

Graphics: an overview

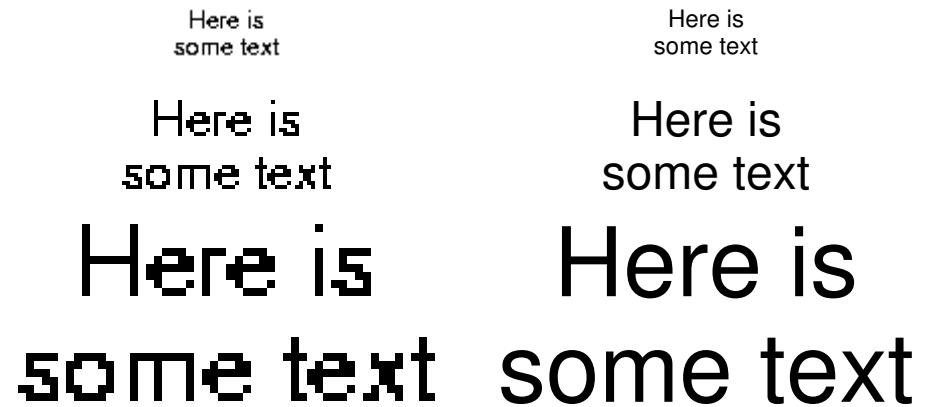
MJ Rutter
mjr19@cam

Easter 2017

Typeset by FoilT_EX

The Basics

Images can be *vector*, *bitmap*, or a combination of both. The difference is familiar:



We shall first consider bitmaps.

Matrices

A bitmap is simply a two-dimensional array of values, stored in a *raster* sequence: row 1, then row 2, then row 3, etc.

A *pixel* (picture element) may be represented by a single bit (black and white images), a byte (greyscale, *paletted* colour), three bytes (full colour), four bytes (full colour with transparency), or perhaps other amounts of data.

One byte can store 256 different values, so 256 different levels of grey for greyscale images, and of red, green and blue independently for RGB images.

Bitmaps map in an obvious fashion to most output devices. Most screens are simply 2D arrays of pixels, probably around two million of them. An A4 sheet of paper as seen by a 600dpi laser printer is about a 36 million pixel bitmap.

Resizing: up

A bitmap is an array of pixels, containing no information about underlying objects such as letters. Once placed in a bitmap, a letter is just a part of that collection of pixels.

To expand a bitmap by 20%, one could repeat every fifth row and column. Bad:

Some text
Some text
Some text

The first line is the ‘original’, the second has every fifth row / column doubled, the third is ‘done properly’ by re-rasterising the font at the higher resolution.

The second line is really bad. The two ‘e’s are different, for the fifth column fell in different places with respect to them, and the ‘m’ is very asymmetric.

Resizing: down

Shrinking by 20% by deleting every fifth row is just as disastrous.

Some text
Some text
Some text

More sophisticated schemes exist, mainly relying on shades of grey. They are not very good either.

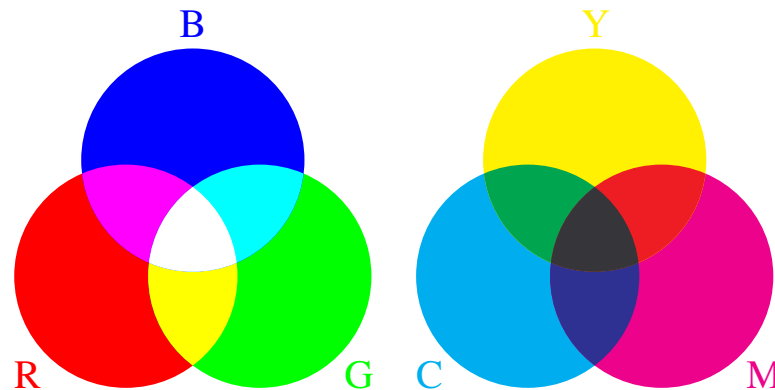
In general, expand bitmaps by integer ratios by repeating every row / column as necessary, reduce by (small) integer ratios by averaging, and don't attempt anything else!

Expansion by large, non-integer ratios also works well. Otherwise printing these low-resolution images on this 600 dpi printer would fail too...

RGB or CYM(K)?

The human eye has receptors for red, green and blue light, and VDUs produce red, green and blue to stimulate those receptors. A human cannot tell the difference between yellow, and a mixture of red and green.

Printer inks work by absorption, not addition. Cyan absorbs red, yellow absorbs blue, and magenta absorbs green. So cyan and magenta absorb red and green leaving blue.



More CYM(K)

A mixture of all three printer inks should be black, but using a separate single black is cheaper, and gives a better quality black, as the mixture can be a muddy brownish colour. Hence the K (blacK) in CYMK.

So:

CRTs use RGB from phosphor emission

LCDs use RGB from dyes backlit by various light sources

Paper uses CYMK and reflected light

OHPs use CYMK and transmitted light

Is it any wonder colours never look the same on different media?

Just as most printers use a separate, though theoretically unnecessary, black, some 'photo' ink-jet printers use six inks, not four, for added clarity.

Media Madness

The following grey bar goes through 256 levels of grey. The pure white part is the width of the line beside the 'white' label, and similarly for pure black.



Humans can distinguish more than 256 levels of grey, and will rightly find this to be unconvincing as a smooth scale. The above bar will look different when printed on our B&W and colour printers, and different again if photocopied.

