APPENDIX 7

Importing PostScript files

importing PostScript files:1

Very often you want to import one PostScript file into another. The one you want to import will quite possibly have been produced by another program, and may be a more or less generic PostScript file, so you have to be prepared for almost anything. You have to **encapsulate** the imported file so that it does not upset the environment into which it is imported.

labels:TEX:1

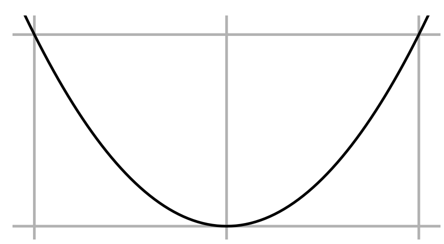
A special case of this is one of the most vexing tasks among all those a professional mathematician encounters, that of putting high quality TEX labels into a mathematical diagram. The most general task of this nature can indeed be daunting, but the exact one described here need not be.

I'll explain what to do by a simple example, then add remarks on fancier or more difficult variations.

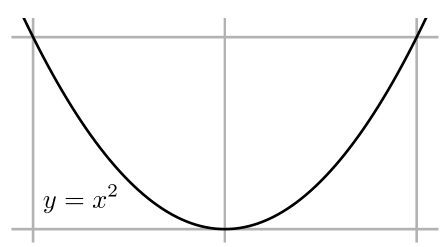
1. Labelling a graph

Let's suppose you have created the graph of a parabola:

```
72 dup scale
1 72 div setlinewidth
/N 100 def
newpath
/x -1.25 def
/dx 2.5 N div def
x dup dup mul moveto
N {
  /x x dx add def
  x dup dup mul lineto
} repeat
stroke
```



and now you want to add a label to it, so it becomes:



You could produce your own label in PostScript, but getting the fonts to look right, and getting the spacing right in mathematical text—for example the superscript in this case—is hard, and better left to some other program. I have used Donald Knuth's program TEX here to make up the label, and then `dvips`, a program written by John Hobby (once a graduate student of Knuth's), to produce from the TEX output a PostScript file. I'll say more about this process later, even though it is not really a PostScript matter. The important thing is that in the end I get a file called, say, `label.eps` which contains the PostScript code to write the text '$y = x^2$'. The basic idea is now simple—to include a line `(label.eps) run` in the PostScript file containing the parabola. There are two additional things to do, however: (1) take into account the different coordinate systems in parabola drawing and in the label file; (2) isolate effects of the program in the label file.

Dealing with the coordinates is simple, at least in normal circumstances. Dealing with the second will usually involve only annulling the effect of a possible `showpage` in the imported file, at least if the imported file is as well behaved as it ought to be. My new file, producing the label and the parabola, now looks like this:

```
gsave
% ...  parabola drawing as listed above ...
grestore

gsave
10 10 translate
-290 -695 translate
save /SavedState exch def
/showpage {} def
(yx2.eps) run
SavedState restore
grestore
```

You might not have encountered `save` and `restore` before. The command `save` does two things: (i) it puts a record of the complete current environment on the stack, and (ii) it saves the current graphics state as `gsave` does. Thus `save /SavedState exch def` (*why not* `/SavedState save def`*?*) defines `SavedState` to be the current state. The command `restore` has one argument, a complete state in the format produced by `save`. The point of using them here, among other things, is to take care of possible `showpage` commands in the imported file, but then bringing back the normal definition of `showpage` after the file is read. Unless `showpage` is disabled, an importing program might try to turn a page after reading each import. This technique is especially important in versions 8.0 and later of Ghostscript, *which seem to add implicitly a* `showpage` *command to EPS files it reads*, even if there isn't one there originally.

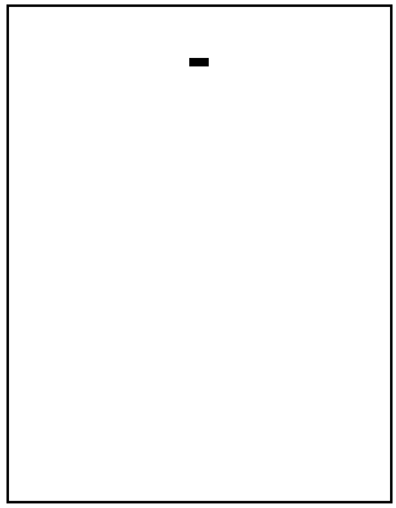
Another and perhaps simpler way to turn off `showpage` is to use a temporary dictionary:

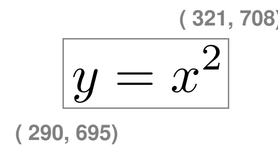
```
1 dict begin
/showpage {} def
(yx2.eps) run
end
```

save:2 **restore:2** But `save ...  restore` is a more flexible technique. Now all you have to know is where the mysterious
PostScript:encapsulated:2 numbers 290 and 695 come from. But that's easy. The file produced by `dvips` is an **encapsulated PostScript file**
bounding box:2 (or EPS file), which means, among other things, that it contains near the beginning a line

```
%%BoundingBox:  290 695 321 708
```

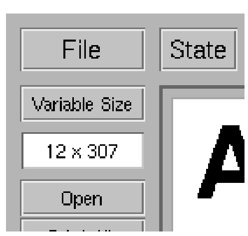
to advertise to applications that want to use it (and that includes us!) what the boundaries of its drawing area—its **bounding box**—are. These are the coordinates of the lower left and upper right corners of that box. The image that TeX and `dvips` produced, in other words, was intended to be placed on a page as on the left, with the label sitting inside its bounding box as in the close-up on the right:





The line `-290 -695 translate` therefore sets the lower left corner of the imported image at the origin of the figure it is being imported to. So dealing with different coordinate systems is very simple, actually. All you need to do is figure out the bounding box of the imported file. The line `10 10 translate` placed before the importation thus has the effect of locating this corner at $(10, 10)$ in the importing file. You might very well want to do some other transformations to that imported file. Suppose that in addition you want to scale the import in place, for example: make it

```
10 10 translate
4 4 scale
-290 -695 translate
```



Order is important here, as it always is in coordinate changes.

Deciding where to place the imported file may not be straightforward, and usually requires some fiddling around. Most Ghostscript viewers track the location of the mouse in default coordinates to help you out in this task. The viewer I use, for example, records those coordinates—here $(12, 307)$—in the upper left corner. This is often extremely useful. In order to take advantage of this feature, you *must* restore default coordinates before importing files.

What I have said so far will handle most imports, but sometimes a somewhat more robust technique is required. The principal new feature is to restore the default graphics environment before importation. Also, messy stack handling by the imported files has to be allowed for.

```
/BeginImport {
  % save the current state
  save /SavedState exch def
  % save the sizes of two stacks
  count /OpStackSize exch def
  /DictStackSize countdictstack def
  % turn off showpage
  /showpage {} def
  % set up default graphics state
  0 setgray 0 setlinecap
  1 setlinewidth 0 setlinejoin
  10 setmiterlimit [] 0 setdash newpath
  /languagelevel where
```

```
  {pop languagelevel 1 ne
    {false setstrokeadjust false setoverprint} if
  } if
} bind def

/EndImport {
  count OpStackSize sub
  dup 0 gt { {pop} repeat} {pop} ifelse
  countdictstack DictStackSize sub
  dup 0 gt { {end} repeat} {pop} ifelse
  SavedState restore
} bind def
```

followed by code like this:

```
BeginImport

% --- import stuff here

EndImport
```

2. Importing TEX text

TEX:PostScript files produced by:4

In this section I want to say more specifically about how to import text produced by TEX, although it is not directly connected with PostScript. Those not familiar with TEX may skip it. I also restrict myself to plain TEX, since the unfortunately more popular variant **Latex** does all kinds of extra formatting I don't want to deal with.

Generally, you will want to import small fragments of mathematics. The only serious requirement for doing this correctly is that, at least in plain TEX, you must put the line `\nopagenumbers` at the start of your file. This means that there will be no secret writing on your output. Then TEX your file as usual. To produce PostScript output, you should run `dvips` on the corresponding `.dvi` file, but with the `-E` option:

```
dvips -E x.dvi -o x.eps
```

if `x.tex` is your TEX file. This produces an EPS file with the bounding box data written into it. If you use `dvips` without the `-E` option, you will get a full page with your TEX on it, whereas the `-E` option will produce a bounding box just covering the text you want.

There is one problem with this procedure. The file produced will be quite large. For example, the file `yx2.eps` I used as an example in the first section was about 17,000 bytes long! The reason is that `dvips` puts in a number of header files and font definitions, and they take up space. These can be amalgamated, but I'm not going to say anything about that here, except to raise the issue. Anyway, in the modern world 17,000 byes is not all that large, although if you have a lot of labels the redundant code will add up.

