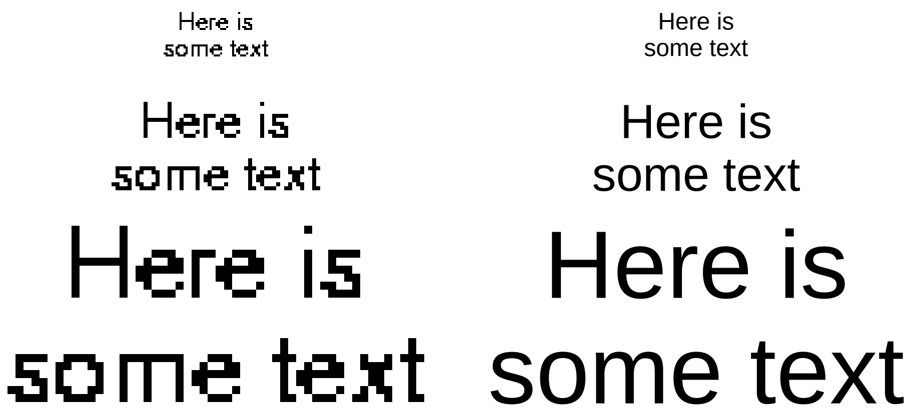
Improving Your Image

Michael Rutter
mjr19@cam.ac.uk

Lent 2006

The Basics

Images can be either *vector* or *bitmap*. The difference is familiar:



We shall first consider bitmaps.

Matrices

A bitmap is simply a two-dimensional array of values, stored in the obvious fashion: whether stored in a file or memory, the pixels are stored in a *raster* sequence: row 1, then row 2, then row 3, etc.

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

Uncompressed data sufficient to fill a typical screen (1280×1024, three bytes per pixel) or A4 sheet of paper (600dpi, one bit per pixel) is approximately 4MB.

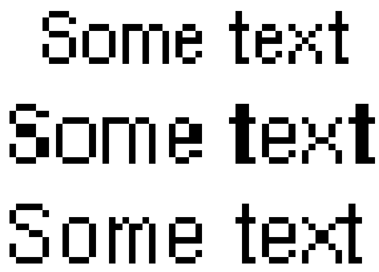
pixel = <u>pi</u>cture <u>el</u>ement

Large area colour printers can have alarming memory requirements, and play clever tricks to reduce this.

Resizing

A bitmap is just 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 greater collection of pixels.

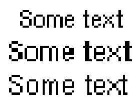
To expand a bitmap by 20%, one could repeat every fifth row and column. The below shows why this is bad.



The first line is the ‘original’, the second has every fifth row / column doubled, the third is ‘done properly.’

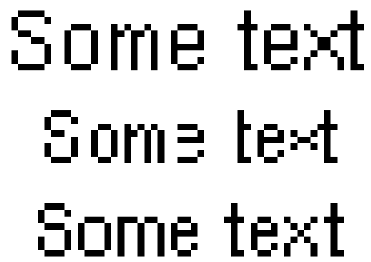
The second line is really bad. The two ‘e’s are different, for the fifth column fell in different places there. The ‘S’ and ‘t’s are rather poor too, and the ‘m’ is very asymmetric.

The third line starts again from the original font. It introduces new subtleties into the top of the ‘m’ and elsewhere. Below the same shrunk somewhat, where it will appear slightly different.



More resizing

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



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…

Palettes

Using 256 levels for Red, Green and Blue in each pixel gives over 16 million colours: more than humans can distinguish. However, three bytes per pixel can be expensive.

Choosing just 256 of the 16,777,216 possible colours, and having a small look-up table from colour number to actual RGB value shrinks the image to just one byte per pixel. The look-up table is the *palette*, and can contain any combination of colours.

A sixteen colour (or four bit) palette allows one to use just half a byte per pixel.

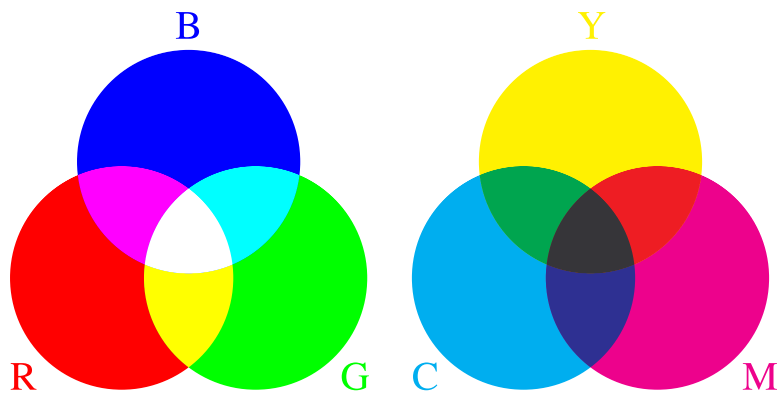
The old VGA computer standard is 640x480 with a sixteen colour palette, and requires just 150KB of video memory.

