Problems on the TEX/PostScript/Graphics Interface

Robert A. Adams
Dept. of Mathematics, The University of British Columbia, Vancouver B.C. Canada V6T 1Y4
Bitnet: useradms@ubcmtsg, Internet: useradms@mtsg.ubc.ca

Abstract

This paper discusses several problems which arose in the process of using TEX and PostScript together to produce two calculus textbooks. Three of these problems were particularly important. The first was getting a reasonable combination of PostScript (scalable) text and math fonts that looked "good" in 1270 dpi output from a Linotronic phototypesetter. The second was devising a practical method for getting suitable (and well aligned) two-colour separation for text and graphics. The third involved incorporating TEX labelling in PostScript graphics. Solutions to these problems were largely dictated by the software available at the time the solutions were needed, about one and a half years ago.

Background

TEX was designed to produce beautiful books, especially ones which contain mathematical formulas. It is therefore natural to choose TEX to typeset a calculus book, but calculus books require numerous diagrams which themselves have mathematical formulas for labels. It is fairly easy to produce even very complex mathematical diagrams in PostScript, either directly or indirectly using high-level software which generates PostScript code. Therefore it is also natural to produce a calculus book in a PostScript environment.

During the past two years I have been involved in many aspects of the production of two calculus textbooks, *Single-Variable Calculus*, and *Calculus: A Complete Course*, both published in Canada by Addison-Wesley. Besides writing these books, I was responsible for all the typesetting, and the construction of all the macros necessary to implement a book design. Many (but not all) of the design elements were specified by a professional book designer.

Anyone who has ever authored a textbook using any system will know what a monumental job that can be. Knowing what information you want to present, and how you want to present it, is only a small part of the task. Getting a respectable typescript copy in the days before personal computers, word processers and computer graphics packages usually meant more hours at a typewriter or drawing board, or preparing and editing handwritten copy for a typist and artist, than the author actually spent composing the material. Such was the state of affairs when I wrote the first edition of *Single-Variable Calculus* for Addison-Wesley in 1981 - 1982. It was my second book done by the *old method*, and I resolved at the time never to write another book! Then in 1984 my Editor sent me Addison-Wesley's newly published MicroTEX, and a copy of *The TEXbook*, and my life was changed forever. He wanted a review of MicroTEX. He got a review, and another book, *Calculus of Several Variables* (Addison-Wesley, 1987).

At that time we were still using Almost Modern fonts, and this author, at least, had never even heard of PostScript. The typesetting was successful enough, though Addison-Wesley (Canada) and I were both feeling our way as far as design was concerned. The problem of two colour separation came up, but was not adequately solved. In the end the production department got out the scissors and glue, and the separation was done *a posteriori* without the help of a computer. The diagrams were all redone by a graphics professional on a Macintosh from plotter copy I supplied, and they left a lot to be desired. Moreover, there were several serious colour alignment errors in the final book, which arose from the fact that the alignment of black and second-colour components for the figures were performed by someone who did not fully understand the devastating consequences of even a minor misalignment in a complicated two-colour figure. I decided at that time that if I ever did another book, I would try to have a system in which I could produce most of the figures and integrate them directly into the production files myself.

By the time Addison-Wesley had raised the issue of a new edition of *Single-Variable Calculus*, (mid 1988), I had become reasonably familiar with PostScript and had acquired a 300dpi PostScript printer for my PC system. I had also developed a preliminary but useful version of a two- and three-dimensional mathematical graphics program MG which produced the kinds of figures I use in my books. (I had used that software to generate rough plotter copy for the several variable book.) There were, however, some problems which still had to be solved. Of these, the most important were

- getting suitable fonts to use with TEX for doing mathematical typesetting in a PostScript environment.
- devising a simple system for getting two-colour separation in text and graphics.
- geting TEX labels into my figures.

I will deal with each of these in turn.

The Font Problem

At the outset I should say that my TEX setup two years ago consisted of an AT clone with colour EGA monitor, Addison-Wesley's MicroTEX, and (version 4.0 of) ArborText's PREVIEW and DVILASER/PS driver programs. I had constructed a modified version of `plain.tex` called `psplain.tex` which used a hybrid of PostScript Times fonts for text mode material and Computer Modern math fonts for math mode material, with a few other minor modifications to clear up some problems which arise from the fact that TEX manufactures some symbols such as "$\neq$" using elements from text and math families.