Here the same for red through to white



Some formats now use ten, twelve or even sixteen bits per 'channel', or colour value. Whether many output devices can reproduce colours with this accuracy is another matter. TCM's monitors all run at eight bits per channel...

Compression

Most bitmaps are highly compressible, sometimes by a factor of ten or more.

When storing bitmaps in a file, or transmitting them as part of web pages, compression is highly recommended.

Compression algorithms divide into two camps: lossless and lossy.

Lossless algorithms are suitable for any data. Compression followed by decompression gives precisely the original data. Lossy algorithms give nearly the same data back after decompression. They are thus hopeless for payroll data, but can be useful for certain images.

In general lossless algorithms work well for ‘line art’ and diagrams (where there are large areas of precisely the same colour), whereas lossy algorithms are better for photographs, where there is a lot of unimportant variation between adjacent pixels.

Lossless algorithms

The LZW (GIF) and zlib (PNG) lossless compression algorithms exploit correlations in data. Both work on a wide range of input, not just images. Text also has obvious correlations – after the letters ‘Saint ’, the sequence ‘Margaret’ is highly likely, etc.

But LZW and zlib look for correlations in a 1D stream, whereas in an image there are obvious correlations in 2D. The PNG algorithm makes some use of this 2D nature. It can encode each line either independently, or as a set of differences from the previous line. Indeed, there are five different predictors (including the null predictor) that can be applied to each row in a PNG file independently.

If we assume that optimal zlib compression is simple, then with five choices of predictor algorithm per line in a bitmap, there are 5^n ways of PNG-compressing a file where n is the number of rows.

In practice, there is no simple method for finding the best zlib compression for a data stream either.

Lossy Compression

Lossy compression is about making approximations which Humans are unlikely to notice. The JPEG compression algorithm, more accurately called the DCT algorithm, is quite good at this, especially considering it is quite old – it was released in 1992.

- Break image into 8x8 pixel squares.
- Discrete Cosine Transform each square.
- Divide coefficients by scaling factors.
- Huffman encode resulting data.

As integer arithmetic is used throughout, the third step is (very) lossy. High frequency components are heavily scaled down, and the great majority of coefficients end up as zero after this step, and the rest tend to be very small. The Huffman coding works well on this very non-uniform distribution.

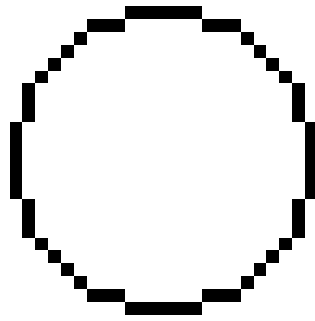
DCT tricks

Humans are approximately equally sensitive to R, G and B channels. However, they are not equally sensitive to Y, C_r and C_b channels, where Y is luminance (the bit left if the image were converted to greyscale), and C_r and C_b the excess red and blue.

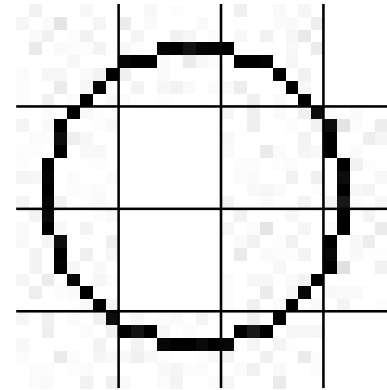
$$\begin{aligned} Y &\approx 0.3R + 0.59G + 0.11B \\ C_r &\approx 0.5R - 0.42G - 0.08B + 0.5 \\ C_b &\approx -0.17R - 0.33G + 0.5B + 0.5 \end{aligned}$$

The C_r and C_b channels can be compressed much more lossily than the Y channel. Indeed, it is common to convert the image to this form, and then compress Y as above, but for the two chrominance channels to halve the resolution before one even starts.

DCT problems

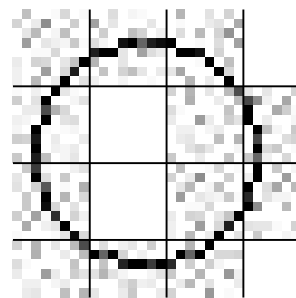


Before



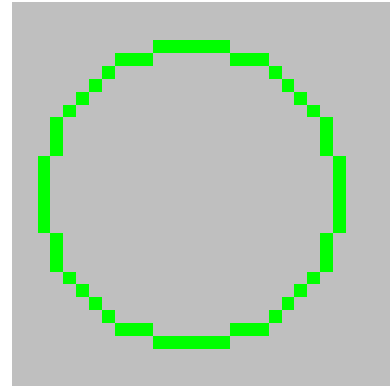
After

A grid showing the 8x8 blocks into which the image was broken up before the DCT stage has been superimposed on the second figure. Note that the blocks which contained no non-zero Fourier components initially do not suffer from erroneous non-zero components after the transform. Unconvinced? Try the higher-contrast version below:

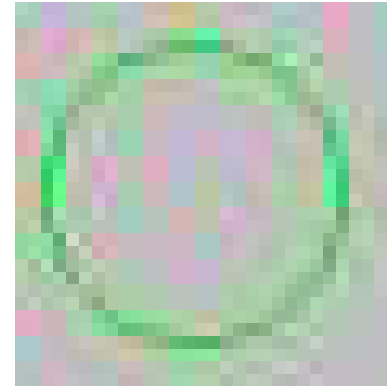


For an image of this size, 30x30 pixels and one bit per pixel, DCT is hopeless anyway, for the required header of quantisers and Huffman tables is (much) bigger than the original uncompressed data.

