Graphics

LaTeX, *dvips*, EPS and the web ...

Sebastian Rahtz

Abstract

Browsers of TeX question *fora* like `comp.text.tex` will often be asked what are the issues surrounding Encapsulated PostScript, and how one goes about making EPS files from LaTeX output, and maybe using them on the World Wide Web. This short note offers some suggestions.

1 What and why is EPS?

EPS stands for Encapsulated PostScript; EPS files *are* PostScript, but they conform to a minimum standard of good behaviour. This is so they can be included in other documents, possibly resized or rotated. In practice EPS means not using certain commands which have global effects (don't worry, this is quite rare), and inserting structured comments (starting with `%%`) which tell other programs something about the file. The *PostScript Language Reference Manual* goes into great depth describing what these comments can contain, but the minumum that is necessary for practical purposes are:

1. A first line starting `%!PS-Adobe`; *dvips*, for instance, puts `%!PS-Adobe-2.0 EPSF-2.0` in its output, meaning that it claims conformance with version 2 of the EPS standard (we are now at version 3);

2. A 'BoundingBox', like `%%BoundingBox: 33 101 584 715` which tells applications how much space on the page is occupied.

How do you turn PS files into EPS files? They probably are already, if they come from a reputable bit of software (avoid anything from Micro$oft) — a good check is to see if there is a BoundingBox.

You will come across three types of problem with files that look like EPS. Firstly, the BoundingBox may not be accurate; since this determines how much space will be left in enclosing applications like TeX, it matters. Keith Reckdahl's recent tutorial in *TUGboat* goes into detail on this problem.

Secondly, your file may be *serious* EPS, and use all the facilities of structured comments to specify what sort of resources (fonts etc) it expects you to supply when you deal with it. This is bad news if you are in TeX world outside a Macintosh. Look out for lines with words like `ProcSetsNeeded`.

Thirdly, your file may think it is EPS, but in fact breaks the rules, and has weird PostScript in it. The rescue technique is to read it with a forgiving PostScript interpreter, and get a new version written out. Three programs to try are:

1. Adobe Acrobat Distiller; this turns PostScript files into PDF, and Acrobat Exchange can then load them, and save them as ordinary PostScript. Since it is written by Adobe, Distiller is an extremely powerful PostScript interpreter, and can cope with almost anything you throw at it. It is not cheap, but worth having.

2. Recent versions of Adobe Illustrator share some of the Acrobat code, and can read PostScript files, as well as edit PDF files.

3. The free GhostScript is now a very mature and sophisticated product. It understands all of the current Level 2 PostScript, and can turn it onto a wide variety of bitmap forms. Version 4 (released in June 1996) also performs many of the functions of Distiller, and it already reads PDF files and writes PostScript. Unfortunately, its handling of PostScript text to PDF is at present unfinished. However, you can still use GhostScript to read your PostScript and write it out again as a bitmap (eg TIFF).

2 What about dvi to Encapsulated PostScript?

Most TEX systems, free or commercial, supply a dvi to PostScript driver; most of them write out more or less acceptable Encapsulated PostScript, but three are especially well-featured (in the author's experience): the Macintosh Textures driver, Y&Y's *dvipsone* for DOS and the free *dvips*. Since the latter is available for all platforms, is well-supported, and is probably the finest of its type,[1] we shall concentrate on that.

If you want to produce re-useable PostScript output from *dvips* (and this includes output destined for Acrobat Distiller), the absolute priority is to use outline fonts, not the PK fonts traditionally used by TEX. You can either use traditional fonts (usually commercial, like Adobe Times, but GhostScript now comes with an excellent free set donated by URW) or Computer Modern itself in PostScript Type 1 format. Either buy these from Y&Y for Windows and Unix or Blue Sky for Macintosh, or use Basil Malyshev's BaKoMa set, of almost comparable quality.[2]

If you do not use outline fonts, and re-use your output scaled up, you will not like the effect of Figure 1 at all, compared to Figure 2. If you want to turn your documents into PDF, Distiller will produce vile results from PK fonts.