Now memory is so cheap it is common to have four bytes per pixel, one byte being unused in order to keep things aligned on word boundaries.

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 absorbtion, 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
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.

On OHPs the far red end will look rather dark, and the brightest red will not be found at an RGB triplet of (1,0,0), but rather around (.75,0,0). Too much ink simply absorbs too much light.

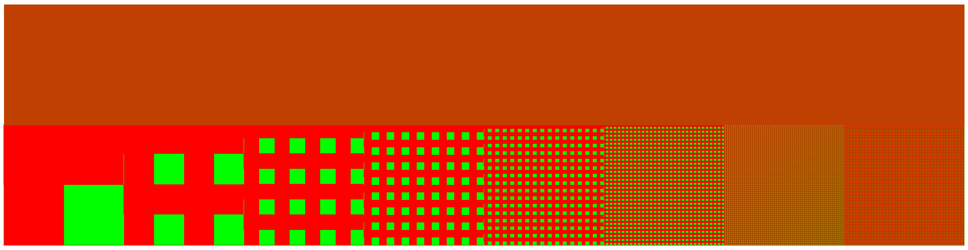
This effect is much less pronounced on paper, when one is viewing reflected light not transmitted light. The type of printer (laser, ink-jet, dye sublimation, wax transfer) can be an important factor.

Colour Reduction

Most images can be reduced to just 256 colours without humans noticing. However, it is necessary to do this with care.

If few colours are available, *dithering* can simulate others.

Below is a bar with RGB values of 0.75, 0.25, 0, and below it a pattern of red and green squares.



At some point, humans will cease seeing red and green squares on the lower bar, and see a shade of orange instead.

The highly periodic pattern used here is a bad choice for dithering, and can cause interesting artifacts. A random scatter of green and red squares in the ratio 1:3 would be better. A commonly-used random algorithm is named Floyd-Steinberg in honour of its inventors.

Unfortunately this example misbehaves with many PostScript interpreters. The upper bar is a solid colour with RGB co-ordinates of (0.75,0.25,0), but the lower right square rarely appears to be the same shade.

Optimal Reduction

By choosing the palette to suit the image, most images can be reduced to around 256 colours whilst retaining sufficient colour detail to keep humans happy.

However, choosing the optimal palette and optimal dithering algorithm is not trivial. The images on the next slide illustrate what can go wrong, and are the result of reducing a 199 colour bitmap to 16 colours.

`ppmquant` appears to have chosen the 16 most common colours in the original image. Wrong. There are two near-identical shades of blue, which should be replaced by a single colour, and it misses the red entirely.

`convert` manages to include the red, but its dithering is ugly.

`gimp` does really rather well, at least in this case.

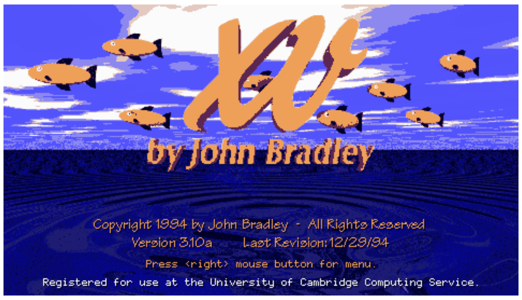
Original

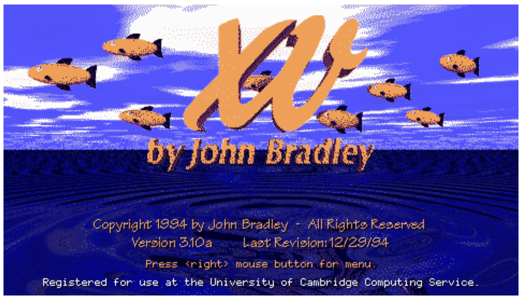
Lines below are:

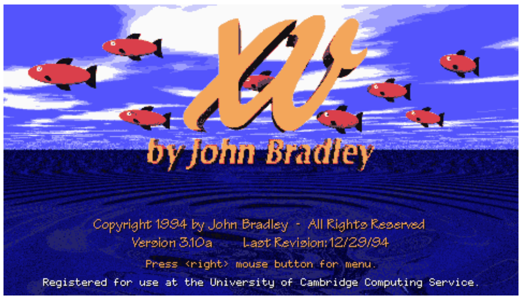
ppmquant (pnmutils)
convert (Image Magick)
gimp (The GIMP)

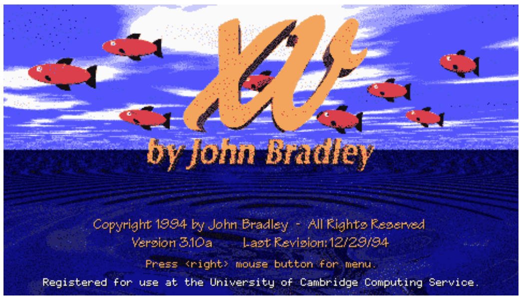
Undithered

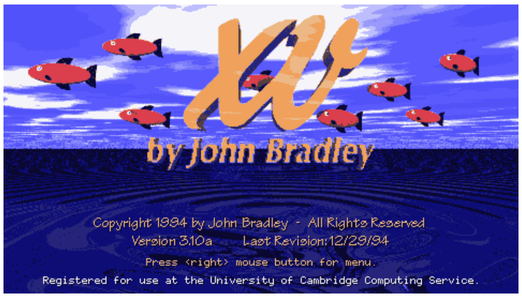
Dithered

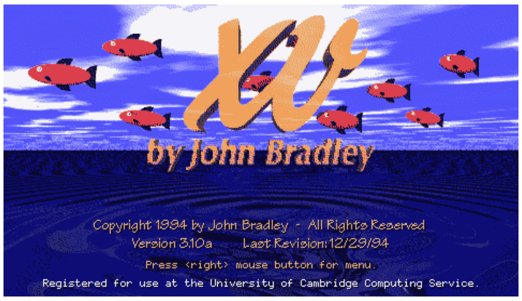












Anti-Aliasing

The use of shades of grey, rather than just black and white, can make text dramatically more readable on low-resolution displays. As the result is very dependent on the angular size of a pixel subtended at one's human observer, it is hard to demonstrate.

For a better demonstration, view a file with `xdvi` and note the change on pressing '0' followed by 'G', then '1' followed by 'G' to restore. Or in `gv` choose 'State' then 'Antialias', or in `acroread` choose 'File' 'Preferences' 'General' 'Smooth text and images'.

Compression

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

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.

Many (poor) algorithms consider the image data as a 1D stream, and make no use of correllations perpendicular to the raster direction. Thus one can get a different compression ratio if one rotates an image through 90°!

0D Algorithms: Huffman coding

Normally each data item takes the same amount of space: one byte. Huffman coding provides shorter bit sequences for common values, at the expense of longer sequences for rarer ones.

If all 256 byte values occur equally frequently, Huffman can do nothing. In English text, where a dozen lower case letters are very common, and fewer than 100 symbols are used in total, Huffman can win.

Whereas RLE relies entirely on correlations to gain compression, Huffman ignores correlations and relies on the underlying distribution being non-uniform. The output is not a byte sequence, as the tokens emitted are not all eight bits long.

Unlike RLE, Huffman encoding is a two-pass process. The first pass calculates the distribution, and thus the optimal representation for each data value. The second pass encodes. Used by the UNIX command `pack`. If the first pass is omitted, and the distribution simply guessed, the result is often called a pseudo-Huffman code.

Variable Length Codes

Huffman, or pseudo-Huffman, codes use a prefix-free coding for data. Consider storing marks from IA labs: these clearly lie between 0 and 4, so could be encoded in three binary bits with the usual codes of 000, 001, 010, 011 and 100. However, some scores are (much) more common than others, so one could encode as 0, 100, 101, 110 and 111, or maybe 0, 10, 1100, 1101 and 1110, or maybe 00, 01, 100, 101 and 110.

Decoding the obvious binary form is trivial: each item is three bits long. The variable length form is also easy: one stops as soon as one reaches a valid code. If 01 is a valid code, no other code starts 01 – that is what prefix-free means.

Huffman, whilst working on his PhD in 1951, found and proved optimal an algorithm for determining what coding sequence would be best for any given statistical distribution of data in which each datum is independently drawn from a constant distribution.

Of course, when storing Huffman encoded data, one also needs to store the corresponding table of codes for decompressing it.

