Image format conversions

R (Chandra) Chandrasekhar

2021-03-07 | 2021-04-18

Although digital images are ubiquitous, one image format does not suit all applications. Printed paper, electronic displays, images on the Web, etc., each call for the same image in a different format. In this tutorial, we explore the different format conversion tools that are currently available. The `ImageMagick` suite, the `cairo` backend, the `poppler` utilities, the `Inkscape` vector graphics editor, and `CairoSVG` are each identified for their individual strengths, that make them the tools of choice for specific image conversion tasks.

Two varieties of digital images

Digital images come in two broad flavours:

- raster or bitmap graphics, and
- vector graphics.

The former leads to image blockiness or pixellation and loss of definition at high magnifications, as shown in Figure 1, while the latter scales without degradation when magnified, as illustrated in Figure 2.

Raster Graphics

There are dozens of image formats, including these three major ones:

1. Tag(ged) Image File Format (TIFF)
    - lossless compression
    - large file sizes
    - used in printing and professional graphics
    - preferred for archival of scanned photographs
2. Joint Photographic Experts Group (JPEG) format
    - small file sizes
    - lossy compression
    - good quality with fast downloads
    - supported by web browsers

Figure 1: Raster graphics image of the letter O (150 dots per inch (dpi) PNG format).

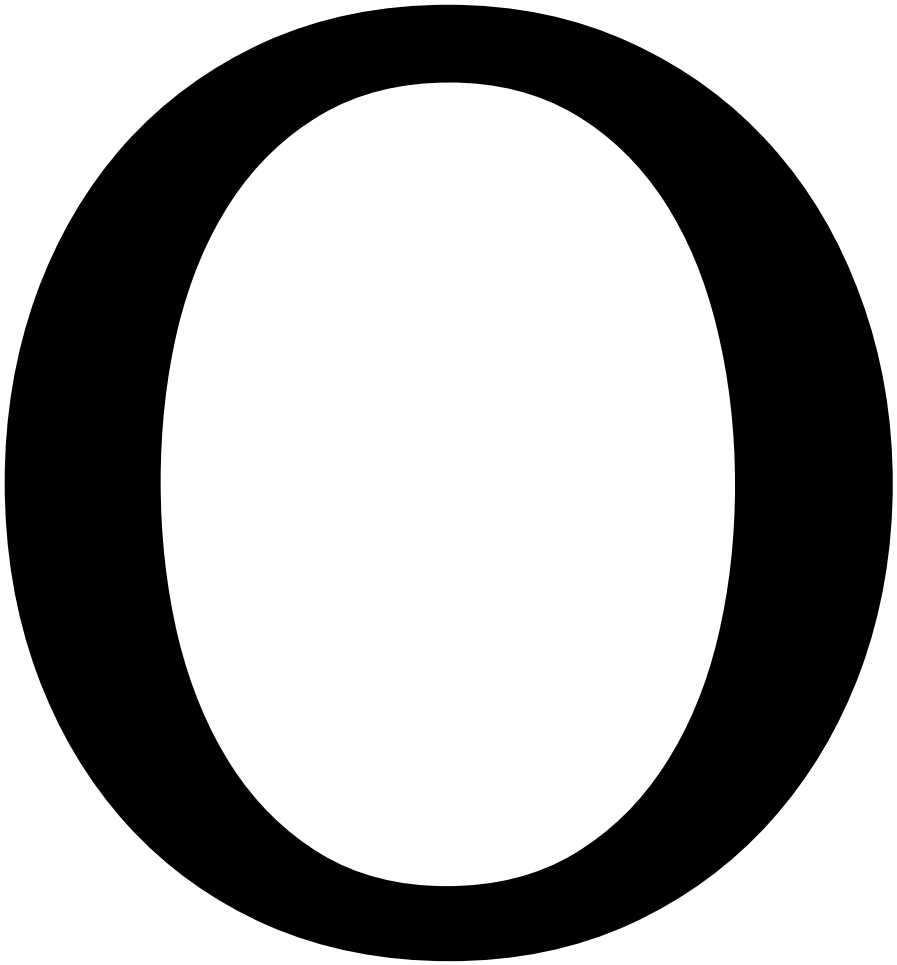

Figure 2: Vector graphics image of the letter O (SVG format).

