The tabularx package*

David Carlisle

1999/01/07

Abstract

A new environment, `tabularx`, is defined, which takes the same arguments as `tabular*`, but modifies the widths of certain columns, rather than the inter column space, to set a table with the requested total width. The columns that may stretch are marked with the new token `X` in the preamble argument.

This package requires the `array` package.

1 Introduction

This package implements a version of the `tabular` environment in which the widths of certain columns are calculated so that the table is is a specified width. Requests for such an environment seem to occur quite regularly in `comp.text.tex`.

`tabularx` `\begin{tabularx}{`⟨*width*⟩`}{`⟨*preamble*⟩`}`

The arguments of `tabularx` are essentially the same as those of the standard `tabular*` environment. However rather than adding space between the columns to achieve the desired width, it adjusts the widths of some of the columns. The columns which are affected by the `tabularx` environment should be denoted with the letter `X` in the preamble argument. The `X` column specification will be converted to `p{`⟨*some value*⟩`}` once the correct column width has been calculated.

2 Examples

The following table is set with `\begin{tabularx}{250pt}{|c|X|c|X|}` ....

Multicolumn entry!		THREE	FOUR
one	The width of this column depends on the width of the table.[1]	three	Column four will act in the same way as column two, with the same width.

If we change the first line to `\begin{tabularx}{300pt}{|c|X|c|X|}` we get:

*This file has version number v2.07, last revised 1999/01/07.

[1] You can now use `\footnote` inside `tabularx`!

Multicolumn entry!		THREE	FOUR
one	The width of this column depends on the width of the table.	three	Column four will act in the same way as column two, with the same width.

3 Differences between `tabularx` and `tabular*`

These two environments take the same arguments, to produce a table of a specified width. The main differences between them are:

- `tabularx` modifies the widths of the *columns*, whereas `tabular*` modifies the widths of the inter-column *spaces*.
- `tabular` and `tabular*` environments may be nested with no restriction, however if one `tabularx` environment occurs inside another, then the inner one *must* be enclosed by `{ }`.
- The body of the `tabularx` environment is in fact the argument to a command, and so certain constructions which are not allowed in command arguments (like `\verb`) may not be used.[2]
- `tabular*` uses a primitive capability of TeX to modify the inter column space of an alignment. `tabularx` has to set the table several times as it searches for the best column widths, and is therefore much slower. Also the fact that the body is expanded several times may break certain TeX constructs.

4 Customising the behaviour of `tabularx`

4.1 Terminal output

`\tracingtabularx` If this declaration is made, say in the document preamble, then all following `tabularx` environments will print information about column widths as they repeatedly re-set the tables to find the correct widths.

As an alternative to using the `\tracingtabularx` declaration, either of the options `infoshow` or `debugshow` may be given, either in the `\usepackage` command that loads `tabularx`, or as a global option in the `\documentclass` command.

4.2 The environment used to typeset the `X` columns

By default the `X` specification is turned into `p{`⟨*some value*⟩`}`. Such narrow columns often require a special format, this may be achieved using the `>` syntax of `array.sty`. So for example you may give a specification of `>{\small}X`. Another format which is useful in narrow columns is ragged right, however LaTeX's `\raggedright` macro redefines `\\` in a way which conflicts with its use in a tabular or array environments. For this reason this package introduces the command `\arraybackslash`, this may be used after a `\raggedright`, `\raggedleft` or `\centering` declaration. Thus a `tabularx` preamble may specify `>{\raggedright\arraybackslash}X`.

`\arraybackslash`

[2]Since Version 1.02, `\verb` and `\verb*` may be used, but they may treat spaces incorrectly, and the argument can not contain an unmatched `{` or `}`, or a `%` character.

\newcolumntype

These preamble specifications may of course be saved using the command, `\newcolumntype`, defined in `array.sty`. Thus we may say
`\newcolumntype{Y}{>{\small\raggedright\arraybackslash}X}`
and then use `Y` in the `tabularx` preamble argument.