1D Algorithms

Run Length Encoding

16 $\times$ black pixel, 4 $\times$ white pixel, 100 $\times$ black pixel, 20 $\times$ uncompressed pixels…

Useless for text, poor for photographs, excellent for line drawings.

In combination with a pseudo-Huffman encoding, RLE gives ITU Group 3 FAX encoding for bilevel (i.e. B&W) images.

LZW Encoding

Lempel, Ziv and Welche. Subject to now-expired patents.

Looks for repeated short sequences. Can compress text reasonably well, usually beats RLE for any images. The UNIX command `compress` uses this algorithm.

The output is not a sequence of bytes, but rather a stream of bits containing variable-length tokens. The input will be interpreted as bytes, the output is not. No means of storing literally uncompressible sections, leading to a >12.5% inflation of uncompressible data.

More 1D, and More Than 1D

Zlib

Never patented, invented by Lempel and Ziv.

Tries to encode data by finding the same sequence in the previous 32K, and then encoding it as an offset and a number of bytes to copy, both the offset and length being encoded using a pseudo-Huffman code. Can store uncompressible sections literally. The commands `gzip` and `zip` use this algorithm. Almost always better than LZW.

1.5D

Some algorithms allow one to describe a scan line in terms of differences from the preceeding line. This is not truly 2D: the two directions in the image are being treated very differently. However, at least the direction perpendicular to the raster is being considered!

Group 4 FAX encoding uses this technique, building on the RLE with pseudo-Huffman of Group 3. The technique is also often used with zlib.

Fundamental Theorems of Lossless Compression

You can't win.

No algorithm can compress all possible inputs. (Consider feeding it its own output, and watch it vanish into nothingness.)

You can't break even.

Every algorithm which compresses something will also expand something.

Hence parallels are drawn between thermodynamics and entropy, and data compression and information content.

Assume exists an algorithm which compresses something and expands nothing.

Assume a sequence of n bits is compressed to p bits, with $p < n$. The 2^{p+1} possible sequences of p or fewer bits must all compress to p or fewer bits, but one of the 2^{p+1} sequences to which they could compress expands to the original n bit sequence, so one is trying to form a one-to-one mapping from 2^{p+1} sequences onto $2^{p+1} - 1$ sequences. Impossible.

If compressing a scanned graph, remember that the noise created by the scanning process may contain more entropy than the original figure, and thus, unless the image is 'cleaned', it will not compress well.

Lossy Compression: DCT

Discrete Cosine Transform (used by JPEG).

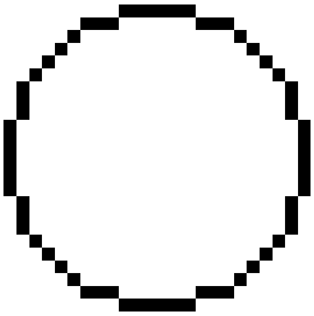
- Break image into 8x8 pixel squares.
- DCT each square.
- Divide coefficients by scaling factors.
- Huffman encode resulting data.

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.

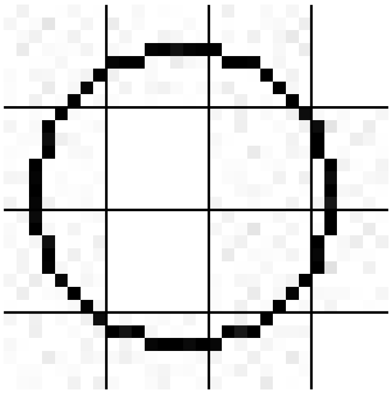
The initial DCT is lossless (ignoring rounding problems), but the integer division stage effectively reduces the precision to which the coefficients are stored, thus introducing errors. For some image types, this works well. For others, not.

This compression method is really 2D: it makes use of correlations in the x and y directions.

DCT problems

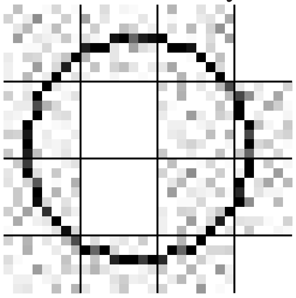


Before



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.

DCT tricks

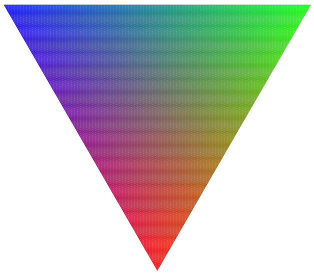
Humans are approximately equally sensitive to R, G and B channels. However, they are not equally sensitive to Y, $\mathrm{C_r}$ and $\mathrm{C_b}$ channels, where Y is luminance (the bit left if the image were converted to greyscale), and $\mathrm{C_r}$ and $\mathrm{C_b}$ the excess red and blue.