Worse DCT problems



Before

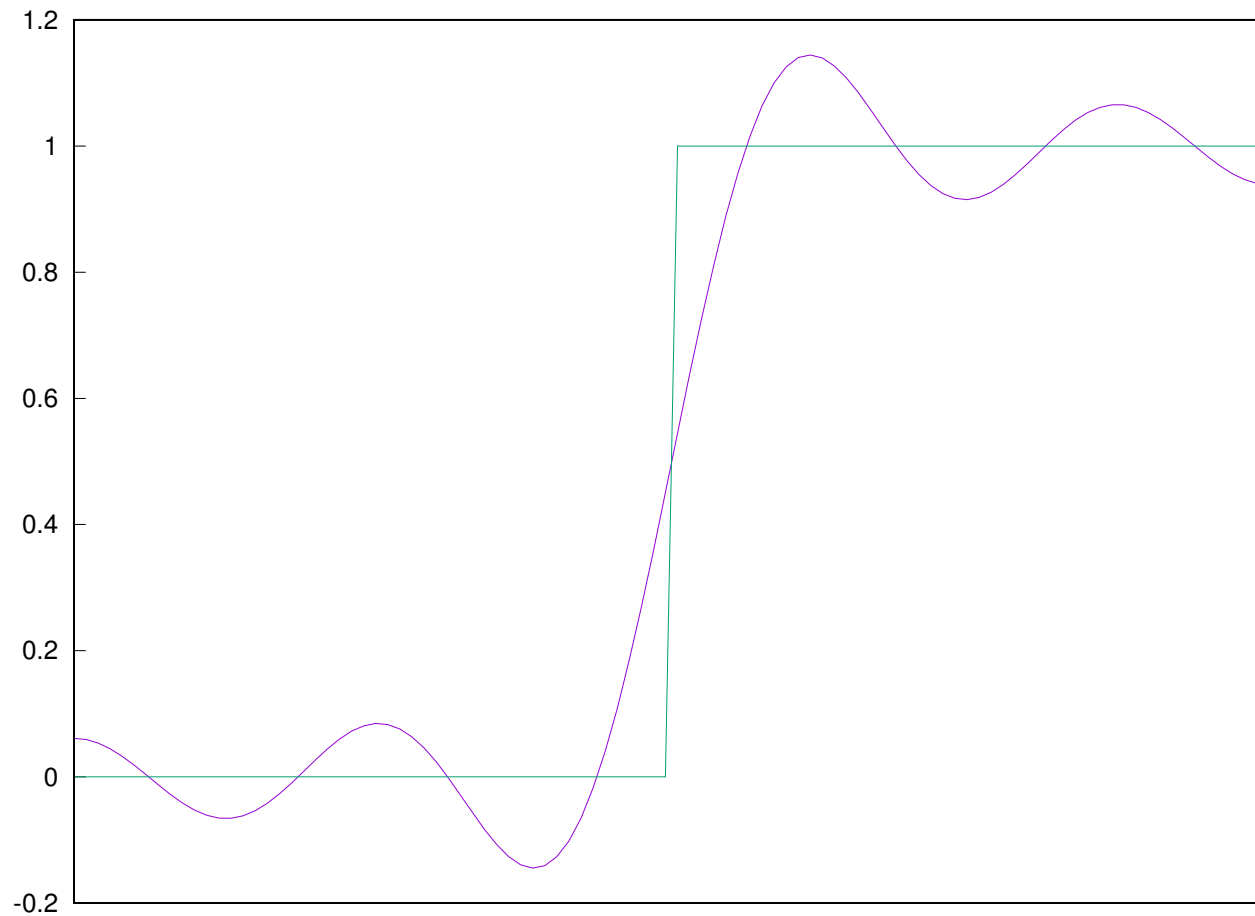


After

This image uses the same set of defaults as the previous, and shows that the losses in the colour information are rather greater than in the luminance. Note the appearance of pink, despite the absence of any pink in the original, which contained just green and grey.

Repeatedly saving an image as a JPEG, loading it, modifying it, and resaving, only ever makes matters worse. One should aim to save as JPEG just once, at the end of all one's manipulations.

Gibbs Phenomenon



(Not strictly identical, given that the DCT would invert exactly if the coefficients were stored with full accuracy.)

DCT successes



900KB uncompressed



74KB JPEG



42KB JPEG



30KB JPEG



20KB JPEG



9KB JPEG

Nasty blocks in the sky are clear in the 9KB version. They are also present in the 20KB version, and, more subtly, the 30KB version. However, this 640×480 image has been compressed by a factor of twenty with no obvious loss of quality. PNG managed a factor of just two, and absolutely no loss of quality. (Displaying six images of this size on a 1920×1080 projector, or on an A5-sized piece of paper, is nonsense anyway.)

Resolution, resolution, resolution

Many people use much higher resolutions than necessary, leading to enormous files. If an image is destined for a talk such as this, the whole screen is only 1920×1080 , and it is unlikely that one wants an image of more than 2M pixels. With suitable compression, this is unlikely to end up as more than 200KB.