- preferred for scenes and portraits
- no transparency

3. Portable Network Graphics (PNG) format
   - lossless compression
   - preferred for text and high definition images
   - supported by most web browsers
   - transparency

All three formats yield images displayed as rectangular arrays of *pixels*.

From pictures to pixels

Imagine a picture on a square canvas being converted into a jig-saw puzzle, only this time with square-shaped rather than irregular pieces. If the picture were divided into a hundred pieces in one direction and another hundred pieces in an orthogonal direction, we would have 100^2 = 10,000 pieces in our jig-saw puzzle. Each piece is a *picture element* or *pixel* contributing to the overall picture.

Digital pictures are much the same, except for one difference. Whereas each jig-saw puzzle piece carries a small, detailed part of the big picture, each pixel consists of only *one* colour. *Pixels are coloured tiles in a picture mosaic. Each tile contributes to the overall picture, but it carries not a small picture, but a single colour*.

Clearly, the more tiles we have, the more detailed the emergent picture. Conversely, if the entire picture were composed of a single tile, we would see not a picture but a square of a single colour. Information has been lost and we would have no idea what the picture was about, save for that one dominant colour.

This possibility of information loss on converting an analog picture into digital form is why, when we scan an image, we should take care to digitize the picture with sufficient detail so that it represents the original with fidelity.

File size, quality, and compression

All raster images are rectangular arrays of pixels. How many pixels in the *width* and *height* directions depends on how the original analog image was scanned. Suppose the original image was captured by a mobile phone, and thus already in digital form. If the phone boasts of a 16 million pixel camera, we have a square image of side $\sqrt{16 \times 10^6} = 4000$ pixels on each side.

While such an image would be incredibly detailed, it is inconvenient—because of its large file size—for transmission on social media or for display on the Web. The picture would have to be *resized* to smaller dimensions to yield a more manageable image file size that still represents the original picture with fidelity. This tradeoff between file size and visual quality or definition is at the heart of digital image manipulation and format conversion.

It should be clear by now that we cannot tack a specific number of pixels onto a digital image and say that is its one and only size. Rather the number of pixels representing the width and height of a picture are *one digital representation* of the original analog image. We may *derive* another equally meaningful digital representation by judiciously resizing the image to have a different number of pixels in the width and height directions, while at the same time yielding a more tractable file size, without loss of visual quality.

Some image file formats are amenable to compression, which further reduces file size. Certain formats are capable only of lossless compression, leading to files that are slow to load, but with no loss of information. Others allow lossy compression, where file size reduction results in faster loading, but accompanied by loss of information. Why a specific file format is used, and with what parameters, is determined by image content, purpose, and application domain.

Pixel densities and such

Raster images have a size denoted by the number of pixels comprising the *width* and *height* of the image. The actual *dimensions* of the displayed image, however, depend on the resolution of the display device.

Such images appear better defined at higher resolutions or pixel densities. The units of resolution, or density, commonly used are *dots per inch* (dpi)—in the context of printers—or *pixels per inch* (ppi)—in the context of displays—both of which reference the number of dots or pixels that may be accommodated in one linear inch. It is possible to specify these in dots per centimetre, or pixels per centimetre, but that usage has not caught on.

Commonly used resolutions for output devices are:

- 72 ppi for low resolution monitors
- 96 ppi for standard resolution monitors
- 150 dpi/ppi,[1] which is often the default value in image conversion programs
- 300 dpi/ppi for some mobile phones and laser printers
- 600 dpi/ppi for higher end mobile phones and laser printers

Suppose we have an image that is a square of side 100 pixels. Its width and height are each 100 pixels, and the image consists of a square array of 100^2 or 10,000 pixels. On a display that has a resolution of 96 dpi, such an image will take up $^{100}/_{96}$ = 1.042 inches on each side. If the same image were displayed on a 300 dpi output device, its image will span $^{100}/_{300}$ = 0.333 inches on each side. Image sizes in pixels are therefore quite different from displayed image dimensions in inches.

> A pixel has no size, no physical value or meaning outside of its mathematical representation.
>
> ...
>
> A pixel in itself has no size or physical representation, it can only carry value through its relationship with the screen physical size, creating the resolution, or PPI. Understand this and screen density will have no secret for you. [1]

[1] We will use dpi and ppi interchangeably hereafter to avoid inelegant expressions.

