0 by \m@ne \repeat
```

This also means that if more decimal digits than this are provided, the excess digit(s) will not be handled. This is truncation, not rounding! If all truncated digits are zero, this truncation is exact and they are silently ignored, see the example Table 10, that was created with `@{}ldfl@{}` (`d` for centering on the decimal mark [3]) as tabular key. If, however, at least one of them is not zero, a `\PackageWarning` will be given, showing the discarded digit(s). There is nothing magical about the constant 19 below: it is simply the concatenation of `1` and the largest decimal digit. The smallest next value that `\FC@r` can have is 100, so all values from 19 up to and including 99 would have worked here.

Table 10: Truncating excess digits.

composer	raw entry	debt	remark
Berg	123,450	123,45	silently ignoring digit "0"
Eisler	234,563	234,56	warning: digit "3" ignored
Schönberg	345,6704	345,67	warning: digits "04" ignored
Webern	2,3456	2,34	warning: digits "56" ignored, i.e., without rounding this entry to 2,35
		706,02	

```
  \ifnum\expandafter\@gobble\number\FC@r>0
    \PackageWarning{fcolumn}{Excess digit\ifnum\FC@r>19 s\fi\space
    ``\expandafter\tw@l\number\FC@r\relax'' in decimal part
```

```
   \MessageBreak ignored near or}
 \fi
```

Don't forget to correct for the sign (once this is done, `\FCs@gn` is free again and can and will be used for other purposes). Then output the result.

```
 \ifx\FCs@gn\m@ne\relax\FC@l=-\FC@l\fi\zetg@ld{\sp@l}{\FC@l}%
\fi}
```

`\i@ts` `\tw@l` Macro `\e@fi` uses two very tiny macros, given here. In previous versions these were defined inside `\e@fi`, so they also got undefined when `\e@fi` ended. The current solution makes execution a bit faster.

```
\def\i@ts#1#2{\count0=#2} \def\tw@l#1#2\relax{#2}
```

`\secd@xt` The second digit from the left is needed from a string of characters representing a decimal number (that should be at least 10 and start with a `1`, but that is guaranteed by `\e@fi`). The macro `\secd@xt` extracts that digit, which is then added to `\FC@l`. A new number is assigned to `\FC@r`, that consists of the digits of `1#2`, unless `#2` was empty; in that case 10 is assigned. In this way `\FC@r` is prepared for insertion in the next invocation of `\secd@xt`. In iterating: 1234 yields 134, yields 14, yields 10, stays 10, etc.

```
\def\secd@xt1#1#2\secd@xt{\advance\FC@l by #1
 \FC@r=1#2 \ifnum\FC@r=1 \FC@r=10 \fi}
```

3.3 Adaptations to existing macros

`\@array` The definition of `\@array` had to be extended slightly because it should also include `\@mksumline` (acting on the same `#2` as `\@mkpream` gets). This change is transparant: it only adds functionality and if you don't use that, you won't notice the difference. It starts by just copying the original definition from v2.4k (or later) of the `array` package [1], compacted.

```
\def\@array[#1]#2{\@tempdima\ht\strutbox\advance\@tempdima by
 \extrarowheight\setbox\@arstrutbox\hbox{\vrule\@height\arraystretch
 \@tempdima\@depth\arraystretch\dp\strutbox\@width\z@}%
```

Here comes the first change: after each `\\` (or `\cr` for that matter) the ⟨*count*⟩ `\FCsc@l` should be reset. This is easiest done with `\everycr`, but `\everycr` is put to `{}` by `\ialign`, so that definition should change. The resetting should be done globally.

```
\def\ialign{\everycr{\noalign{\global\FCsc@l=0 }}\tabskip\z@skip\halign}
```

Then the definition is picked up again.