The second priority is to get the right bounding box. Surprisingly many applications cheat by simply making it the page size, regardless of whether the whole area is used. *dvips* does this by default too, but has a command-line option `-E`, which asks it to try and calculate the actual extent used. Note that EPS files are, by definition, only one page, so you also have to use *dvips* options to select just one page. There are two caveats when preparing the input. Firstly, make sure you do not include a page number (try `\pagestyle{empty}` in LATEX), or else the bounding box will cover that too. Secondly, *dvips* does not always work out the extent of text correctly. For instance, if you wrote (why, I have no idea):

```
Hello\raisebox{10pt}[0pt][0pt]{Up there}!
```

you would be asking LATEX to raise *Up there* off the baseline, but to pretend that it has no effect on the height calculation. *dvips* will believe this, and calculate a bounding box on the *claimed* height. If you use complicated add-in packages like PSTricks, which add in arbitrary PostScript code, you will also end up in real trouble. In these cases you can either adjust the BoundingBox by hand, or place invisible marks in LATEX to make sure that *dvips* recognizes the full extent.

A useful trick to remember if you think that TEX knows what you want, but *dvips* does not, is to make judicious use of color. Suppose you wanted to use PSTricks to encircle a mathematical symbol, you might write:

```
absurd \pscirclebox{$\surd$}
```

TEX leaves the right space, since the PSTricks macros understand what is going on, but *dvips* is told to draw the circle in raw PostScript, and the bounding box calculation ignores that. The result is that the limits are set just around the size of the letters. If we wrote:

```
\framebox{absurd \pscirclebox{$\surd$}}
```

it would work correctly, because *dvips* would look at the enclosing frame, not just the words. But you end up with an unwanted box; so make it (in effect) invisible by writing:

```
{\color{white}\fboxsep{0pt}%
 \framebox{%
  {\color{black}absurd
  \pscirclebox{$\surd$}}%
  }%
}
```

This creates a white frame around black text; LATEX proceeds happily, and so does *dvips*, calculating the right extents, but nothing shows on paper. Obviously, this only works in a monochrome environment.

3 LATEX to EPS to GIF to Web

Why do we do all this in practice? Often, these days, because people want their LATEX mathematical output on the World Wide Web, and their only recourse is to embed GIF images in their HTML. The sophisticated *latex2html* program does all this for you; its technique is worth understanding, as it has general utility; the sequence of events is:

1. Place bits of LATEX in an special file, one fragment per page, and with no page numbers;
2. Run LATEX to generate a multi-page dvi file;
3. Use *dvips*' `-i` and `-S` options to generate one self-contained output file per page;
4. Give each page to GhostScript, and ask it to render them in pbm (Portable Bitmap) form;

[1] For several years, *dvipsone* has offered partial downloading of fonts, a very powerful feature, but this is now coming into *dvips*; there are also flaws in *dvips*' use of structured EPS comments, and Textures is superior in this respect.

[2] Windows-worshippers may prefer to get into the world of TrueType fonts, which are available for Computer Modern from Kinch Computer Company.

$$-\Phi_0\frac{\partial}{\partial\varphi}\left(\Phi_1 a\sin\varphi\right)-\Phi_1\frac{\partial}{\partial\varphi}\left(\Phi_0 a\sin\varphi\right)-A_1\left[\Phi_0+\frac{\partial}{\partial a}\left(a\Phi_0\right)\right]\sin\varphi=-a\Phi_0 f\sin\varphi \tag{1}$$

Figure 1: Bitmap EPS file, enlarged and distorted

$$-\Phi_0\frac{\partial}{\partial\varphi}\left(\Phi_1 a\sin\varphi\right)-\Phi_1\frac{\partial}{\partial\varphi}\left(\Phi_0 a\sin\varphi\right)-A_1\left[\Phi_0+\frac{\partial}{\partial a}\left(a\Phi_0\right)\right]\sin\varphi=-a\Phi_0 f\sin\varphi \tag{1}$$

Figure 2: Outline font EPS file, enlarged and distorted

5. Use the PBMplus/Netpbm utility *pnmcrop* to trim away white space;
6. Use the *ppmtogif* utility to convert the result to a GIF image.