If you are using TEX, then you will probably not be able to avoid meeting the PostScript files produced by `dvips`. It's also likely that sooner or later you'll want to modify one of these by hand, so I'll say something here about their structure. In fact, I'll look at the one responsible for the label '$y = x^2$' dealt with above. It starts off with something like

```
%!PS-Adobe-2.0 EPSF-2.0
%%Creator:  dvips(k) 5.86 Copyright 1999 Radical Eye Software
%%Title:  yx2.dvi
%%BoundingBox:  290 695 321 708
%%DocumentFonts:  CMMI10 CMR10 CMR7
%%EndComments
```

```
%DVIPSWebPage: (www.radicaleye.com)
%DVIPSCommandLine: dvips -o yx2.eps -E yx2.dvi
%DVIPSParameters: dpi=600, compressed
%DVIPSSource:   TeX output 2003.03.12:1926
```

comments:document structure:5
PostScript:encapsulated:5 EPS:5

These are all comment lines. The initial one declares that this file conforms to the conventions for document structure mentioned in an earlier appendix. It also declares that this is an encapsulated PostScript file. The comments beginning with %% are part of the document structure (see Appendix 3). Next comes two sections like this:

```
%%BeginProcSet: texc.pro
%!
/TeXDict 300 dict def TeXDict begin/N{def}def/B{bind def}N/S{exch}N/X{S
...
end
%%EndProcSet
%%BeginProcSet: texps.pro
...
%%EndProcSet
```

This just contains two fragments of PostScript code that were imported by dvips to set up its own dictionaries. Next comes

```
%%BeginFont: CMR7
%!PS-AdobeFont-1.1: CMR7 1.0
%%CreationDate: 1991 Aug 20 16:39:21
% Copyright (C) 1997 American Mathematical Society.   All Rights Reserved.
...
%%EndFont
```

which just defines a font to be used. More fonts follow. The structure of fonts is a very specialized part of PostScript, a world pretty much isolated from the rest of the language. Among other things, font files usually contain large chunks of almost indecipherable code.

fonts:PostScript:5

Finally, the last few lines of the file are

```
TeXDict begin 40258437 52099154 1000 600 600 (yx2.dvi)
@start /Fa 205[33 50[{}1 58.1154 /CMR7 rf /Fb 194[65
61[{}1 83.022 /CMR10 rf /Fc 134[41 47 120[{}2 83.022
/CMMI10 rf end
%%EndProlog
%%BeginSetup
TeXDict begin
%%EndSetup
1 0 bop 1830 183 a Fc(y)26 b Fb(=)d Fc(x)2032 148 y Fa(2)p
eop
%%Trailer
end
userdict /end-hook known{end-hook}if
%%EOF
```

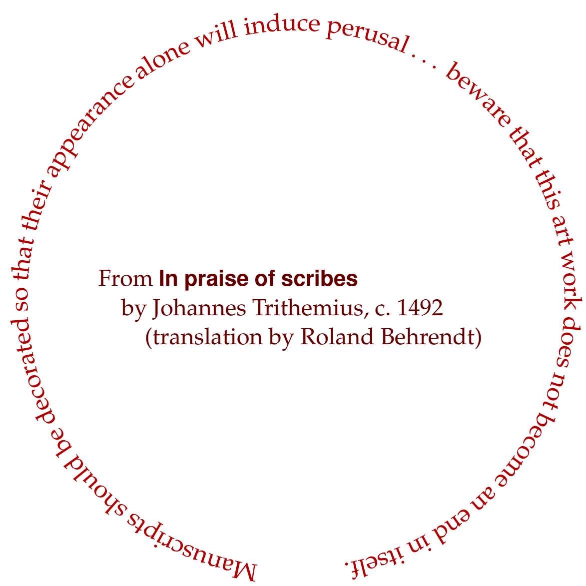
This is the real meat of the file. Some abbreviations are defined, and then the actual typesetting is done in the single line

```
1 0 bop 1830 183 a Fc(y)26 b Fb(=)d Fc(x)2032 148 y Fa(2)p
```

This doesn't make a lot of sense immediately, but actually it's pretty simple. The program `dvips` abbreviates font changes for efficiency, as well as other commonly used PostScript commands. It defines `bop` to begin pages. So all that's going on here is that characters from different fonts are being placed on the page, and the characters being placed are those in the string '$y = x^2$'. As with a modern army at war, then, almost all the effort goes into logistics!