```
\begingroup\@mkpream{#2}\xdef\@preamble{\noexpand\ialign\@halignto
\bgroup\@arstrut\@preamble\tabskip\z@\cr}%
```

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

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

Much of the techniques here are repeated in `\LT@array`, see Section 3.6.

3.4 The sumline, close to a postamble

`\@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 `\s@ml@ne`, e.g., `@`, `!`, and `|` can be fully ignored. Ampersands are only inserted by `c`, `l`, `r`, `p`, `m`, and `b`. So, if `\a` is a macro that prints the desired result of the column (see below), then a specifier like `@{}lflf@{}` will yield the sumline `&\a&&\a\\`. 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.,{3,2}{}}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;
- close with a `\\`.

(For completeness' sake it should be mentioned that prior to the `\\` also the column check is inserted, see `\aut@check`.) To discriminate, a special version of `\@testpach` [1] 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 done only once per `tabular` or `longtable`.

The start is copied from `\@mkpream`.

```
\def\@mksumline#1{\gdef\s@ml@ne{}\@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=0 \let\prr@sult=\relax
```

Then `\@mkpream` is picked up again.

```
\@temptokena{#1}\@tempswatrue\@whilesw\if@tempswa\fi{\@tempswafalse
\the\NC@list}\count0\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 F-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 applying the `\aut@check` and appending the `\\` to the sumline. Note that the `\aut@check` is performed *in* the last column, but since it does not put anything in the horizontal list—it only writes to screen and transcript file—, this is harmless.

```
\xdef\s@ml@ne{\s@ml@ne\noexpand\aut@check\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\s@ml@ne{\s@ml@ne #1}}
```

`\@classfx` Class f10 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 f0 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 f10.

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

`\@classfii` Here comes the nice and nasty part. Class f2 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 LaTeX-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`, i.e., we encountered a `<{\e@fi}` where we expect one to find, 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. Note that `\FCsc@l` was set to 0 in the beginning of `\@mksumline`, so it is well-defined when `\@classfii` is used.

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

Apparently the number of requested columns is larger than the currently available number of relevant ⟨*count*⟩s and ⟨*dimen*⟩s, so new ones should be created. What is checked here is merely the existence of `\FCtot@<some romannumeral>`. If it already exists—although it may not even be a ⟨*count*⟩; that cannot be checked—it is not created by `fcolumn` and an error is given. In case it is a ⟨*count*⟩ you're just lucky, and you could ignore that error, although any change to this ⟨*count*⟩ is global anyway, so things will be overwritten. In the case it is not a ⟨*count*⟩, things will go haywire and you'll soon find out. The remedy then is to rename your ⟨*count*⟩ prior to `fcolumn` to avoid this name clash.

```
    \expandafter\ifx\csname FCtot@\romannumeral\FCsc@l\endcsname\relax
     \expandafter\newcount\csname FCtot@\romannumeral\FCsc@l\endcsname
    \else
     \PackageError{fcolumn}{Name clash for <count>}{\expandafter\csname
     FCtot@\romannumeral\FCsc@l\endcsname is already defined and it may
     not even be a <count>. If you're\MessageBreak sure it is a <count>,
     you can press <enter> now and I'll proceed, but things\MessageBreak
     will get overwritten.}%
    \fi
```

And the same is applicable for the ⟨*dimen*⟩: in case of a name clash you have to rename your ⟨*dimen*⟩ prior to `fcolumn`.

```
    \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
     \PackageError{fcolumn}{Name clash for <dimen>}{\expandafter\csname
     FCwd@\romannumeral\FCsc@l\endcsname is already defined and it may
     not even be a <dimen>. If you're\MessageBreak sure it is a <dimen>,
     you can press <enter> now and I'll proceed, but things\MessageBreak
     will get overwritten.}%
    \fi
   \fi
  \fi
\fi}
```

Once created, it is not necessary to initialise them here because that is done later in one go.

`\sumline` The command for the sumline has one optional argument: the separation between the rule and the total. By default this is 2 pt, but the user may specify `\sumline[10pt]` if that separation needs to be 10 pt. The assignment needs to be global, because it is done in the first column of the tabular, but is valid for the whole line.

```
\newdimen\s@mlinesep
\def\sumline{\@ifnextchar[\s@mline{\s@mline[2pt]}}
\def\s@mline[#1]{\global\s@mlinesep=#1 \s@ml@ne}
```

In the introduction it was stated that `\sumline` has two options, but in reality that second option is the option to `\\` that is issued by `\s@ml@ne`.

