Hints and Tricks

'Hey — it works!'

Jeremy Gibbons

Welcome to *'Hey — it works!'*, a column devoted to (LA)TEX tips, tricks and techniques. In this issue, we have an article by Robert Tolksdorf, on automatically inserting or avoiding spaces after macros that expand to text (such as the macro `\TUB`, which generates '*TUGboat*'); this is based on a macro by Donald Arseneau in an earlier column in *TTN*. We also have an article by Pedro Aphalo on generating dashed lines of various kinds in LaTeX.

I have decided to expand the scope of the column to include also METAFONT and METAPOST techniques, prompted by a recent question on the METAFONT mailing list. To get the ball rolling, this issue concludes with an article of mine on drawing double-headed arrows in METAPOST. Please send me any more little METAFONT or METAPOST snippets you might have, along with the usual TeX and LaTeX ones.

◇ Jeremy Gibbons
CMS, Oxford Brookes University
Gipsy Lane, Headington
Oxford OX3 0BP, UK
jgibbons@brookes.ac.uk
http://www.brookes.ac.uk/~p0071749/

1 Smart spaced macros everywhere

The article *Italic correction everywhere* by Donald Arseneau in *TTN* 3,1:15 addresses the issue of inserting an italic correction automatically, if there is no punctuation following the italicized text.

A similar problem is the generation of spaces after a macro that generates text, such as the `\TUB` macro from the *TUGboat* document class. Consider the sentence "*TUGboat* uses the macro `\TUB` to generate '*TUGboat*'." The source for this sentence reads:

```
\TUB\ uses the macro \verb"\TUB"
to generate `\TUB'.
```

What one would like to avoid is the manually inserted `\␣` after the first `\TUB`. The following lines introduce the macro `\smartspace` that automagically inserts it when no punctuation follows the macro:

```
% smart insertion of space
% Robert Tolksdorf (tolk@cs.tu-berlin.de)
% Following Donald Arseneau,
% Italic correction everywhere, TTN 3,1
\def\smartspace#1{{\protect
  \aftergroup\smartspaceit#1}}
\def\smartspaceit{\futurelet\spta\sptest}
\def\sptest{\ifcat\noexpand\spta,\else\ \fi}
```

Now, we can define a macro `\TUGboat` by

```
\def\TUGboat{\smartspace{\TUB}}
```

and use

```
\TUGboat uses the macro \verb"\TUGboat"
to generate `\TUGboat'.
```

And hey, to quote Donald Arseneau, this works for 99 44/100% of the time only, as '`\TUGboat --`' shows. Someone tell me why![1]

◇ Robert Tolksdorf
Technische Universität Berlin
tolk@cs.tu-berlin.de

2 Dashed lines

Sometimes, for example when including data plots, it is necessary to include in the caption to a figure different dashed or entire line segments used to identify different lines, like this:

A (··············), B (— — —)

The size and location of these line segments should match the surrounding text. After reading Norbert Schwarz's 'Introduction to TeX' book, I wrote a very small package which I have been using for some time with LaTeX. It is based on a command which can generate most commonly used dashed lines.

```
\def\dashedrule#1#2#3{{%
  % #1 is length of dash
  % #2 is length of gap between dashes
  % #3 is number of dashes

  \dimen1=#2 \divide\dimen1 by 2

  \def\@ruledash{%
    \rule{\dimen1}{0pt}%
    \rule[0.5ex]{#1}{0.4pt}%
    % line is 0.5ex above the baseline
    % and 0.4pt thick
    \rule{\dimen1}{0pt}}%

  \count1=0
  \loop%
  \ifnum\count1<#3%
    \advance\count1 by 1%
    \@ruledash%
  \repeat}}
```

How does it work? `\@ruledash` draws a single dash plus half a gap in front of it, and half a gap after it. A loop draws as many dashes, surrounded

[1] Note that if you make the macro `\TeX` smart-spaced, then '`\TeX book`' no longer works as it used to!

by half gaps, as indicated by the third argument. Using this command it is extremely easy to define different dashed line segments of equal length, as the example below shows for 3em-long line segments.

```
% length of line segment is (#1 + #2) * #3
\def\longdashes{\dashedrule{.8em}{.2em}{3}}
\def\mediumdashes{\dashedrule{.3em}{.2em}{6}}
\def\shortdashes{\dashedrule{.1em}{.1em}{15}}
\def\solidline{\dashedrule{3em}{0em}{1}}
\def\sparsedashes{\dashedrule{.5em}{.5em}{3}}
```

'— — —'

'- - - - - -'

'...............'

'———'

'_ _ _'

◇ Pedro J. Aphalo
Faculty of Forestry
University of Joensuu
pedro.aphalo@joensuu.fi
http://cc.joensuu.fi/~aphalo/

3 Double-headed arrows

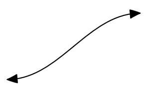
A recent request on the METAFONT mailing list was for help in drawing double-headed arrows. One correspondent provided the following definition of a macro to draw a path with an arrowhead at each end:

```
def draw_dbl_arrow text t =
  path p, q;
  p := t;
  q := subpath (0,.5) of p;
  drawarrow reverse q;
  q := subpath (.5,1) of p;
  drawarrow q
enddef;
```

For example,

```
draw_dbl_arrow (0,0){right} .. {right}(50,25);
```

produces:



This definition can be improved in several ways. For one thing, there is no need to use assignments like this. METAPOST[2] has a powerful expression language, and in particular you can use an expression as the argument to `drawarrow`:

```
def draw_dbl_arrow text t =
  drawarrow reverse (subpath (0,.5) of p);
  drawarrow subpath (.5,1) of p
enddef;
```

[2] Although we use the term 'METAPOST' to refer to the language, everything in this article applies equally to the METAPOST and METAFONT systems.

For another thing, that '`1`' should be '`length p`', otherwise the macro will only work for a path of length 1.

Indeed, there is no need to draw just subpaths of `p`; there is no harm in drawing `p` itself twice:

```
def draw_dbl_arrow text t =
  drawarrow reverse p;
  drawarrow p
enddef;
```

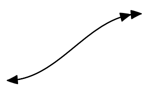
In fact, the original poster asked for a triple-headed arrow, with two arrow heads at (for the sake of argument) the end of the path. If you can pick the right small value of `e`, you can achieve this by just drawing the path three times:

```
def draw_trp_arrow text t =
  drawarrow reverse p;
  drawarrow p;
  drawarrow subpath (0, length p - e) of p
enddef;
```

But what value to pick for `e`? You could just experiment, but different values will be needed for different paths to get consistent results. A better approach is to find the time at which a point traversing path `p` is a certain fixed distance (namely, the length of an arrow head, `ahlength`) from the end of `p`, and to draw the corresponding subpath of `p`. You can find that time using `intersectiontimes`, intersecting `p` with a circle of the appropriate size centred on the end of `p`.

```
def draw_trp_arrow text t =
  drawarrow reverse p;
  drawarrow p;
  path q;
  q := fullcircle scaled (2*ahlength)
       shifted (point (length p) of p);
  numeric tp, tq;
  (tp,tq) = p intersectiontimes q;
  drawarrow subpath (0, tp) of p
enddef;
```

On the same path as before, `draw_trp_arrow` gives



It is assumed that `p` is suitably well-behaved, that is, that it crosses the small circle just once.

(Again, it is possible to do it without those assignments, but then the argument to the third `drawarrow` gets rather unwieldy.)

◇ Jeremy Gibbons
Oxford Brookes University
jgibbons@brookes.ac.uk