\tabularxcolumn

The `X` columns are set using the `p` column which corresponds to `\parbox[t]`. You may want them set using, say, the `m` column, which corresponds to `\parbox[c]`. It is not possible to change the column type using the `>` syntax, so another system is provided. `\tabularxcolumn` should be defined to be a macro with one argument, which expands to the `tabular` preamble specification that you want to correspond to `X`. The argument will be replaced by the calculated width of a column.

The default is `\newcommand{\tabularxcolumn}[1]{p{#1}}`. So we may change this with a command such as:
`\renewcommand{\tabularxcolumn}[1]{>{\small}m{#1}}`

4.3 Column widths

Normally all `X` columns in a single table are set to the same width, however it is possible to make `tabularx` set them to different widths. A preamble argument of `{>{\hsize=.5\hsize}X>{\hsize=1.5\hsize}X}` specifies two columns, the second will be three times as wide as the first. However if you want to play games like this you should follow the following two rules.

- Make sure that the sum of the widths of all the `X` columns is unchanged. (In the above example, the new widths still add up to twice the default width, the same as two standard `X` columns.)
- Do not use `\multicolumn` entries which cross any `X` column.

As with most rules, these may be broken if you know what you are doing.

4.4 If the algorithm fails...

It may be that the widths of the 'normal' columns of the table already total more than the requested total width. tabularx refuses to set the `X` columns to a negative width, so in this case you get a warning "X Columns too narrow (table too wide)".

The `X` columns will in this case be set to a width of 1em and so the table itself will be wider than the requested total width given in the argument to the environment. This behaviour of the package can be customised slightly as noted in the documentation of the code section.

5 The Macros

```
⟨*package⟩
\DeclareOption{infoshow}{\AtEndOfPackage\tracingtabularx}
\DeclareOption{debugshow}{\AtEndOfPackage\tracingtabularx}
\ProcessOptions
```

This requires `array.sty`.

```
\RequirePackage{array}[1994/02/03]
```

First some registers etc. that we need.

```
\newdimen\TX@col@width
\newdimen\TX@old@table
\newdimen\TX@old@col
\newdimen\TX@target
\newdimen\TX@delta
\newcount\TX@cols
\newif\ifTX@
```

Now a trick to get the body of an environment into a token register, without doing any expansion. This does not do any real checking of nested environments, so if you should need to nest one `tabularx` inside another, the inner one must be surrounded by `{ }`.

`\tabularx` Prior to v1.06, this macro took two arguments, which were saved in separate registers before the table body was saved by `\TX@get@body`. Unfortunatly this disables the `[t]` optional argument. Now just save the width specification separately, then clear the token register `\toks@`. Finally call `\TX@get@body` to begin saving the body of the table. The ``{\ifnum0=`}\fi`` was added at v1.03, to allow `tabularx` to appear inside a `\halign`.[3]

This mechanism of grabbing an environment body does have the disadvantage (shared with the AMS alignment environments) that you can not make extension environments by code such as

```
\newenvironment{foo}{\begin{tabularx}{XX}}{\end{tabularx}}
```

as the code is looking for a literal string `\end{tabularx}` to stop scanning. Since version 2.02, one may avoid this problem by using `\tabularx` and `\endtabularx` directly in the definition:

```
\newenvironment{foo}{\tabularx{XX}}{\endtabularx}
```

The scanner now looks for the end of the current environment (`foo` in this example.) There are some restrictions on this usage, the principal one being that `\endtabularx` is the *first* token of the 'end code' of the environment.

```
\def\tabularx#1{%
```

Allow `\tabularx` `\endtabularx` (but not `\begin{tabularx}` `\end{tabularx}`) to be used in `\newenvironment` definitions.

```
\edef\TX@{\@currenvir}%
  {\ifnum0=`}\fi
```

`\relax` added at v1.05 so that non-expandable length tokens, like `\textwidth` do not generate an extra space, and an overfull box. `\relax` removed again at v4.09 in favour of `\setlength` so if you use the calc package you can use a width of `(\textwidth-12pt)/2`.

```
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}
```

`\endtabularx` This does not do very much...

```
\let\endtabularx\relax
```

[3]This adds an extra level of grouping, which is not really needed. Instead, I could use ``\iffalse{\fi\ifnum0=`}\fi`` here, and ``\ifnum0=`{}\fi`` below, however the code here would then have to be moved after the first line, because of the footnote to page 386 of the TeXBook, and I do not think I should be writing code that is so obscure as to be documented in a footnote in an appendix called "Dirty Tricks"!