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

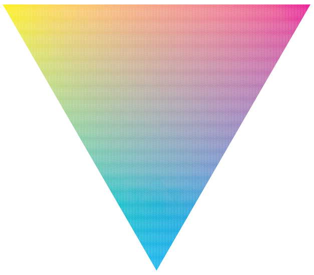
The $\mathrm{C_r}$ and $\mathrm{C_b}$ channels can be compressed much more lossily than the Y channel. Indeed, it is common to convert the image to this form, and then compress Y as above, but for the two chrominance channels to halve the resolution before one even starts.

The formula for Y above suggests that the eye is rather more sensitive to green than blue. However, here one is more concerned by sensitivity to changes than to absolute values.

Different Colour Spaces



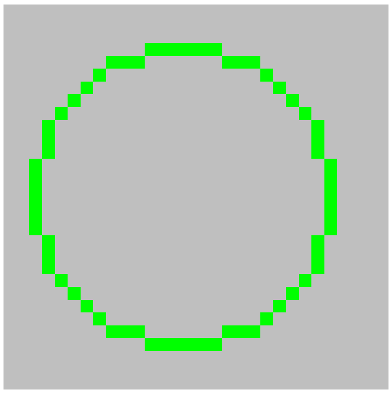
RGB with $\mathrm{R} + \mathrm{G} + \mathrm{B} = 1$



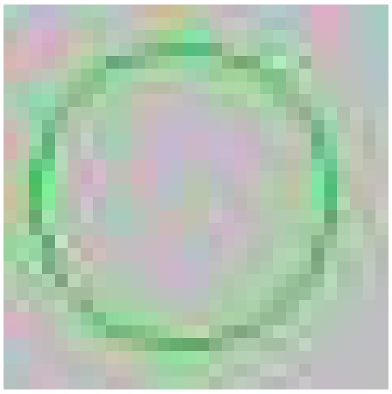
CYM with $\mathrm{C} + \mathrm{Y} + \mathrm{M} = 1$

$\mathrm{YC_rC_b}$ with $\mathrm{Y} = 0.4$

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.

DCT successes

Original Image

640x480 24 bit

900KB raw

450KB png

74KB JPEG

42KB JPEG

30KB JPEG

20KB JPEG

9KB JPEG

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.

No paletted form exists, and the image is stored uncompressed. The data are usually binary, but a (rarer) form of each time exists using ASCII data.

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 the following, 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.

BMP

BitMaP.

Invented by Microsoft and IBM, with Windows and OS/2 disagreeing on the details. Exists in the following types:

Uncompressed: Palette of 2, 16 or 256 colours, or 24 bit colour.
RLE compressed: 16 colour palette, 256 colour palette.

The header and palette storage differ between the Windows and OS/2 versions.

One format, 12 subformats.

Many programs which can read BMP files cannot read the compressed versions, and the compression algorithm is not very good anyway. Not recommended!

GIF

Graphics Interchange Format.

Invented by CompuServe in 1987 and clarified in 1989.

Uses LZW compression. Paletted with palette length 2^n with $1 \leq n \leq 8$.

In the 1989 version, a single colour can be marked as being transparent.

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.

Patents covering the compression algorithm expired in 2003/4.

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.

Provision for text comment. Provision for 2D interlacing.

Animations via the almost-identical MNG format.

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.

TIFF tries to resolve this by dividing 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. LZW used to be in the baseline specification, but was removed due to patent problems.

Few TIFF users understand they have a choice between poor/absent compression, or mediocre compression and a file which not all programs which can claim to read TIFF will actually be able to 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.

JPEG actually includes many sub-formats, not all of which are even DCT compressed. However, a ‘baseline JPEG’ standard is described, which all programs claiming JPEG compatibility must fully support, and which has the two formats above.

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…

A quality factor of 100 usually means use unity for all scale factors, and do not undersample chrominance. But it might not. Indeed, some products have quality factors ranging from 0 to 1, or 0 to 100 with 0 being best. Some products treat undersampling and quantisers completely separately.

Summary

Name	PNM	BMP	GIF	PNG	JPEG
Bilevel	PBM	Y	Y	Y	N
Greyscale	PGM	Y	Y	Y	Y
Palleted	N	Y	Y	Y	N
Full Colour	PPM	Y	N	Y	Y
RLE	N	Y	N	N	N
LZW	N	N	Y	N	N
ZLIB	N	N	N	Y	N
DCT	N	N	N	N	Y
Text comments	Y	N	Y	Y	Y
Animations	N	N	Y	MNG	N

Future Formats

Various new formats are technically good, but have yet to achieve a critical mass of usage.