Copy at 300 dpi resolution obtained from `psplain` looked fine to me, and to my editors. However, a sample generated on a 1270 dpi PostScript phototypesetter exposed the first serious problem. While the text mode material came out at 1270 dots per inch, the math mode material was still at 300 dots per inch, because DVILASER/PS had downloaded raster patterns for the `cmmi`, `cmex`, and `cmsy` fonts into the PostScript file. I suppose we could have tried to obtain 1270 dpi versions of those fonts, but I had no access to METAFONT.

About that time, I was given a copy of some PostScript (scalable) versions of these fonts produced on a Macintosh using the FONTOGRAPHER program, so we tried them. The combination again looked good at 300 dpi, but at 1270 dpi a new problem became apparent. The CM math fonts have considerably less weight than the Times family of text fonts. Here is a sample formula involving characters from both families. It is magnified (to 20 points) to show the difference in weights.

$$\max\{a^j, x_k\} > \cos\psi$$

The combination would not do at all. At this point I would have given a good deal for a working version of John Hobby's *MetaPost* program [Hobby, 1989], or any other program that would produce PostScript outline fonts from METAFONT descriptions. I had a preprint copy Leslie Carr's paper [Carr, 1988] on converting METAFONT logfile output into a PostScript font description, but I was certain I was not a good enough programmer to implement it, at least not quickly.

A solution for this problem was finally found, and it was definitely a hack. The FONTOGRAPHER program generates the characters of a PostScript outline font in a coded format which is preceeded in its output file by a PostScript prolog with definitions which enable the PostScript interpreter to understand the code and construct the character in the printer's memory. Being machine produced PostScript, even these definitions are a bit hard to read, but after some study I was able to conclude that the character outlines were merely being *filled* (with black) rather than *stroked* with a PostScript pen. Lines 10 and 11 and 22 in this prolog began

```
/Fill{{fill}Cfill}def
/Eofill{{eofill}Cfill}def

/StrokeWidth 0 def
```

I altered the definition of the `Fill` and `Eofill` operators being used so that in addition it `stroked` the outline with a pen of a prescribed thickness. After some experimentation, I determined that the thickness should be about 0.22 points for a nominal 10 point font. Thus, the PostScript prolog for the outline fonts `cmmi`, `cmsy`, and `cmex` was modified to become

```
/Fill{{gsave fill grestore stroke}
      Cfill}def
/Eofill{{gsave eofill grestore
      stroke}Cfill}def

/StrokeWidth 22 def
```

(The StrokeWidth variable is measured in thousandths of the nominal design size of the font.) Off went another test to the phototypesetter, this time successfully. Here is a sample of the output with the same mathematical formula shown earlier.

$$\max\{a^j, x_k\} > \cos\psi$$

I'm sure that experts in font design would find any combination of two such different typefaces as Times and Computer Modern aesthetically unsatisfactory, but the average mathematics student or instructor, and maybe even the average editor, does not. As so often happens in the real world, there was a problem which needed an immediate solution. While not ideal, an acceptable solution was found.

The Colour Separation Problem

Most textbooks these days use two or even four or more colours to achieve greater visual impact. Several queries about how to accomplish this with TEX have appeared in TEXhax in the last few years (one was from me), and I have never seen an adequate response. Of course, in a sense SLITEX has solved the multi-colour problem by using blank fonts (which have TFM files corresponding to those of printing fonts but themselves print only blank characters. It has, however, never been clear to me how to obtain (or construct) such blank fonts. There is also the TEX \phantom command, but I'm not sure how that would react to a pagefull of text to be blanked out. (I admit, I've never tried.)

Ideally, one would like to arrange the following situation for two-colour separation. There should be defined two control words, \black and \red say, so that you would insert one of these words in the TEX source code at points where you wanted to switch from red to black or from black to red. There should also be defined at the beginning of the TEX source two Boolean controls, \printblack and \printred, which should be set to *true* or *false* according to whether "black" output, "red" output, or both combined is desired. There remains, however, the problem of how to get the nonprinting colour to leave blank areas on the page exactly corresponding to the material that would appear if it were printing.