`\TX@get@body` Place all tokens as far as the first `\end` into a token register. Then call `\TX@find@end` to see if we are at `\end{tabularx}`.

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

`\TX@find@end` If we are at `\end{tabularx}`, call `\TX@endtabularx`, otherwise add `\end{...}` to the register, and call `\TX@get@body` again.

```
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
```

`\TX@` The string `tabularx` as a macro for testing with `\ifx`.

```
\def\TX@{tabularx}
```

Now that all the parts of the table specification are stored in registers, we can begin the work of setting the table.

The algorithm for finding the correct column widths is as follows. Firstly set the table with each `X` column the width of the final table. Assuming that there is at least one `X` column, this will produce a table that is too wide. Divide the excess width by the number of `X` columns, and reduce the column width by this amount. Reset the table. If the table is not now the correct width, a `\multicolumn` entry must be 'hiding' one of the `X` columns, and so there is one less `X` column affecting the width of the table. So we reduce by 1 the number of X columns and repeat the process.

`\TX@endtabularx` Although I have tried to make `tabularx` look like an environment, it is in fact a command, all the work is done by this macro.

```
\def\TX@endtabularx{%
```

Define the `X` column, with an internal version of the `\newcolumntype` command. The `\expandafter` commands enable `\NC@newcol` to get the *expansion* of `\tabularxcolumn{\TX@col@width}` as its argument. This will be the definition of an `X` column, as discussed in section 4.

```
  \expandafter\TX@newcol\expandafter{\tabularxcolumn{\TX@col@width}}%
```

Initialise the column width, and the number of `X` columns. The number of `X` columns is set to one, which means that the initial count will be one too high, but this value is decremented before it is used in the main loop.

Since v1.02, switch the definition of `\verb`.

```
  \let\verb\TX@verb
```

Since v1.05, save the values of all LaTeX counters, the list `\cl@@ckpt` contains the names of all the LaTeX counters that have been defined so far. We expand `\setcounter` at this point, as it results in fewer tokens being stored in `\TX@ckpt`, but the actual resetting of the counters occurs when `\TX@ckpt` is expanded after each trial run. Actually since v1.07, use something equivalent to the expansion of the original definition of `\setcounter`, so that `tabularx` works in conjunction with `calc.sty`.

```
  \def\@elt##1{\global\value{##1}\the\value{##1}\relax}%
  \edef\TX@ckpt{\cl@@ckpt}%
```

```
\let\@elt\relax
\TX@old@table\maxdimen
\TX@col@width\TX@target
\global\TX@cols\@ne
```

Typeout some headings (unless this is disabled).

```
\TX@typeout@
  {\@spaces Table Width\@spaces Column Width\@spaces X Columns}%
```

First attempt. Modify the `X` definition to count `X` columns.

```
\TX@trial{\def\NC@rewrite@X{%
          \global\advance\TX@cols\@ne\NC@find p{\TX@col@width}}}%
```

Repeatedly decrease column width until table is the correct width, or stops shrinking, or the columns become two narrow. If there are no multicolumn entries, this will only take one attempt.

```
\loop
  \TX@arith
  \ifTX@
  \TX@trial{}%
\repeat
```

One last time, with warnings back on (see appendix D) use `tabular*` to put it in a box of the right size, in case the algorithm failed to find the correct size.