File size and quality again

If an image were resized to *twice* its dimensions, the number of pixels in it will become *fourfold* as will, roughly, its file size.

When photographic images are scanned, higher pixel densities lead to much larger file sizes, resulting in images of remarkable detail. Conversely, low resolution scans sacrifice image quality for small file size. Good visual quality at reasonable file size is the much sought-after optimal goal for any image.

Blocky images arise when the zoomed image is not matched to the display resolution, allowing individual pixels to show themselves as discernible "blocky" elements, as in Figure 1.

The dpi/ppi used for format conversion, especially between raster and PDF, is critical to avoid getting "blurry" or "grungy-looking" images. Merely ratcheting up the source dpi/ppi before conversion from raster to PDF will result in bloated PDF files that convey no discernible improvement in visual quality. There is always a sweet spot for pixel density—in image scanning and format conversions—that gives good visual quality at a decent file size. We should aim for that.

Vector Graphics

The two principal vector graphics formats are:

1. Portable Document Format (PDF)
    - preferred for archival quality electronic and printed documents
    - supported by browsers with integrated PDF readers
    - file sizes comparable to raster images
    - machine-readable files
2. Scalable Vector Graphics (SVG) format
    - preferred for scalable graphics on web browsers
    - used in digital image animations and digital art
    - small file sizes
    - human- and machine-readable files

Both these formats yield images which consist of mathematically defined points, lines, curves, and shapes, which do not degrade in visual quality when magnified.

Page size and viewBox

A raster image has a *width* and a *height* that are directly related to the number of pixels in those two orthogonal directions. Vector images, on the other hand, do not embody an intrinsic resolution, and may therefore be zoomed in or out arbitrarily without loss of detail. But vector images *do have* intrinsic sizes that endow them with a fixed aspect ratio, no less than raster images.

The PDF format was developed for documents meant to be printed on paper. It therefore embodies an intrinsic *page size* in points, where one point is $^{1}/_{72}$ of an inch. Points are absolute linear dimensions which do not change with magnification. In the PDF format, each graphic unit is paginated to the page size parameter in points defined in the file.

With an image format that is designed primarily for display on the Web via a Web browser—as is the case with SVG—image size is defined in terms of a *viewBox*. The latter are again measured in points, and have a fixed aspect ratio, determined by the image. Although such an SVG image may be zoomed without loss of visual quality when displayed on the Web, it too has intrinsic dimensions in terms of its viewBox.

The fact that raster images use pixels, which are relative to an image array, and vector images use points, which are an absolute linear measure, must be kept in mind during format conversion, to avoid perplexing results after conversion.

Format conversions

For many reasons, it is often necessary to convert from one image format to another. There are four broad possibilities for this, as shown below, with typical examples:

1. Raster to raster: PNG to JPEG and vice versa;
2. Raster to vector: PNG to PDF or PNG to SVG;
3. Vector to raster: PDF to PNG or SVG to PNG; and
4. Vector to vector: PDF to SVG, or vice versa.

We consider each of these in turn using platform-neutral open source tools. Since I run GNU/Linux on my desktop, my examples will feature commands from that setup.[2]

Tools for image format conversion

Among the very many tools available, we examine below four that support image format conversion:

1. `ImageMagick`
    - versatile graphics library for image manipulation and display
    - standalone utilities like `convert`, `display`, `identify`, `mogrify`, etc.
    - scripting language support
    - pixel-based
    - raster to raster conversions
    - raster to vector conversions
2. `cairo`
    - vector-based 2D drawing and rendering library
    - multiple output devices/formats
    - used by other programs as a backend, rather than in standalone mode
3. `poppler`
    - vector-based PDF rendering library
    - used by several PDF viewers

[2] There are many websites that promise conversion online, requiring you to upload the input file and download the output file. These *might be* fraught with security risks. Use them with caution.

- uses cairo as backend
- standalone utilities like `pdftotext`, `pdftocairo`, and `pdftoppm`

4. `Inkscape`

- GUI-based vector graphics editor
- suitable both for technical illustration and digital art
- uses SVG as the working format
- can export to a wide variety of output formats
- option to use cairo for export to raster formats

`ImageMagick`: the Swiss Army knife

`ImageMagick` is the name given to a suite of image processing tools originally created in 1987 by John Cristy, then working for Du Pont. In 1990, it was freely released by Du Pont, who transferred copyright to ImageMagick Studio LLC who now maintain the project. It is distributed under a derived Apache 2.0 license. The authoritative source code repository shows active development even today, 34 years after the suite was first released [2].