I still do not know how to solve this problem using TEX, but there are fairly easy PostScript solutions. PostScript has a setgray operator which determines the gray-level of printing. Thus 0 setgray causes printing in black; 1 setgray causes printing in white, i.e. no printing at all unless the background is not white. Numbers between 0 and 1 result in different levels of gray. Define printblack and printred as PostScript Boolean variables which you set to true or false according to whether you want either or both colours to print. Then have the TEX \black and \red commands insert PostScript operators black and red respectively, into the PostScript file via \special commands to the PostScript driver. The PostScript operator black could be defined as 0 setgray if printblack is true, and 1 setgray if printblack is false. A similar defintion is made for red.

The above solution works well for text (e.g. headings, boxes and such items where black and red are never overlaid), and it is clearly generalizable to more colours. However, it poses problems for graphic material. PostScript is designed so that graphic elements plotted later always obscure ones plotted earlier in regions of overlap. For example, in a figure where a red curve (or pink shaded region) crosses or overlaps an earlier plotted black curve (or gray shaded region), the red element will blank out those parts of the black element where it overlaps. This is not what you want! In the final copy black ink is quite opaque, red less so, and light shades of pink or gray are not at all opaque and should not blank out one another.

The solution to this problem was to redefine the PostScript operators black and red so that, depending on the values of printblack and printred, each translates the PostScript origin some large distance in one direction or another. This causes printing of the undesired colour to occur well outside the boundaries of the physical page, and thereby leaves the printed elements intact. The PostScript driver DVILASER/PS can insert some PostScript prolog code of its own at the beginning of the PostScript output file it creates from a TEX dvi file. In my installation, that prolog code begins

```
% SET THE FOLLOWING BOOLEAN SWITCHES true
% FOR WHICHEVER "COLOUR" OF OUTPUT IS
% DESIRED.  SET BOTH TO true TO PRINT
% BOTH COLOURS SIMULTANEOUSLY.  DO NOT
% SET BOTH SWITCHES false AT THE SAME TIME
%
/printblack { true  } def
/printred   { true  } def
%
/rred { printred {0 setgray} {1 setgray }
        ifelse } def
/bblack { printblack {0 setgray}
        {1 setgray} ifelse } def
%
/black { bblack firstswitch
```

```
    {/doingblack {true} def
    printblack not {5000 5000 translate
    /firstswitch {false} def} if }
    { doingblack not {5000 5000 translate
    /doingblack {true} def } if }
    ifelse } def
  %
  /red { rred firstswitch
    {/doingblack {false} def
    printred not {-5000 -5000 translate
    /firstswitch {false} def} if }
    { doingblack {-5000 -5000 translate
    /doingblack {false} def } if }
    ifelse } def
  %
  /setoldgray { currentgray dup
    /oldgray exch def } def
  %
  /restoregray { oldgray setgray } def
  %
  /fixblack { setoldgray pop 0 setgray } def
  %
  /maxgray { dup /newgray exch def
    setoldgray ge {newgray} {oldgray}
    ifelse setgray } def
```

These definitions are a little complicated, probably more so than absolutely necessary to achieve the desired effect. The definitions of setoldgray, fixblack, and restoregray are made to facilitate the printing of crop marks, registration crosshairs, and manuscript header information outside the margins of what will be the final trimmed page, on all printed pages regardless of the settings of printblack and printred. The operator maxgray is useful when colour separating shaded figures.

These PostScript definitions are accessed in the TeX source code by means of the following control words defined at the beginning of the macro file containing all the macros for the book.

```
  %    SVC-PS.TEX
  %
  %    Format for Calculus Book
  %    (PostScript Version)
  %    R. Adams   revised 15 Dec 89
  %
  % first some defs to set up
  % Postscript for two colours
  \def\red{\special{ps:: rred }}
  \def\black{\special{ps:: bblack }}
  \def\fixblack{\special{ps::
    bblack fixblack }}
  \def\restoregray{\special{ps::
    restoregray }}
  \def\logo{\hbox to17pt{\special{ps::
    rred logo bblack}\hfil}}
  \def\regmark{\hbox to60pt{\special{ps::
    regmark}\hfil}}
```

Here logo and regmark are PostScript procedures which produce a logo character for use in section headings in the book, and the registration crosshairs mentioned above. To illustrate the use of \red and \black in the TeX source, here is the definition of the macro \examp used to introduce examples in the book.

```
  \long\outer\def\examp #1\par{\penalty-200
    \vskip 12pt plus 2pt minus2pt
    \global\advance\itemno by1
    \noindent\llap{{\exampfont\red EXAMPLE
    \itemlabel\hskip1pc\black}}#1}
```