JBEG Joint Bilevel Experts' Group. A much better lossless b&w compressor than the previous ITU Fax encodings.

JPEG2000 A wavelet-based lossy compressor from the JPEG group. Fewer nasty artifacts than the previous JPEG, but may contain some lurking IPR problems, so little used so far. Still deals poorly with sharp edges, producing a similar 'ringing' to JPEG.

DjVu An excellent, but little-adopted, lossy compressor from AT&T, optimised for scanned documents. Can store OCRed text along with the image, and does not suffer from the JPEG 'ringing' problems, yet offers better compression.

Line art

Such 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 the MacOS X clipboard, which likes converting all sorts of things to JPEG without asking.

Beware the PDF *Distiller* program, which does likewise.

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 (2002), everything you will find supports level 2, and everything you buy will 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.

DSC

Writing spaghetti code in PostScript is easy, and manipulating the result hard. So Adobe introduced the idea of Document Structure Comments.

A DSC-compliant piece of PostScript has a prologue, followed by one or more independent pages, and an epilogue. The page boundaries are marked by easily-found comments (a line starting '`%%Page:`'), and page deletion, extraction and reordering is trivial.

In a non-DSC piece of PostScript, such tricks can be very hard to impossible.

The DSC comments all start with %% on a new line. Interesting ones include:
%%Pages: total number of pages in file
%%DocumentRequiredFonts: fonts which are needed but not included
%%Creator: program which wrote the file
%%BoundingBox: `llx lly urx ury` region of page used

if the information reads '(atend)' it means the real information is to be found in a comment at the end of the file.

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:

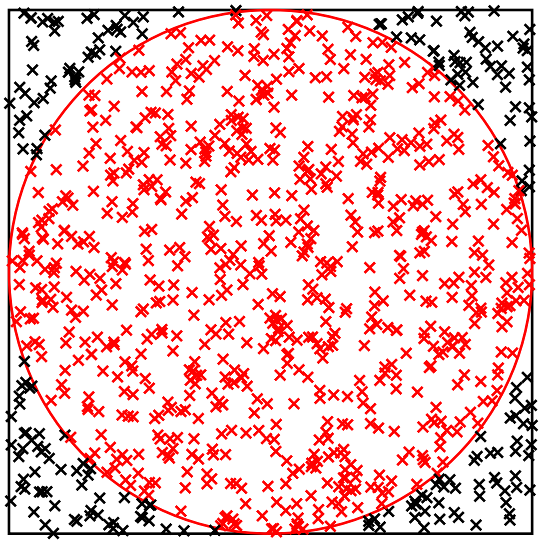
- Cannot contain multiple pages
- Cannot perform media selection
- Must be DSC compliant
- Must start '`%!PS-Adobe-n.n EPSF-n.n`'
- Must have a `%%BoundingBox:` comment
- Must not change the stack nor the global dictionaries
- Must not use `initclip` or `initgraphics`
- Should not execute `showpage`

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

EPS files, in general, do not print: they should be converted to PostScript first. If one ignores the `showpage` restriction, creating a file which is both valid EPS and PostScript is possible.

Pi

$\pi \approx 3.136$



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.

PostScript and Bitmaps

A nasty, non-scalable, bitmap is the last thing a PostScript document needs, but occassionally it is necessary, and PostScript can cope, which means that an EPS file can mix bitmap and vector items in the same image.

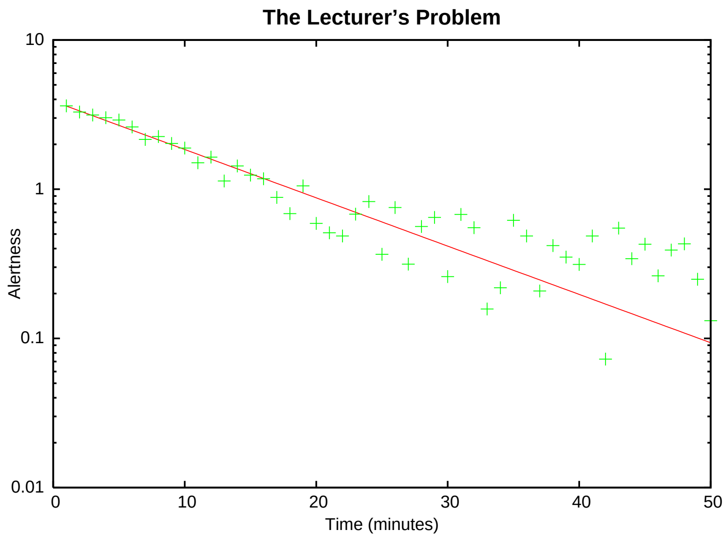
In order to keep the document as plain text, rather than binary, the bitmap is usually encoded. Level 1 PostScript uses hex encoding, where each byte becomes two characters. Level 2 also offers ASCII85, where four bytes are encoded in five characters.