3. Fancy work

You can in fact apply any 2D transformation to the image produced by any PostScript code—for example that in a file to be imported—without modifying the code itself, at least in most circumstances. Why would you want to do this? Well, this is one of those things about which it is said, "If you have to ask . . . " Any serious book on mysticism will tell you that the highest levels of experience are not accessible to everyone. **Trithemius:J.:6**



The techniques involved here are much less trivial than others in this book. The difficult point here is that you do not want to examine the internals of the code of the transformed image, except perhaps taking into account its size, as specified by its bounding box. This means that *some of the very basic PostScript drawing commands must be modified*, instead. There are a number of approaches to this problem—the basic choice is whether to modify the paths in the imported code as they are laid down or as they are drawn. In the first scheme the operators `moveto` etc. are redefined, and in the second `stroke` etc. I'll choose here the second method. It requires quite a bit less programming, is slightly more efficient, and is also somewhat more flexible.

There are only a few operators involved in actually realizing a path in PostScript, as opposed to building it. The obvious ones are `stroke`, `fill`, and `clip`. In addition is the operator `show`, which essentially fills in the path made up by a string of characters in whatever the current font is. There are others—principally those concerned with **user paths**, which I have not talked about—but I'll ignore them here. In redefining the drawing operators, it must be kept in mind that the coordinate system may be changed continually in the segment of code to be transformed. For this reason, a base coordinate system in which the transformation is to be applied must be fixed. **user paths:6**

One concern that has to be taken into account is that transforming a path will mean transforming the pieces of that path, which may be line and curve segments. Doing this assumes that these pieces are small enough that the transformation is well approximated by an affine transformation on them. This may not be valid for the original path, so we must allow for path subdivision.

Yet another concern is that we want to be able to escape from our redefinitions, since we might want to keep on drawing normally after we have drawn the transformed imported file. This will be handled by including the redefinitions in a special dictionary which can be pushed and popped on and off the dictionary stack with `begin` and `end`.

All these things are handled in a package `transform.inc`. It has procedures

integer	`set-sd`	sets subdivision depth
procedure	`set-transform`	defines the transform
PS matrix	`set-base`	sets coordinates for the transform to be applied in
	`subdivide`	subdivides the current path
	`path-transform`	applies the transform at hand to the current path

Just below is the part of the code that manages the final drawing. The file `trithemius.eps` contains the text *Manuscripts* ... *itself*, all laid out in a line. The file `trithemius2.eps` contains the text to be placed inside the circle, just as is. The bounding box data is taken from the file `trithemius.eps`, to be used to transform it correctly.

As for the transform, it knows nothing of the contents of the file it is transforming, and simply wraps a rectangle around a circle, starting at an angle A_0 and ending at an angle A_1, which in this case have been chosen at $-90° \pm 7°$.

```
 % define the base coordinate system
/B matrix currentmatrix def
(transform.inc) run
 % --- first the outer text ---
 % set up the transform
 % the bounding box of the imported file
/llx 81 def
/lly 713 def
/urx 389 def
/ury 719 def
 % the dimensions of the box we are displaying
/boxwidth urx llx sub def
/boxheight ury lly sub def
 % the angles where the circular text starts and ends
/A0 270 7 sub def
/A1 -90 7 add def
 % radius of the circular text
/radius 100 def
 % proportion of a full circle the text takes up
/factor A0 A1 sub 360 div def
 % the length it will take up on the circle
/truelength factor 3.1416 mul 2 mul radius mul def
 % the transform itself
/f { 1 dict begin
  /y exch lly sub def
  /x exch llx sub def
  /T 1 x boxwidth div sub A0 A1 sub mul A1 add def
  /R 100 boxheight 2 div sub y truelength boxwidth div mul add def
  T cos R mul T sin R mul
end } def

gsave
transformdict begin
```

```
    /f load set-transform
     % set the base coordinate system
    B set-base
    1 dict begin
    /showpage{} def
    (trithemius.eps) run
    end      % the temporary dictionary
  end      % transformdict
  grestore
   % --- now the inner text ---
  0.92 dup scale
  -91 -683 translate
  58 98 translate

  1 dict begin
  /showpage{} def
  (trithemius2.eps)  run
  end
```

References

fonts:type 1:8 1. Adobe Systems, **Adobe Type 1 Font Format**, 1990. This is the original edition of the definitive document. Newer versions are available on line.

Trithemius:J.:8 2. Johannes Trithemius, **In Praise of Scribes**, translated into English from **De Laude Scriptorum** by Roland Behrendt, Coronado Press, 1974.