The word "EXAMPLE" and its label number are printed in red in the left margin. The \makeheadline macro illustrates the use of \fixblack to ensure that the crop marks \ulc and \urc, and headline material outside the crop boundaries print regardless of which colour is printing. Exceptions are the words "black" which is printed only if black is printing, and "colour" which prints only if colour is printing.

```
  \def\makeheadline{\vbox to 0pt{%
    \vskip-82pt\hbox to\pagewidth{%
      \fixblack\kern-196pt\copy\ulc
      \qquad\raise12pt\hbox{%
        \figfont ADAMS:
        Single-Variable Calculus
        Chapter \number\chapno\ -- page
        \number\pageno\quad
        \red colour \black black
        \fixblack\quad\today} \quad
      \raise12pt\regmark
      \hfill\rlap{\kern42pt\copy\urc}
      \restoregray}
    \vskip26pt
    \hbox to\pagewidth{\the\headline}\vss}
    \nointerlineskip}
```

This system for colour separation works well. In PostScript code for figures, one inserts the PostScript operators red and black where colour changes are desired.

TeX Labels on PostScript Graphics

In a calculus textbook mathematical graphics, both two and three dimensional, are a very important tool for presenting information and making it intelligible to the student. Most such graphics require labels involving mathematical formulas, sometimes almost as complicated from a typesetting point of view as the formulas appearing in the text. It is therefore very helpful to be able to use TeX to label figures. Most commercial software programs which produce mathematical graphics do not support this capability yet. On the other hand, programs designed specifically for doing graphics within a TeX environment are not of sufficient sophistication to produce the quality of mathematical graphics which can be generated by writing PostScript code.

I wanted the best of both worlds, and it seemed necessary in this instance to do some actual programming to produce a mathematical graphics package which would produce both PostScript output for the graphic and, simultaneously, TEX labelling information. Over several previous years I had developed, using Turbo Pascal, a graphics program, MG, which produced a variety of two-dimensional plots of functions and equations as well as lines, vector fields, freehand spline curves and such, and was also able to produce three-dimensional diagrams of curves and surfaces ruled by families of curves. The program produced HPGL output, because I happened to have a Hewlett Packard plotter at the time. Such output was not of suitable quality for publication.

About two years ago my colleague, Dr. Robert Israel (Mathematics Department, The University of British Columbia) took over the MG project and redesigned the user interface, making the program much easier to use, and at the same time much more functional. Meanwhile, I altered the file output of the program to produce, instead of a single HPGL file, two files for each figure created, one a PostScript description of the graphic, and the other a text file containing labelling information in TEX-readable form. All that was then needed was a TEX `\figinsert` macro to pass on the PostScript file in a `\special`, and read the label file, typesetting the labels at the correct positions.

Specifically, the command

```
\figinsert{myfig}
```

carries out the following operations:

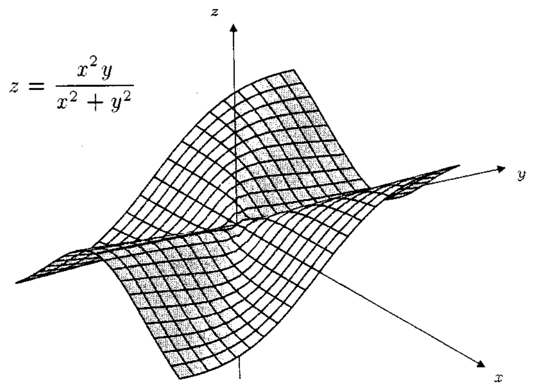
- First it opens the label file `myfig.lbl` and reads the first two lines, which contain integers giving the width and height of the graphic in points. It builds a vbox with those dimensions and vfills it so that the PostScript `currentpoint` (from the PostScript driver's point of view) is at the bottom left corner of that box.
- Next it passes the file `myfig.ps` to the driver with a `\special`. The PostScript code in `myfig.ps` (which is bracketed by a PostScript `gsave`—`grestore` pair) translates the PostScript origin to the current point and draws the figure.
- Finally the macro processes the remaining lines of the label file in groups of five. These are x and y coordinates of the position of the label in the vbox, two codes representing the horizontal and vertical justification or centering of the label, and finally the TEX label itself.
- Reading of the label file terminates when a negative x coordinate is read. (MG inserts -1 as the last line of the file.)