For printing, larger images are reasonable, but not that much larger. A colour printer may claim a resolution of 600 dpi, but one is wasting one's time offering it more than about 150 dpi with a photographic image. Line art would be reasonable at 600 dpi, but line art should be sent in vector form anyway. Given that the final image is unlikely to be much more than 5 inches wide, and 4 inches tall, one may conclude that half a million pixels is again a reasonable number. Megapixel images are necessary only if one needs to fill most of an A4 sheet *and* one expects people to stare at the detail.

For web pages, lower resolutions are fine, for it is rare for a single image to be displayed at more than half the screen width, and often much less.

Progressive?

Some bitmap formats have a ‘progressive’ version which permits a low-quality representation of the full image to be displayed before all the data are received. This is intended to improve web pages on slow connections. Although data are not transmitted twice, progression does impact compressibility.

GIF’s approach is to transmit every eighth row, then the intervening fourth rows, then the intervening second rows, then the remaining odd rows. This has a marginal negative impact on compression.

PNG uses a 2D Adam7 scheme, which first transmits every eighth pixel horizontally and vertically, then after six more passes, completes the image which has effectively been broken into 8x8 squares. It has a significant negative impact on compression as it breaks correlations within a row (save for odd-numbered rows, which get transmitted in full on the last pass), and it impacts negatively prediction from the previous row, as the different passes predict from the $n - 8$ to $n - 2$ row, and never the $n - 1$ row.

JPEG rearranges the coefficients so that rather than transmitting all coefficients from one block, then the next, it transmits the low-frequency components of all blocks together, then progressively higher frequencies. This can actually improve the compression ratio.

Displaying a progressive image is always more work than displaying a non-progressive one, as pixels get updated multiple times.

File Formats

Many different graphics file formats, with differing strengths, have independently evolved.

Most start with a *signature* or *magic number* which identifies the file type, then have a header describing the resolution of the image, the colour depth, the compression algorithm (if appropriate), the palette (if appropriate). They may also provide for a text comment to be attached to the image.

Then there is the data section, usually stored as left to right top to bottom scans, and compressed in the manner the header indicated. The byte order could be RGB, BGR, or RRRR...GGGG...BBBB...

Some files provide for *interlacing*, which enables a coarse representation of the full image to be displayed after reading just the first part of the file. This is useful for slow WWW downloads.

Most files can be identified by some form of *magic number*. Sometimes it is obvious: GIF files start with the characters 'GIF'. Other times less obvious: JPEG starts with the hex codes ff, d8, ff.

PNM

Portable aNyMap.

Old and simple, and comes in six subtypes.

PBM: Portable BitMap, B&W only.

PGM: Portable GreyMap, greyscale.

PPM: Portable PixMap, RGB colour.

Unsuited for storage due to lack of compression, but ideal as an intermediate format which all programs can be expected to understand, and ideal for writing from your own programs. Can include textual comments.

My PPM

The binary form can be written with this, hideously inefficient (hint: example – don't use in real life) C.

```
fprintf(img, "P6\n%d %d\n255\n", width, height);
for(h=0;h<height;h++) for(w=0;w<width;w++) {
    fwrite(&red[h][w], 1, 1, img);
    fwrite(&green[h][w], 1, 1, img);
    fwrite(&blue[h][w], 1, 1, img);
}
```

There is also a bloated, pure ASCII, form for Fortran programmers:

```
write(*,10)width,height
10 format('P3 ',I6,I6)
write(*,*)255
do h=1,height ; do w=1,width
    write(*,'(3I4)') red(w,h),green(w,h),blue(w,h)
enddo ; enddo
```

For greyscale, replace 'P6' with 'P5' (or 'P3' with 'P2') and write one byte per pixel. Health warning: I have not even checked that the above compile, they are merely intended to demonstrate that one can write (some) standard graphics formats without exotic support libraries.

GIF

Graphics Interchange Format.

Invented by CompuServe in 1987 and clarified in 1989.

Uses LZW compression, and limited to 256 colours (free choice from a palette of 16 million).

Provision for text comment. Provision for 1D interlacing. Provision for animation.

Needing no more than 16KB of memory for decompression was a design criterion.

Offers little that PNG does not do better: kept alive by inertia.

Small animated GIFs are still popular on web pages though.

PNG

Portable Network Graphics.

Uses zlib compression, and supports predictors.

Paletted or unpaletted, and has full alpha support for 256 levels of transparency in 32 bit images, as well as GIF's single transparent colour trick. Supports 16-bit greyscale images too.

Provision for text comment. Provision for 2D interlacing.

Generally the preferred format for bitmaps.

TIFF

Tagged Image File Format – I hate this format.

It provides for no compression, RLE, ITU (three varieties), LZW (with optional predictor), DCT, both possible bit orderings, greyscale, paletted (4 or 8 bit), RGB, CYMK and YC_bC_r colour spaces, multiple images per file, and much else besides.

Writing such a file is easy. Being able to read all possible TIFF files is hard: few products achieve it.

The result is that TIFF divides its features into ‘baseline’, which all decoders should support, and optional ‘extensions’. Unfortunately the baseline features include RLE and ITU Group 3 compression, but nothing which will actually compress any 24 bit image. So a 24 bit TIFF image is either uncompressed, or compressed in a manner which not all TIFF readers will read.

JPEG

Joint Photographic Experts Group. (1992)

Grey scale or 24 bit colour DCT compressed image.

Provision for text comment. Provision for 2D interlacing. Provision for uncompressed ‘thumbnail’ representation.