Since v1.04, locally make `\footnotetext` save its argument in a token register. Since v1.06, `\toks@` contains the preamble specification, and possible optional argument, as well as the table body.

```
{\let\@footnotetext\TX@ftntext\let\@xfootnotenext\TX@xftntext
  \csname tabular*\expandafter\endcsname\expandafter\TX@target
    \the\toks@
  \csname endtabular*\endcsname}%
```

Now the alignment is finished, and the `}` has restored the original meaning of `\@footnotetext` expand the register `\TX@ftn` which will execute a series of
`\footnotetext[`⟨*num*⟩`]{`⟨*note*⟩`}`
commands. We need to be careful about clearing the register as we may be inside a nested `tabularx`.

```
\global\TX@ftn\expandafter{\expandafter}\the\TX@ftn
```

Now finish off the `tabularx` environment. Note that we need `\end{tabularx}` here as the `\end{tabularx}` in the user's file is never expanded. Now use `\TX@` rather than `tabularx`.

We also need to finish off the group started by ``{\ifnum0=`}\fi`` in the macro `\tabularx`.

```
\ifnum0=`{\fi}%
\expandafter\end\expandafter{\TX@}}
```

`\TX@arith` Calculate the column width for the next try, setting the flag `\ifTX@` to false if the loop should be aborted.

```
\def\TX@arith{%
  \TX@false
  \ifdim\TX@old@table=\wd\@tempboxa
```

If we have reduced the column width, but the table width has not changed, we stop the loop, and output the table (which will cause an over-full alignment) with the previous value of `\TX@col@width`.

```
    \TX@col@width\TX@old@col
    \TX@typeout@{Reached minimum width, backing up.}%
  \else
```

Otherwise calculate the amount by which the current table is too wide.

```
    \dimen@\wd\@tempboxa
    \advance\dimen@ -\TX@target
    \ifdim\dimen@<\TX@delta
```

If this amount is less than `\TX@delta`, stop. (`\TX@delta` should be non-zero otherwise we may miss the target due to rounding error.)

```
      \TX@typeout@{Reached target.}%
    \else
```

Reduce the number of effective `X` columns by one. (Checking that we do not get 0, as this would produce an error later.) Then divide excess width by the number of effective columns, and calculate the new column width. Temporarily store this value (times -1) in `\dimen@`.

```
      \ifnum\TX@cols>\@ne
        \advance\TX@cols\m@ne
      \fi
      \divide\dimen@\TX@cols
      \advance\dimen@ -\TX@col@width
      \ifdim \dimen@ >\z@
```

If the new width would be too narrow, abort the loop. At the moment too narrow, means less than 0 pt!

Prior to v2.03, if the loop was aborted here, the X columns were left with the width of the previous run, but this may make the table far too wide as initial guesses are always too big. Now force to `\TX@error@width` which defaults to be 1em. If you want to get the old behaviour stick
`\renewcommand\TX@error@width{\TX@col@width}`
in a package file loaded after tabularx.

```
        \PackageWarning{tabularx}%
           {X Columns too narrow (table too wide)\MessageBreak}%
        \TX@col@width\TX@error@width\relax
      \else
```

Otherwise save the old settings, and set the new column width. Set the flag to true so that the table will be set, and the loop will be executed again.

```
        \TX@old@col\TX@col@width
        \TX@old@table\wd\@tempboxa
        \TX@col@width-\dimen@
        \TX@true
      \fi
    \fi
  \fi}
```

`\TX@error@width` If the calculated width is negative, use this instead.

```
\def\TX@error@width{1em}
```

`\TX@delta` Accept a table that is within `\hfuzz` of the correct width.

```
\TX@delta\hfuzz
```

Initialise the X column. The definition can be empty here, as it is set for each tabularx environment.

```
\newcolumntype{X}{}
```

\tabularxcolumn The default definition of X is p{#1}.

```
\def\tabularxcolumn#1{p{#1}}
```

\TX@newcol A little macro just used to cut down the number of \expandafter commands needed.

```
\def\TX@newcol{\newcol@{X}[0]}
```

\TX@trial Make a test run.

```
\def\TX@trial#1{%
  \setbox\@tempboxa\hbox{%
```

Any extra commands. This is used on the first run to count the number of X columns.

```
    #1\relax
```