Of course one should also compress the data, and level 2 PostScript offers RLE, LZW and DCT. It also supports RGB, CYMK and paletted images, whereas level 1 barely supports colour at all. Level 3 adds zlib compression.

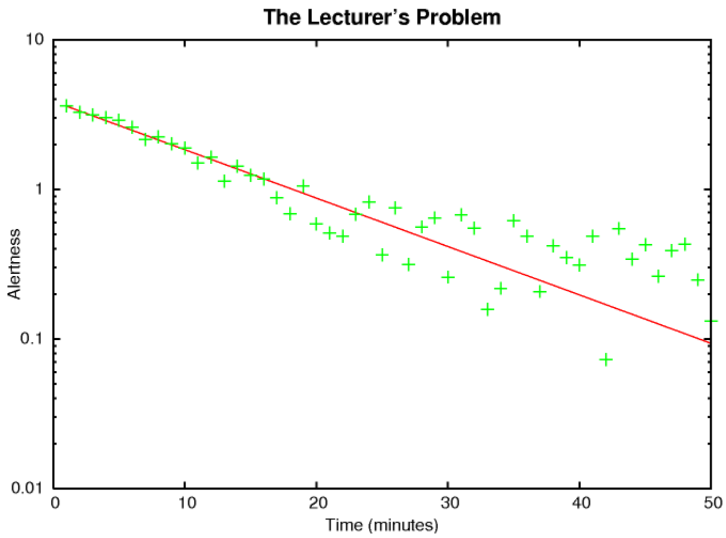
If you are faced with a single page PostScript file several MB in size, look at it in an editor. You may find a long section reading ‘0000000’ or ‘ffffffff’. If so, you have an uncompressed hex encoded bitmap. Congratulations. It can probably be shrunk losslessly by a factor of ten or so.

An EPS file should be asymptotically just 25% larger than the corresponding GIF or JPEG, and can be smaller than the corresponding GIF.

EPS vs Bitmap in Real Life



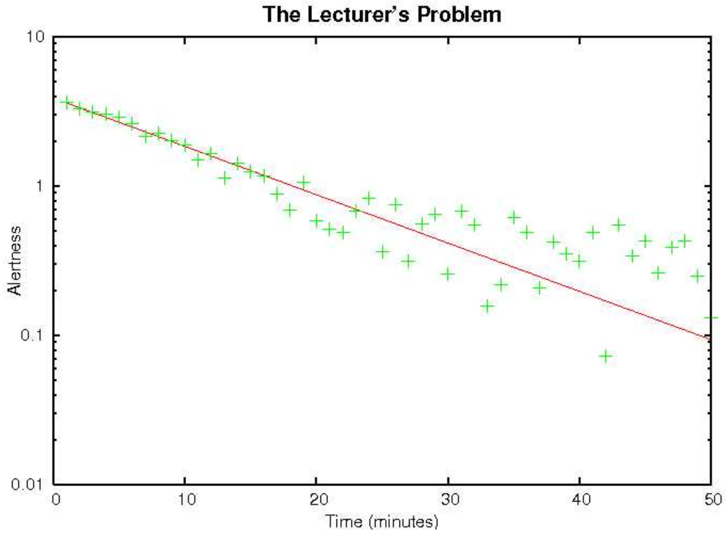
The original EPS figure. 17K fresh from Gnuplot, or $7\frac{1}{2}$K after hand editing.