Like TIFF, JPEG actually includes many sub-formats, not all of which are even DCT compressed. Like TIFF, a ‘baseline’ feature set is described. Unlike TIFF, everything useful is in the ‘baseline’ set, and no-one sees any need to support any of the extensions.

(Save that progressive JPEGs are not in the baseline specification.)

JPEG and Quality Factors

Baseline JPEG permits two separate quantisation tables (scale factors for the DCT coefficients), one for luminance, and one shared by the two chrominance channels.

It also permits the chrominance channels to be sampled as frequently as luminance, or half or a third as frequently, with different frequencies in x and y .

The two quantisation tables contain 64 numbers each, and there are nine possible subsampling procedures. This gets presented to the user as a single 'Quality Factor'. This may not mean the same thing for all programs...

Not all are equal

Some PNG-writing programs are lazy, writing everything as 32-bit images with fixed predictors. Better choice of depth and compression can produce images which are 10 to 20% smaller, and still completely lossless.

Some JPEG-writing programs (especially those embedded in cameras) use a fixed ‘Huffman’ table, rather than modifying the table according to the data actually present in the image. The ‘-optimise’ option of ‘jpegtran’ undoes this inefficiency losslessly. (The optimised table requires an extra pass through the data on compression. Decompression is identical.)

Vector Formats

Bitmap images are rarely of use to theoreticians. They are useful for photographic images, but are poor for ‘*line art*’, that is images containing lines and text such as one would draw with a pen (rather than paint with a brush). Line art is better expressed in a resolution-independent vector language, and is clearly made up of elements such as lines and curves (possibly filled) with sharp edges.

DCT compression destroys such diagrams, due to the inability of cosine transforms to cope with step functions. But JPEG stands for Joint *Photographic* Experts Group: it was not designed for line art at all.

Beware *Distiller*, which has a habit of recompressing losslessly compressed bitmaps with DCT compression.

All three of the formats below (PS/EPS, PDF and SVG) allow one to include bitmaps within a vector image.

PostScript

PostScript was invented and trademarked by Adobe in 1985. It is used by most laser printers, \LaTeX , and much else besides.

It is a programming language, albeit one primarily designed for graphics. It has loops, variables and procedures, can be written by humans, and can be expressed in plain text. It strongly supports vector graphics, although it can now offer reasonable support for bitmaps too.

Like most languages, it has evolved. Level 2 PostScript appeared in 1990 and level 3 in 1999. Today (2007) it is very rare to find a PostScript device which does not support level 3.

PostScript is designed to be device independent. A B&W device must accept colour commands, and do its best. Again there is no concept of device resolution in PostScript: a device will draw the specified lines and curves as carefully as it can.

EPS: Encapsulated PostScript

An EPS file is a piece of PostScript designed to be included within another piece of PostScript. It can be thought of as a subroutine rather than a complete program. The rules for making EPS files are more restrictive than for PostScript, and turning EPS into PostScript is trivial, whilst the reverse is not possible in the general case.

EPS files:

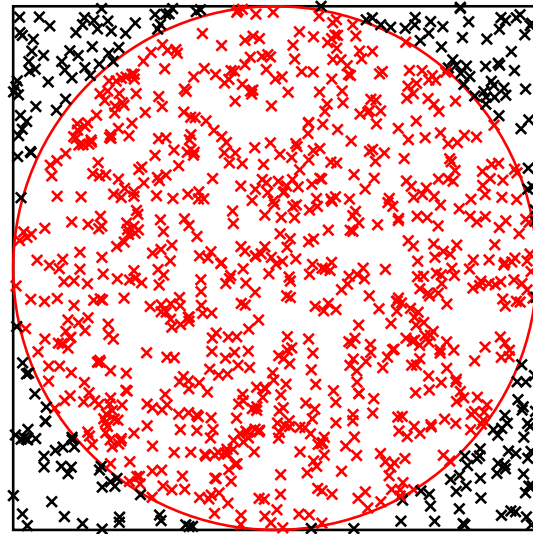
- Must not contain multiple pages nor perform media selection
- Must start `'%!PS-Adobe-n.n EPSF-n.n'`
- Must have a `%%BoundingBox: comment`
- Must not change the stack nor the global dictionaries
- Must not use `initclip` or `initgraphics`
- Should not execute `showpage`

There is rarely a good reason for using PostScript, rather than EPS, for saving a figure.

Printers may (should?) refuse to print EPS, but converting EPS to PS is trivial (in TCM, `eps2ps` does so).

Pi

$$\pi \approx 3.14$$



The above piece of PostScript picks 1000 points at random, draws them, and prints the value of π that results. It should give a different answer each time it is run.

PDF

Portable Document Format, or PDF (also a trademark of Adobe), is a poor relative of PostScript. It is simpler to interpret than PostScript, and almost always uses a binary form which makes it smaller than PostScript. Unlike PostScript, it can contain hyperlinks, but it is not a language: it has none of PostScript's conditional or loop operators.

There is no well-defined way of including PDF files in other documents: Adobe has not defined an 'encapsulated PDF.' However, if the PDF file contains just a single page, and is reasonably well-behaved, things tend to work. This is what pdf \LaTeX and `dvipdfm` rely on!

PDF is quite good for publishing papers on the WWW: usually much better than HTML.