Since v1.04, make \footnotetext gobble its arguments. Also locally clear \TX@vwarn so that the warning is generated by the final run, and does not appear in the middle of the table if \tracingtabularx.

```
  \let\@footnotetext\TX@trial@ftn
  \let\TX@vwarn\@empty
```

Do not nest tabularx environments during trial runs. This would waste time, and the global setting of \TX@cols would break the algorithm.

```
   \expandafter\let\expandafter\tabularx\csname tabular*\endcsname
   \expandafter\let\expandafter\endtabularx\csname endtabular*\endcsname
```

Added at v1.05: dissable \writes during a trial run. This trick is from the TEXBook.[4]

```
   \def\write{\begingroup
     \def\let{\afterassignment\endgroup\toks@}%
        \afterassignment\let\count@}%
```

Turn off warnings (see appendix D). Also prevent them being turned back on by setting the parameter names to be registers.

```
    \hbadness\@M
    \hfuzz\maxdimen
    \let\hbadness\@tempcnta
    \let\hfuzz\@tempdima
```

Make the table, and finish the hbox. Since v1.06, \toks@ contains the preamble specification, and possible optional argument, as well as the table body.

```
    \expandafter\tabular\the\toks@
    \endtabular}%
```

Since v1.05 reset all LaTeX counters, by executing \TX@ckpt.

```
  \TX@ckpt
```

[4]Actually the TEXBook trick does not work correctly, so changed for v2.05.

Print some statistics. Added \TX@align in v1.05, to line up the columns.

```
\TX@typeout@{\@spaces
   \expandafter\TX@align
      \the\wd\@tempboxa\space\space\space\space\space\@@
   \expandafter\TX@align
      \the\TX@col@width\space\space\space\space\space\@@
   \@spaces\the\TX@cols}}
```

\TX@align Macro added at v1.05, to improve the printing of the tracing info.

```
\def\TX@align#1.#2#3#4#5#6#7#8#9\@@{%
  \ifnum#1<10 \space\fi
  \ifnum#1<100 \space\fi
  \ifnum#1<\@m\space\fi
  \ifnum#1<\@M\space\fi
  #1.#2#3#4#5#6#7#8\space\space}
```

\arraybackslash \\ hack.

```
\def\arraybackslash{\let\\\@arraycr}
```

\tracingtabularx Print statistics on column and table widths.

```
\def\tracingtabularx{%
  \def\TX@typeout{\PackageWarningNoLine{tabularx}}%
  \def\TX@typeout@##1{\typeout{(tabularx) ##1}}}
```

\TX@typeout The default is to be to be quiet

```
\let\TX@typeout\@gobble
\let\TX@typeout@\@gobble
```

\TX@ftn A token register for saving footnote texts.

```
\newtoks\TX@ftn
```

\TX@ftntext
\TX@xftntext Inside the alignment just save up the footnote text in a token register.

```
\long\def\TX@ftntext#1{%
  \edef\@tempa{\the\TX@ftn\noexpand\footnotetext
                    [\the\csname c@\@mpfn\endcsname]}%
  \global\TX@ftn\expandafter{\@tempa{#1}}}%
\long\def\TX@xftntext[#1]#2{%
  \global\TX@ftn\expandafter{\the\TX@ftn\footnotetext[#1]{#2}}}
```

\TX@trial@ftn On trial runs, gobble footnote texts.

```
\long\def\TX@trial@ftn#1{}
```

This last section was added at Version 1.02. Previous versions documentented the fact that \verb did not work inside tabularx, but that did not stop people using it! This usually put LaTeX into an irrecoverable error position, with error messages that did not mention the cause of the error. The 'poor man's \verb' (and \verb*) defined here is based on page 382 of the TeXBook. As explained there, doing verbatim this way means that spaces are not treated correctly, and so \verb* may well be useless, however I consider this section of code to be error-recovery, rather than a real implementation of verbatim.

The mechanism is quite general, and any macro which wants to allow a form of \verb to be used within its argument may \let\verb=\TX@verb. (Making sure to restore the real definition later!)