Note that it does *not* use the `-E` option for *dvips*, but relies on simply removing all white pixels until just text is left. This has the advantage that it avoids the problem we saw in the last section, but it has three disadvantages:

1. The PBM utilities are primarily Unix tools, and many people do not have access to them;
2. The cropping process is memory-intensive, slow and eats temporary disk space;
3. The cropping forces everything to the baseline, effectively. A character like em-dash (—) which sits above the baseline, will be cropped above and below, so that the placed GIF looks wrong.

The core of the problem is the use of GhostScript, which always creates a page-sized bitmap, even if there is only one word on the page. What we want is for GhostScript to render just the portion of the image inside the bounding box, if we *do* use the `-E` flag for *dvips*. We can achieve this by giving GhostScript a customized page size, which is the size of the bounding box. Then we can insert some extra PostScript code to move the image so that it starts at the 0,0 coordinate (adjusting the bounding box accordingly). GhostScript then displays or converts the image just within the desired area, and no cropping is needed.

The transformations of the bounding box can be achieved using *epsffit*, which is part of Angus Duggan's `psutils` collection (CTAN:`support/psutils`); the page size change is most easily done using a Level 2 PostScript operator `setpagedevice`. Thus a PostScript file which starts:

```
%!PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: 135 528 284 668
...
```

needs to be transformed to something like:

```
%!PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: 0 0 149 140
<< /PageSize [149 140] >> setpagedevice
gsave -135 -528 translate
...
grestore
```

Here we have worked out the width and height of the enclosing rectangle (149 × 140 units), moved the origin down to 0,0 on the page, and set the page size. PostScript purists will shudder at the `setpagedevice` command, and point out that this is probably illegal in Encapsulated PostScript, but as long as we only use this file strictly in the controlled environment of GhostScript, we are safe enough. Figure 3 lists a simple Perl script which performs the necessary changes to a PostScript file for GhostScript to eat, without any need for *epsffit*[3]

Now that GhostScript is only rendering the desired area, we can use its builtin bitmap output facilities. The Unix or DOS command line:

```
gs -dNOPAUSE  -q -r100 -sDEVICE=tiffg4 \
 -sOutputFile=foo.tif foo.ps -c quit
```

[3] I am aware that it does not cope with an `(atend)` bounding box...

```
#!/usr/local/bin/perl
$bbneeded=1;
$bbpatt="[0-9\.\-]";
while (<>) {
 if ( /%%BoundingBox:(\s$bbpatt+)\s($bbpatt+)\s($bbpatt+)\s($bbpatt+)/ )
 {
     if ($bbneeded) {
         $width = $3 - $1;
         $height = $4 - $2;
         $xoffset = 0 - $1;
         $yoffset = 0 - $2;
         print "%%BoundingBox: 0 0 $width $height\n";
         print "<< /PageSize [$width $height] >> setpagedevice\n";
         print "gsave $xoffset $yoffset translate\n";
         $bbneeded=0;
    }
}
else {  print; }
}
print "grestore\n";
};
```

Figure 3: A Perl script to transform an EPS file for GhostScript

will generate a TIFF fax group 4 image (GhostScript does not support GIF output directly, for legal reasons) at 100dpi of just the imaged area of the PostScript file `foo.ps` with no further ado. GhostScript version 4 adds anti-aliasing facilities; using the Netpbm tools under Unix, we can create a variant GIF image, using the command line:

```
gs -r100 -dNOPAUSE  -q -sOutputFile=-  \
 -sDEVICE=pnm -dTextAlphaBits=4 \
 -dGraphicsAlphaBits=4 foo.ps -c quit | \
 ppmtogif -interlace \
 -transparent \#ffffff > \
 equation.gif
```