PDF's Problems

Despite being more modern than PostScript, there are quite a lot of versions of it. At least fifteen (1.0 to 1.7, 1.7 EL 1, 3, 5, 6, and 8, PDF/X and PDF/A), and more are being developed.

Whereas in PostScript an included EPS file effectively has its own namespace, there is no analogue of this in PDF, where there is a single, global, namespace. As a PDF file consists of many numbered 'objects', to include one PDF file in another one must renumber all objects (and all references to them) to avoid clashes. This is hard. To include a conforming EPS file in a conforming PS document, one does not need to change one byte of the EPS file.

Editing a PDF file with a text editor is impossible, even if it does not use compression.

Unfortunately PDF has won as far as popularity is concerned. If one wishes to write vector graphics from one's code, writing PostScript is much easier though – I have done it from shell scripts!

Validating a PDF file is hard. Writing a PDF file which displays different text on different viewers, and gives warnings on almost none, seems easy. (Try writing text with no font set – invalid, some viewers display the text in a default font, some do not display.)

PDF's killer features

PDF supports 'proper' transparency for objects (i.e. 256 levels of transparency). PostScript does not: it simply offers a one-bit mask.

This means that many PDF files cannot be readily converted to PostScript. Converters often resort to converting the PDF image, or large parts of it, to a bitmap, and then converting that to a large, non-vector, PostScript file.

PDF supports clickable links (both internal, and external web links). PostScript does not, so PDF is much better for any form of e-reader.

Some PDF to PS converters err on the side of writing a partial PDF interpreter in PostScript, then passing on chunks of PDF. This produces inelegant, and sometimes incorrect, PostScript.

Older printers do not support PDF, but only PS.

SVG

Scalable Vector Graphics, an XML-based vector format which is the only vector format that most web browsers support for vector images within a page.

Its support for bitmaps is limited to supporting embedded JPEG or PNG images, which must be Base64 encoded to make them ASCII.

(PostScript uses Base85 encoding to support binary data in an ASCII file. Whereas Base64 needs four characters to store three bytes of data, Base85 needs five characters to store four bytes, as $85^5 > 256^4$.)

The **TCM** logo on TCM's web pages is an SVG.

PS vs PDF vs SVG

PS

```
%!  
/Helvetica 12 selectfont  
144 360 moveto  
(Hello, PostScript world) show  
showpage
```

Writing programs which produce PostScript, whether from python, Fortran, C or bash, is fairly easy.

PDF

```
%PDF-1.3
```

```
1 0 obj
<</Length 50>> stream
BT /F0 12 Tf 144 360 Td
(Hello, PDF world!) Tj ET
endstream
endobj
```

```
2 0 obj
<<
  /F0 << /Type /Font /Subtype /Type1
  /BaseFont /Times-Roman >>
>>
endobj
```

```
3 0 obj
<</Type /Catalog
/Pages 4 0 R
>>
endobj
```

```
4 0 obj
<</Type /Pages
/Count 1
/Kids [5 0 R]
>>
endobj
```

```
5 0 obj
<</Type /Page
/Parent 4 0 R
```

```
/Resources <<
  /ProcSet [/PDF /Text]
  /Font 2 0 R
>>
/MediaBox [0 0 596 841]
/Contents 1 0 R
>>
endobj

xref
0 6
0000000000 65535 f
0000000010 00000 n
0000000108 00000 n
0000000193 00000 n
0000000242 00000 n
0000000299 00000 n
trailer
<<
  /Size 6
  /Root 3 0 R
>>
startxref
439
%%EOF
```

Byte offsets and counts in **red**. object references (to single global namespace) in **magenta**. Fonts are also named in a single global namespace.

PDF objects are usually compressed, so are not Human-readable.

SVG

```
<svg xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink">

<text x="20" y="40"
      style="font-family: Helvetica; font-size: 12;"
>Hello SVG world!</text>
</svg>
```

Much closer to PostScript than PDF in friendliness. But like PDF, not a language.

Conversions

The DCT compression algorithm is supported by PostScript, PDF and SVG. There is no reason to decompress and recompress a JPEG file when converting to these formats, and attempts to do so are very likely to result in a loss of quality.

Unfortunately many programs do decompress and recompress, including convert.

Try `bmp2eps` instead, which doesn't, and will write EPS, PS, SVG or PDF.

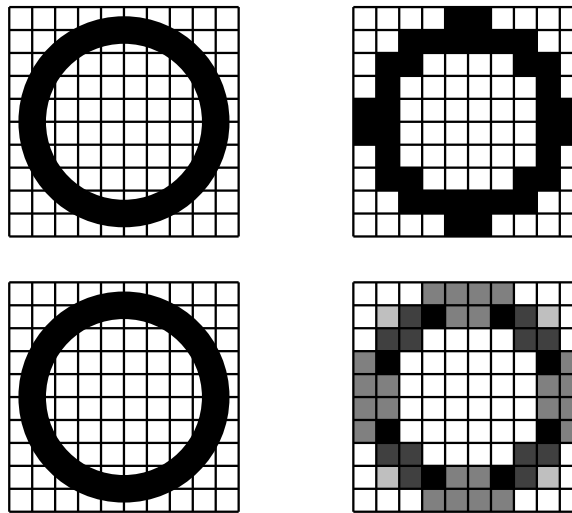
Converting a PNG or JPEG file to PDF should produce something of about the same size. To EPS or PS should produce an expansion of about 27%, plus an overhead of about 1KB. To SVG an expansion of about 33%, plus an overhead of under 1KB.