\verb and \verb* are subject to the following restictions:

1. Spaces in the argument are not read verbatim, but may be skipped according to TeX's usual rules.
2. Spaces will be added to the output after control words, even if they were not present in the input.
3. Unless the argument is a single space, any trailing space, whether in the original argument, or added as in (2), will be omitted.
4. The argument must not end with `\`, so `\verb|\|` is not allowed, however, because of (3), `\verb|\ |` produces `\`.
5. The argument must be balanced with respect to `{` and `}`. So `\verb|{|` is not allowed.
6. A comment character like `%` will not appear verbatim. It will act as usual, commenting out the rest of the input line!
7. The combinations `` ?` `` and `` !` `` will appear as ¿ and ¡ if the `cmtt` font is being used.

`\TX@verb` The internal definition of `\verb`. Spaces will be replaced by `~`, so for the star-form, `\let ~` be ␣, which we obtain as `\uppercase{*}`. Use ``{\ifnum0=`}\fi`` rather than `\bgroup` to allow `&` to appear in the argument.

```
{\uccode`\*=`\ %
\uppercase{\gdef\TX@verb{%
  \leavevmode\null\TX@vwarn
  {\ifnum0=`}\fi\ttfamily\let\\\ignorespaces
  \@ifstar{\let~*\TX@vb}{\TX@vb}}}}
```

`\TX@vb` Get the 'almost verbatim' text using `\meaning`. The '`!`' is added to the front of the user supplied text, to ensure that the whole argument does not consist of a single `{ }` group. TeX would strip the outer braces from such a group. The '`!`' will be removed later.

Originally I followed Knuth, and had `\def\@tempa{##1}`, however this did not allow `#` to appear in the argument. So in v1.04, I changed this to to use a token register, and `\edef`. This allows `#` appear, but makes each one appear twice!, so later we loop through, replacing `##` by `#`.

```
\def\TX@vb#1{\def\@tempa##1#1{\toks@{##1}\edef\@tempa{\the\toks@}%
    \expandafter\TX@v\meaning\@tempa\\ \\\ifnum0=`{\fi}}\@tempa!}
```

`\TX@v` Strip the initial segment of the `\meaning`, including the '`!`' added earlier.

```
\def\TX@v#1!{\afterassignment\TX@vfirst\let\@tempa= }
```

As explained above we are going to replace `##` pairs by `#`. To do this we need non-special `#` tokens. Make `*` into a parameter token so that we can define macros with arguments. The normal meanings will be restored by the `\endgroup` later.

```
\begingroup
\catcode`\*=\catcode`\#
\catcode`\#=12
```

`\TX@vfirst` As a special case, prevent the first character from being dropped. This makes `\verb*| |` produce ␣. Then call `\TX@v@`. This is slightly tricky since v1.04, as I have to ensure that an actual `#` rather than a command `\let` to `#` is passed on if the first character is `#`.

```
\gdef\TX@vfirst{%
  \if\@tempa#%
    \def\@tempb{\TX@v@#}%
  \else
    \let\@tempb\TX@v@
    \if\@tempa\space~\else\@tempa\fi
  \fi
  \@tempb}
```

`\TX@v@` Loop through the `\meaning`, replacing all spaces by `~`. If the last charcter is a space it is dropped, so that `\verb*|\LaTeX |` produces `\LaTeX` not `\LaTeX␣`. The rewritten tokens are then further processed to replace `##` pairs.

```
\gdef\TX@v@*1 *2{%
  \TX@v@hash*1##\relax\if*2\\\else~\expandafter\TX@v@\fi*2}
```

`\TX@v@hash` The inner loop, replacing `##` by `#`.

```
\gdef\TX@v@hash*1##*2{*1\ifx*2\relax\else#\expandafter\TX@v@hash\fi*2}
```

As promised, we now restore the normal meanings of `#` and `*`.

```
\endgroup
```

`\TX@vwarn` Warn the user the first time this `\verb` is used.

```
\def\TX@vwarn{%
  \@warning{\noexpand\verb may be unreliable inside tabularx}%
  \global\let\TX@vwarn\@empty}
```

```
⟨/package⟩
```