`\prr@sult` The macro `\prr@sult` actually puts the information together. It starts like `\leeg`.

```
\def\prr@sult{$\egroup \let\e@fi=\relax \let\FCform@t=\FCform@tt
```

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 happen that the sum is wider (in points) than any of the entries, e.g., when the result of $6+6$ (using specifier `3,2`) is typeset. The width of the rule should be equal to the width of `\hbox{$12{,}00$}` then. On the other hand the width of the rule when summing 24 and -24 should be that of `\hbox{$-24{,}00$}` (or `\hbox{$(24{,}00$}`, see above), not the width of the result `\hbox{$0{,}00$}`. Therefore the maximum of all entry widths, including the result, was calculated. This excludes the extension to the right in case parentheses are used, again for aesthetic reasons.

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

3.5 Other checks

`\leeg` This macro is used to overrule the default behaviour of the pair `\b@fi` and `\e@fi`. It starts with ending the groups in the same way that `\e@fi` would normally do. Then the effect of `\e@fi` (that is still in the preamble) is annihilated by `\let`ting it to be `\relax`. This `\let` is only local to the current column. Then the argument to `\leeg` is processed in the normal way for a right aligned column.

Since the user may from time to time also need a column entry other than a number in the table, e.g., `\leeg{p.m.}`, this definition is without at-sign. By defining `\leeg` in this way, instead of `\multicolumn1r{}` (which contains `\omit`), the default spacing in the column is retained. It doesn't alter the width of the sumrule, but has its normal effect on the column width, so be careful: don't insert the unabridged version of Romeo and Juliet [10] here. It is not typeset in math mode, nor does it use the extra (formatting) information of `#4` of the fcolumn, so you're completely free here.

```
\def\leeg#1{$\egroup \let\e@fi=\relax #1}
```

Note that anything may be given as argument to `\leeg`, so in principle it can also be used to cheat: `\leeg{0,03}` will insert the text `0,03` in the table but it doesn't increase the totals of that column by 3 (assuming `3,2` coding for the separations). But you won't cheat, won't you?

`\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. A side effect of this macro is that `\FCsc@l` is reset to 0. This is an advantage: it should be zero at the beginning of a line in the table (for other lines this is done by the `\\`).

```
\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=\z@
 \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}}
```

`\aut@check` If the number of F-columns is even, it is assumed that they are part of two sets of columns of which each column of the first set should balance the appropriate column of the second set. If on the other hand the number of columns is odd, then at least one column has nothing to balance against and no checking occurs. It is correct to check for oddness of `\FCsc@l` since this `\aut@check` is only performed in the last column of the tabular: the value of `\FCsc@l` now equals the number of columns used in the current tabular (and may be less than `\FCtc@l`).

The output is only to screen and the transcript file; it doesn't change the appearance of your document, so in case the assumption is wrong you can safely ignore the result and go on. The ⟨*count*⟩s 0 and 1 are used here and this can be done because any content of those ⟨*count*⟩s from previous calculations has become irrelevant at this moment.

If the list `\FC@chklist` is empty, the list for the automatic check is generated (which will remain empty if `\FCsc@l` is odd).

```
\def\FC@chklist{}
\def\aut@check{\ifx\@empty\FC@chklist\relax
 \ifodd\FCsc@l\else
  \count0=\@ne \count1=\FCsc@l \divide\count1 by \tw@
  \loop\ifnum\count1<\FCsc@l \advance\count1 by \@ne
   \xdef\FC@chklist{\FC@chklist\number\count0,\number\count1;}%
   \advance\count0 by\@ne \repeat
 \fi
\fi
```

Then this list is peeled off and processed. The comparison is done between the internal representation of the totals, so this only makes sense when the decimal part, i.e., the `2` of `3,2` in the default value for `#3` of `f` is the same for both columns. That is almost always the case due to the nature of this type of tabulars. The ⟨*count*⟩ `\FC@l` is free to be used for the calculations needed here.

```
\loop\ifx\FC@chklist\@empty\else
 \expandafter\fre@t\FC@chklist\fre@t
 \FC@l=\csname FCtot@\romannumeral\count0\endcsname
 \advance\FC@l by -\csname FCtot@\romannumeral\count1\endcsname
 \ifnum\FC@l=0 \else \ifnum\FC@l<0 \FC@l=-\FC@l \fi
  \PackageWarning{fcolumn}{Representations of F-columns \number\count0
  \space and \number\count1 \space differ by\MessageBreak
  \number\FC@l\space due to \string\sumline\space near or}%
 \fi
\repeat}
```