If the PNG or JPEG compression has been done carelessly, then much smaller expansions, and sometimes compression, can be achieved.

Antialiasing

The trivial way of converting a vector-based image to a bitmap calculates which pixels are hit by a given line, and, if the line covers more than 50% of the pixel, the pixel acquires the line's colour, and if not, not. In this case, colours in the bitmap are all colours found in the original image.

A more sophisticated method proceeds as above, but at twice (or three times) the required resolution. Then the resolution is halved (or thirded) to that required by averaging the colours of each group of four (nine) pixels in order to create intermediate shades. The effectiveness of the result depends on the angle subtended by a pixel at the eye of the Human observer...



Web Optimisation

Reduce resolution (but remember that ‘retina’ displays may use a 2x2 cluster of pixels for each ‘CSS’ pixel).

Reduce JPEG quality. This is a trade-off with resolution reduction – a higher resolution but lower quality might be preferable for a given filesize.

Do optimise the final output, by using programs which compress efficiently, and strip unnecessary comments etc., yet make no change to the quality of the actual image. Including JPEG ‘thumbnail’ preview images is particularly pointless.

Don’t use progressive/interlaced GIFs and PNGs unless the image is likely to take several seconds to download. If it is likely to take several seconds to download, worry that it is too big anyway (would a small image which is a link to the full resolution image be better?). Use of progressive/interlaced GIFs and PNGs will increase total download time, and require more processing, even though they do provide more entertainment for Humans as the image downloads.

Do wonder whether an unaliased image at twice the resolution (with the client reducing the resolution) might be both small and better than an aliased image. Aliasing can increase the colour count, and thus reduce the effectiveness of the compression algorithms.

Some examples

I ran 'bmp2eps -png' on all .png images in people's web_profile directories. A selection of results are:

Before	After	change
52453	31416	-40%
110546	102768	-7%
444306	376263	-15%
29496	16936	-42%
56960	58437	+3%
65926	59600	-9%
45359	45701	+1%
51926	25163	-52%

The 52% size reduction came from an image saved as 24 bit which actually contained just three unique colours.

The bmp2eps program claims to be good, but not perfect. Clearly it is not always optimal, but other programs are less optimal.

JPEG examples

I ran ‘`jpegtran -optimise`’ on all .jpg images in people’s web_profile directories.

In most cases there was no change in size.

In many cases there was a reduction of around ten percent.

The biggest reduction was from about 800KB to about 100KB. That image came from Photoshop, and contained a 256x256 pixel base64-encoded preview image, along with a 658x652 pixel four-component JPEG. Use of a four-component JPEG is brave – ImageMagick fails to display it correctly, and some browsers might also struggle.

A few images became larger. This is because they were encoded as progressive JPEGs, which is the JPEG equivalent of 2D interleaving. It is not in the baseline standard, but often produces slightly better compression. I tend not to use it as not all JPEG-supporting software can read progressive images, though all serious web browsers can. By default, `jpegtran` outputs without progressive encoding.

Bitmap Extraction

The program `pdimages` will extract all bitmap images from a PDF file, saving those which were DCT compressed as JPEGs, and those which were losslessly compressed and ppms.

I have written `psimages` which does something similar for PostScript and EPS files. It works by redefining PostScript's image operator...

Both have man pages. Usage is typically

```
pdimages -j foo.pdf x
psimages foo.ps x
```

where `x` is the root of the filename used for the many output files (which will be named `x-0001.ppm`, etc.).

EPS or PDF to bitmaps

Various programs will convert EPS/PS/PDF to a given resolution of bitmap. If the original simply contains a bitmap, it is better to extract that directly. If not, then

```
eps2gif -png -res 1000 in.eps > out.png
```

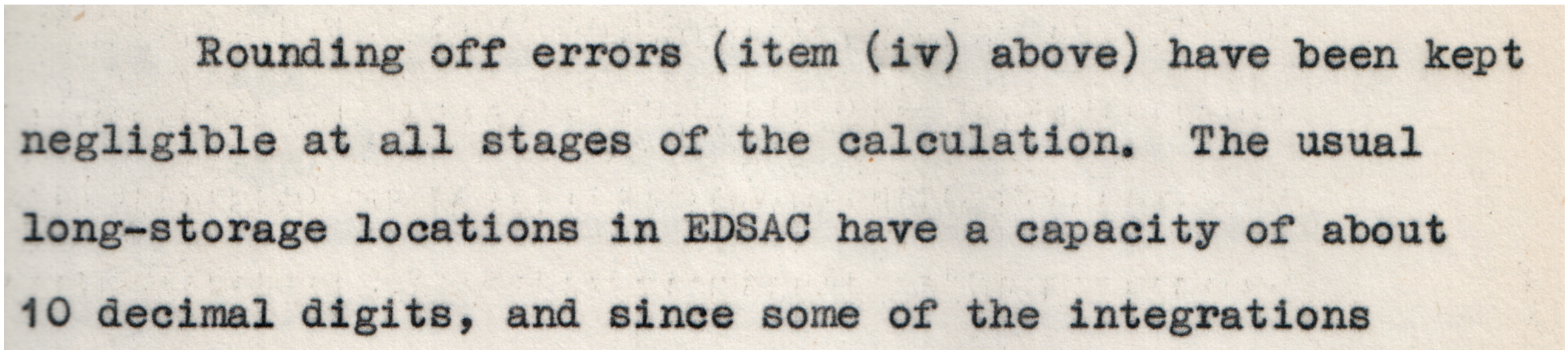
works with EPS and most single-page PostScript files, and PDF I tend to convert to EPS/PS first, and then convert as above, but other methods work too, such as `convert`.