  Here is a list of the `\figinsert` macro.

```
\newcount\pswidth
\newcount\psheight
\newcount\justx \newcount\justy
\global\justx=0 \global\justy=0
\newcount\vpos \newtoks\label
\newread\labelfile
\newcount\xcoord \newcount\ycoord
\newif\ifdoit \newbox\labox
%
\def\newfiginsert#1{\openin\labelfile=#1.LBL
\global\read\labelfile to\pswidth
\global\read\labelfile to\psheight
\vbox to\psheight pt{\vfill
  \special{ps:: /firstswitch {true} def }
  \special{ps: plotfile #1.PS}
  \special{ps:: printblack not
     {5000 5000 translate} if }
  \vskip-\psheight pt\ninepoint%
  \hbox to\pswidth pt{\hss}%
  \parindent=0pt\offinterlineskip
  \vpos=0
      % read in label information
  \loop
    \global\read\labelfile to\xcoord
      % test for end of labelfile
    \ifnum \xcoord < 0 \doitfalse
      \else\doittrue\fi
    \ifdoit \global\read\labelfile to\ycoord
    \global\read\labelfile to\justx
    \global\read\labelfile to\justy
    \global\read\labelfile to\label
    \global\setbox\labox=\hbox{\label}
      % insert he label, suitably justified
    \advance\vpos by-\ycoord
    \vskip-\vpos pt \vpos=\ycoord%
    \hbox to\pswidth pt{\hskip\xcoord pt%
     \hbox to 0pt{\ifnum\justx>0\hss\fi%
      \vbox to0pt{%
       \ifnum\justy<2\vss\fi%
       \nointerlineskip\vbox
          to\dp\labox{\vfil}
       \nointerlineskip\copy\labox%
       \nointerlineskip\vbox
          to\ht\labox{\vfil}
       \nointerlineskip%
       \ifnum\justy>0\vss\fi}%
      \ifnum\justx<2\hss\fi}%
     \hss}
  \repeat
  \special{ps:: printblack not
    {-5000 -5000 translate} if }
  \advance\vpos by-\psheight%
  \vskip-\vpos pt}
\closein\labelfile}
```

The `\figinsert` macro is a low-level one. In practice, figures are inserted by more high-level macros which call `\figinsert` to place one or more figures across a line or in a box, and which may also supply titles or headings for the figures.

The system is not perfect. MG takes no account of the actual size of a label, vertically or horizontally. (It does not attempt, for instance, to read any TFM files.) Therefore, each figure needs to be printed once after it is created to check on the positions of labels. Occasionally a label needs to be moved in one direction or another to avoid colliding with other elements in the figure. Such moving of labels is most easily accomplished by directly editing the label file to alter the coordinates of the label. All one needs is a pica ruler such as the handy plastic ones given out at TEX Users Group meetings to advertise PCTEX. Here is an example of a figure created by MG for *Calculus: A Complete Course*,



and here is its label file.

```
217
173
204
149
0
0
$\ss x$
214
66
0
2
$\ss y$
108
7
1
2
$\ss z$
75
57
2
0
$z=\frac{x^2 y}{x^2+y^2}$
-1
```

Here \ss is an abbreviation for \scriptstyle.

One final comment about producing PostScript graphics for inclusion with TEX. There are numerous programs on the market which can be used to produce PostScript graphics output on an IBM PC, a Macintosh, or on other personal computers. (See J. T. Renfrow's paper [Renfrow, 1989] for methods of integrating such graphics into a TEX document.) More and more of these programs are capable of generating what is called Encapsulated PostScript. This means that the PostScript code which defines the graphic is bracketed by interfacing code in a standard format which appears as ignorable comments to the PostScript interpreter, but which conveys necessary information (for example the size of the graphic) to external programs which must assimilate the graphic as part of a larger document. It is encouraging to see implementers of DVI-PostScript drivers such as ArborText are now taking note of this standard and providing for easy inclusion of Encapsulated PostScript in TEX documents via `\special` commands.

Bibliography

Carr, Leslie. "Of METAFONT and PostScript." *TEXniques* 5, pages 141–152, 1988.

Hobby, John D. "A METAFONTlike System with PostScript Output." *TUGboat* 10(4), pages 505–512, 1989.

Renfrow, J. T. "Methodologies for Preparing and Integrating PostScript Graphics." *TUGboat* 10(4), pages 607–626, 1989.