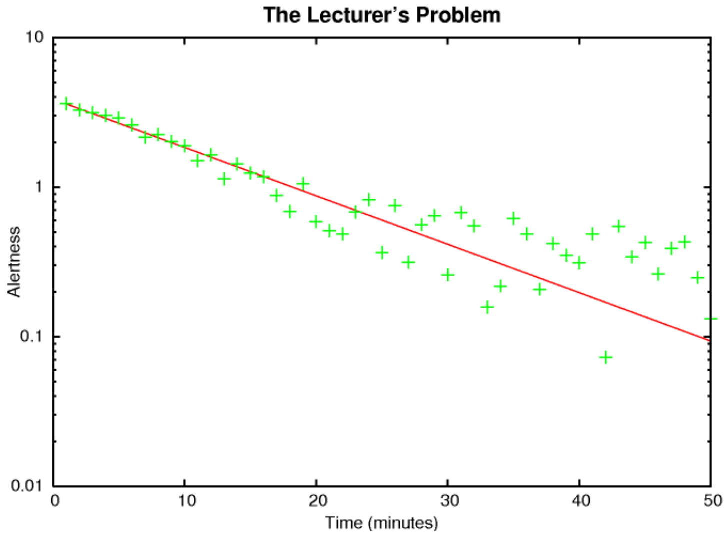
The same converted to a PNG file, width 720 pixels. Still $7\frac{1}{2}$K when converted back to EPS, but the fonts look less good, even when antialiased as here.

If a graph has many points, a bitmap can be more compact than EPS, although the fonts never look quite as good.

Bitmaps losing



The same converted to a JPEG file, width 720 pixels. Now 20K when converted back to EPS, but everything has some strange fuzz around it: don't use DCT on line art!



The same scanned in and converted to a PNG file, width 720 pixels. Now 90K when converted back to EPS, and poorer quality than the direct conversion to PNG.

Scanning is always a poor solution. There appears to be ten times as much 'information' in the random noise introduced by the scanner as in the original figure.

PDF

Portable Document Format.

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 pdfLaTeX and `dvipdfm` rely on!

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

There exist currently (2006) seven different versions of PDF (1.0 to 1.6). Versions later than 1.3 cause compatibility problems with older software, yet offer few useful new features.

Acroread version $X + 1$ is needed to read PDF version $1.X$.

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 1024×768, and it is unlikely that one wants an image of more than 500,000 pixels. With suitable compression, this is unlikely to end up as more than 100KB.

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.

This whole document is around 1.2MB, of which half comes from the pictures of the bench on page 24.

Moving Images

The above arguments all hold for moving images, but with some additions. As consecutive frames are similar, compression algorithms should take account of this. And as data sizes may be huge (about 25 images per second), the compression had better be excellent. For this reason, most algorithms are lossy, often noticably lossy. And, of course, some will want sound as well.

The Motion Picture Experts' Group has produced three well-known collections of algorithms. The first, MPEG-1 (1993), survives in its third (and best) audio compression algorithm: MPEG-1 audio layer 3, or MP3 to its friends.

The second, MPEG-2 (1995), is what you will find on most DVDs. Unfortunately it is covered by patents. According to *Wikipedia*, 640 of them, held by 21 different entities. This makes it awkward for legal free software.

The third, MPEG-4, includes H-264, which is what most digital TV broadcasts use. MPEG-4 also includes some lossless codecs too, as well as everything in MPEG-2.

Later standards are able to assume more processor power for decoding, and draw on better research, so tend to be significantly superior.

The good, bad and ugly

I dislike mentioning specific software as anything said will be out of date before it gets out of the printer. But:

xv: Views and converts between most bitmap formats. Writes GIFs with 256 colour palette always, thus small GIFs are inflated. Unable to compress PostScript decently, as uses RLE only, and uses an 8-bit RLE algorithm even on 24-bit data.

Image Magick: (a.k.a. 'display' and 'convert') Jack of all trades, master of few. See above for bad dithering after colour reduction. Also does bad conversion of JPEG to level 2 PostScript with DCT: it uncompresses the JPEG, then recompresses with its own quality factor. `convert` has the advantage of being a command-line utility, `display` is a GUI.

GIMP: An enormous do-anything GUI thing. Quite good, on a big, fast machine.

pnmutils: A set of command-line utils for converting anything to pnm, pnm to anything, and many transforms on pnm. Sometimes good.

jpegtran: A command-line JPEG mangler. Also `djpeg` and `cjpeg`. Written by the people who defined JPEG.

eps2gif: An EPS to bmp/gif/pnm/jpeg converter. Written by MJR, so bug reports to him, please. (Mainly a front-end to ghostscript.)

bmp2eps: A bmp/gif/pnm/jpeg to EPS level 2 and PDF converter. Written by MJR, so bug reports to him.

eps2ps: That MJR again.

gv, xpdf, acroread: Viewers for (Encapsulated) PostScript, PDF and PDF respectively.

xfig: A vector drawing package.