When `\aut@check` is finished, `\FC@chklist` is empty again, i.e., well prepared for the next time it is used. This also means that the default behaviour kicks in again: if that's not what you want, you should specify the appropriate `\checkfcolumns` lines again.

`\fre@t` This function eats the first two numbers off `\FC@chklist`.

```
\def\fre@t#1,#2;#3\fre@t{\count0=#1 \count1=#2 \xdef\FC@chklist{#3}}
```

`\checkfcolumns` But the assumptions for `\aut@check` may be wrong, therefore manual control on this checking is also made possible here. The macro `\checkfcolumns` 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 `F`-column numbers to compare. It is the responsibility of the user to provide the correct numbers here, otherwise bogus output is generated. If this manual check is inserted, the automatic check will not be performed.

```
\def\checkfcolumns#1#2{\noalign{\xdef\FC@chklist{\FC@chklist #1,#2;}}}
```

3.6 Support for multipage tables

Packages `longtable` [4] and `supertabular` [5] can be used for tables that span multiple pages. Package `supertabular` works with `fcolumn` out of the box (no changes needed), but may lead to different column widths on individual pages. If you don't want that, use `longtable`. The packages `fcolumn` and `longtable` also work together provided `longtable` is loaded first, so that `fcolumn` can adapt one definition of `longtable`. It's a long definition, so if `longtable` is not loaded, it's a waste of memory having it; that's why it is only defined when necessary. For that reason the user is politely warned (not loading `longtable` is obviously not an error if you don't use it) if `fcolumn` is loaded without prior loading of `longtable`.

```
\ifx\longtable\@undefined
\PackageWarningNoLine{fcolumn}{fcolumn is loaded without package
longtable.\MessageBreak That's perfectly OK, but if you want to
load\MessageBreak longtable as well, make sure it is done before
\MessageBreak loading fcolumn}\else
```

`\LT@array` And here is the only definition of `longtable` that needs to be extended to make `fcolumn` work with that package. The lines are compacted a bit w.r.t. the original `longtable` code; if you want to study the code, have a look at the documentation [4].

```
\def\LT@array[#1]#2{\refstepcounter{table}\stepcounter{LT@tables}\if
 l#1 \LTleft\z@\LTright\fill\else\if r#1 \LTleft\fill\LTright\z@\else
 \if c#1 \LTleft\fill\LTright\fill\fi\fi\fi\let\LT@mcol\multicolumn
 \let\LT@@tabarray\@tabarray\let\LT@@hl\hline\def\@tabarray{\let
 \hline\LT@@hl\LT@@tabarray}\let\\\LT@tabularcr\let\tabularnewline\\
 \def\newpage{\noalign{\break}}\def\pagebreak{\noalign{\ifnum`}=0\fi
 \@testopt{\LT@no@pgbk-}4}\def\nopagebreak{\noalign{\ifnum`}=0\fi
 \@testopt\LT@no@pgbk4}\let\hline\LT@hline\let\kill\LT@kill\let\caption
 \LT@caption\@tempdima\ht\strutbox\let\@endpbox\LT@endpbox\ifx
```

```
  \extrarowheight\@undefined\let\@acol\@tabacol\let\@classz\@tabclassz
  \let\@classiv\@tabclassiv\def\@startpbox{\vtop\LT@startpbox}\let
  \@@startpbox\@startpbox\let\@@endpbox\@endpbox\let\LT@LL@FM@cr
  \@tabularcr\else\advance\@tempdima\extrarowheight\col@sep\tabcolsep
  \let\@startpbox\LT@startpbox\let\LT@LL@FM@cr\@arraycr\fi\setbox
  \@arstrutbox\hbox{\vrule\@height\arraystretch\@tempdima\@depth
  \arraystretch\dp\strutbox\@width\z@}\let\@sharp##\let\protect\relax
  \begingroup\@mkpream{#2}\xdef\LT@bchunk{\global\advance\c@LT@chunks
  \@ne\global\LT@rows\z@\setbox\z@\vbox\bgroup\LT@setprevdepth\tabskip
  \LTleft\noexpand\halign to\hsize\bgroup\tabskip\z@\@arstrut\@preamble
  \tabskip\LTright\cr}
```