Figures 4 and 5 show the result of transformations with and without anti-aliasing. There is one remaining problem — the World Wide Web browsers can usually align images top, middle or bottom; but what if we have an image of some characters with descenders below the base line? Bottom alignment of the images places the bottom of the descenders on the baseline; top alignment is riduculous, and middle alignment is not quite right either. The answer is to use middle alignment, and make TEX lie to *dvips* (and thence down the chain) about the extent of the character; making its depth equal to its height, and then middle aligning it in the Web browser, has the desired effect. So how do we make TEX lie? Here is my suggestion:

```
\newsavebox{\@Fragment}
\def\Fragment#1{%
 \savebox{\@Fragment}{#1}%
 \@tempdima\ht\@Fragment
 \@tempdimb\dp\@Fragment
 \ifdim\@tempdima>\@tempdimb
  \dp\@Fragment\@tempdima
 \else
  \ht\@Fragment\@tempdimb
 \fi
 \fboxsep0pt
 \color{white}%
 \fbox{%
 {\color{black}%
  \box\@Fragment}%
 }%
}
```

I use the LATEX box framing command to ensure that *dvips* thinks the depth is there, with the same color trick as we saw earlier.

Unfortunately, there is a side effect — an HTML browser loading the resulting GIF image mid-aligns the image and sticks the 'ballast' white space into the line below, making an unsightly gap (see Figure 6, where the Greek etas have a small descender). With the current browser technology, there is little than be done about this. In practice, we will have to check first whether there *is* any descender; if so, we use the mid-align technique, and accept the gap;

$$-\Phi_0 \frac{\partial}{\partial \varphi}(\Phi_1 a \sin\varphi) - \Phi_1 \frac{\partial}{\partial \varphi}(\Phi_0 a \sin\varphi) - A_1 \left[\Phi_0 + \frac{\partial}{\partial a}(a\Phi_0)\right] \sin\varphi = -a\Phi_0 f \sin\varphi$$

Figure 4: LaTeX → dvi → EPS → GIF

$$-\Phi_0 \frac{\partial}{\partial \varphi}(\Phi_1 a \sin\varphi) - \Phi_1 \frac{\partial}{\partial \varphi}(\Phi_0 a \sin\varphi) - A_1 \left[\Phi_0 + \frac{\partial}{\partial a}(a\Phi_0)\right] \sin\varphi = -a\Phi_0 f \sin\varphi$$

Figure 5: LaTeX → dvi → EPS → GIF, anti-aliased

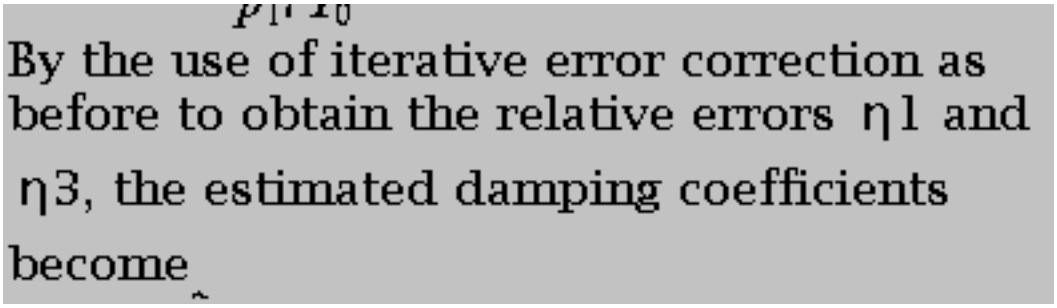

Figure 6: Mid-aligned GIF image in Netscape

if there is not, we can make a simpler process and use bottom alignment.

It is imperative, of course, that Web-making readers do not take these examples as 'recipes', without both a precise specification of the desired Web page, or an understanding of some of the basic image-processing techniques. The aim here has simply been to show how relatively trivial and efficient it is to create bitmap output from LaTeX and *dvips* using the free facilities of GhostScript.

◇ Sebastian Rahtz
Elsevier Science Ltd
The Boulevard
Langford Lane
Kidlington
Oxford
UK
s.rahtz@elsevier.co.uk