Using screen-grabbing programs can work reasonably, but does limit the resolution to approximately the screen resolution (depending on how one's window manager like windows bigger than the screen, and then grabbing their off-screen contents).

If one has an electronic copy of the original, printing then scanning is never required.

Scanning

The last resort. Colours get washed out, horizontal lines of text become skewed, and noise in the background tends to be more uncompressible than the data. If scanning mostly line-art and text, one can adjust the colour curves to force the background to pure white. The result will look better, and be a lot smaller.



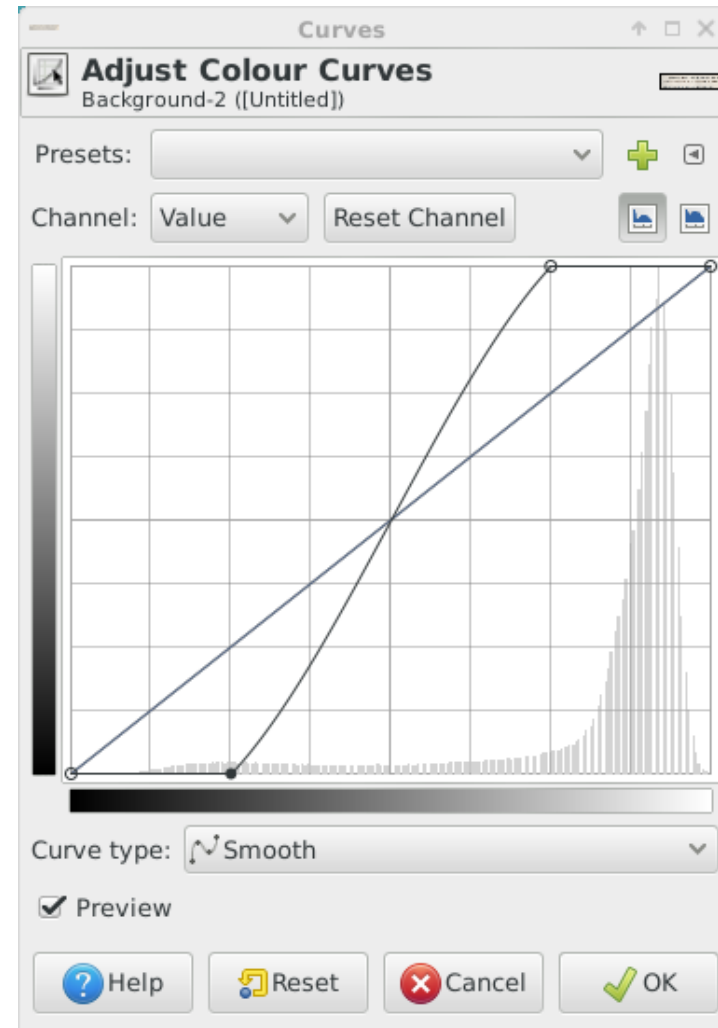
Rounding off errors (item (iv) above) have been kept negligible at all stages of the calculation. The usual long-storage locations in EDSAC have a capacity of about 10 decimal digits, and since some of the integrations

Rounding off errors (item (iv) above) have been kept negligible at all stages of the calculation. The usual long-storage locations in EDSAC have a capacity of about 10 decimal digits, and since some of the integrations

Improve!

GIMP

The second version is so much easier to read there is no excuse for not tidying things up like this. The first was 24-bit colour with no post-processing, and was 1.3MB as a PNG. The second is 8-bit greyscale, and 80KB as a PNG (still quite large for what it is, but the image is a rather unnecessary 1800 pixels wide).



Mixtures

Many images are best expressed as a mixture of a bitmap and vector graphics. Anything which looks like an annotated photograph, including most coloured surface plots with axes, would be an example. Persuading graph-drawing packages that this is how they should produce their output can be hard. They often try to vectorise everything, producing surfaces containing many millions of poorly-compressed vector triangles, each with its own colour defined.

If the output is EPS, post-processing may be possible. Essentially one tries to split the file into two, one with all the parts which ought to be kept in vector form retained, the other containing just the parts which should be kept as a bitmap. Then one rasterises the part which should have been a bitmap, and rejoins the files. Tedious, but for final publication purposes this gives the best tradeoff between size, display time, and image quality.

I have a python script (`pm3d-shrink`) which tries to do this automatically to Gnuplot 4.0's pm3d output.

Other Mixtures

The more GUI alternative is to convert an image to a bitmap, remove the parts which would be better in a vector form (mostly text), and then re-insert using some GUI program which will support a vector/bitmap mix: e.g. inkscape, scribus, dia, xfig.

Again a manual process best done on the final version of an image.

(Inkscape is very PDF-friendly, dia very SVG-friendly, and xfig prefers EPS.)

Keep the Source

As a final comment, whenever producing images for papers, theses, etc., keep sufficient of the source files that one can reproduce the figure in a slightly different form (with a line added, with units changed, with the widget rotated, with the resolution changed, etc.). If one retains just the final image, then reuse in different contexts is much harder.