Until this line it was just the code for `\LT@array` from package `longtable`. The two lines of the next chunk are new to `\LT@array`. Their purpose is the same as in `\@array` above.

```
  \@mksumline{#2}\endgroup\res@tsumline
  \everycr{\noalign{\global\FCsc@l=0 }}%
```

From here on `\LT@array` is picked up again, ending with the `\fi` that belongs to the `\ifx` that started this definition.

```
  \expandafter\LT@nofcols\LT@bchunk&\LT@nofcols\LT@make@row\m@th\let
  \par\@empty\lineskip\z@\baselineskip\z@\LT@bchunk}
\fi
```

That's it!

Acknowledgement

Thanks to Karl Berry for valuable comments regarding the consistency of the installation procedure of this version. Frank Mittelbach gave various useful suggestions for improving the input parsing as well as hints to make the package more LaTeX-like. He also challenged me to make `fcolumn` compatible with `longtable`. Christian Hoff's request on column formatting triggered many happy hours of coding.

References

[1] Frank Mittelbach and David Carlisle. A new implementation of LaTeX's `tabular` and `array` environment.
[2] Simon Fear. The `booktabs` package. Publication quality tables in LaTeX.
[3] David Carlisle. The `dcolumn` package.
[4] David Carlisle. The `longtable` package.
[5] Theo Jurriens and Johannes Braams. The `supertabular` package.
[6] David Carlisle. The `color` package.
[7] Donald Knuth, *Computers & Typesetting/B, "TeX: the program,"* Addison-Wesley, Reading (1991).

[8] Donald Knuth, *Computers & Typesetting/A, “The TEXbook,”* Addison-Wesley, Reading (1991).
[9] Donald Knuth, *The Art of Computers Programming, volume 1, “Fundamental Algorithms,”* Addison-Wesley, Reading (1997).
[10] William Shakespeare, Romeo and Juliet, a tragedy (1597).

Change History

v0.1
General: First working version. . . . 1

v1.0
General: Three-argument version is working properly. 1

v1.1
General: Automatic checking of column balance performed when number of F-columns is even (behaviour can be overridden). Empty entries are now recognised and correctly treated as such, except for the one ended by the double backslash. Not serious; workaround possible. Furthermore optimisation of code: minimised the number of private counts and resetting of column counter done in a nicer way. 1

v1.1.1
General: Installation procedure changed from .ins-in-.dtx to separate .ins and .dtx after discussion with Karl Berry, as well as some minor code improvements. 1

v1.1.2
General: Some inconsistencies between explanatory text and actual code removed. 1

v1.2
General: Input parsing changed after comment from Frank Mittelbach. He (Frank) also gave various suggestions for improving robustness or user friendliness of this package. This version is only backwards compatible when zero decimal digits were and are specified as modifier. 1

v1.3
General: Christian Hoff requested the possibility to provide extra formatting information to a column, e.g., colouring. That was implemented by having an extra parameter to the generic F-column. The current solution is not very robust, as font and/or size change in math environment are very tricky, but providing colour information works, see the example in the main text. This version is now compatible with package longtable and backwards compatible to fcolumn v1.2: it only adds functionality. 1

v1.4
General: More robust version on the extra formatting information: alternative formatting after the comma (if any), instead of additional information. Corrected a few typos. 1

v1.4.1
General: Definition needed for longtable only if that package is loaded. General update, streamlining code. Excess digits silently ignored when all zero. 1

v1.4.2
General: Better documentation. Removed redundant endgroup/begingroup pairs and other small improvements. Unbalanced columns are reported by stating their difference. Implementation of option “red”. No changes in user experience. 1

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.