`ImageMagick` is so versatile and useful that it may rightfully be called the Swiss Army knife of the image processing world. It comes with several command line utilities, each replete with options. Among these are:

- `convert` which converts from one format to another;
- `display` which displays one or more images;
- `identify` which identifies the type of image and displays its characteristics;
- `mogrify` which transforms an image, modifying its appearance; and
- `montage` which generates an image montage from several images.

The above list is far from exhaustive. The interested reader is referred to the excellent online documentation for further details. The power of `ImageMagick` is enhanced with the magick-script Image Scripting Language. The examples in this blog use the command line versions of invoking `ImageMagick`. If they seem daunting, refer to this explanation [3].

Test images

Two quite different images are used to illustrate the format conversions we perform here. The two test images are:

1. a coloured, text-only test image contained in the file `text-only.pdf`; and
2. a coloured, non-text, graphically rich image contained in the file `animals.jpg`.

We will succinctly refer to these two images as `text-only` and `animals`, respectively hereafter.

The `text-only` image

The text-only image was first generated programmatically as a PDF file, `text-only.pdf`, by compiling a LaTeX source file. PDF is the *native* or natural format for this image. The `text-only` image converted into any other format must be measured against the benchmark of the original PDF in file size and visual quality.

PDFs may be displayed on *separate* browser tabs, but cannot be displayed, among other content, *within* a web page. Click here to see `text-only.pdf` as a zoomable PDF on a separate browser tab.

Converting `text-only` from PDF to PNG and JPEG To display `text-only` on this page, the original PDF file was converted to the PNG and JPEG formats using the methods discussed later to yield the raster images `text-only-600-dpi-cairo.png` and `text-only-600-dpi-cairo.jpg`. The commands we used are shown below for completeness, but explained later:

```
# PDF to PNG at 600 dpi
pdftocairo -png -r 600 -singlefile \
text-only.pdf text-only-600-dpi-cairo

# PDF to lossless JPEG at 600 dpi
pdftocairo -jpeg -jpegopt quality=100 -r 600 -singlefile \
text-only.pdf text-only-600-dpi-cairo
```

Figure 3: Text-only image in 600 dpi PNG format.

Figure 4: Text-only image in 600 dpi lossless JPEG format.

```
ls -Xsh text-only.pdf text-only-600-dpi-cairo.* | \
awk '{print $1 "\t" $2}'
---
```



```
120K    text-only-600-dpi-cairo.jpg
16K     text-only.pdf
40K     text-only-600-dpi-cairo.png
```

File sizes As a convention hereafter, when there is a `---` separator between a command and some results, the latter are the results displayed on execution of the command.

The file sizes are displayed above merely for information. Note that the JPEG file is an order of magnitude larger than both the original PDF and the PNG. The relative strengths and weaknesses of different file formats for displaying different image types are discussed later.

The `animals` image

The non-text `animals.jpg` image is a cropped version of the original image `animals-original.jpg` downloaded from the Web. It is a colourful, graphically rich image with much detail, and is from a hand-drawn illustration of microscopic marine animals by the German naturalist Ernst Haeckel, scanned as a JPEG, and made available in the public domain.

Note that `animals-original.jpg` has been scanned from a printed, hard copy illustration and saved as a JPEG raster file. That is its native format. All conversions should be gauged against the file size and visual quality of the cropped original `animals.jpg`.

How the original image was cropped to get the `animals` image is explained next.

Pre-processing `animals` Cropping is strictly not image format conversion, but is often a necessary pre-processing step in image manipulations. For example, Figure 5 has a whitish, non-monochromatic border around the block print, containing annotations. For our purposes, this border is at best a distraction. It may be removed altogether by *cropping*, leaving us with only the illustration. The resulting cropped image, `animals.jpg` will be the source image in our examples below.

Cropping Cropping is usually better done interactively using a GUI (Graphical User Interface), than on the command line. However, the latter, even if a bit tedious, is precisely repeatable.

The `display` utility of `ImageMagick` pops up a GUI, shown in Figure 6, when the mouse is over the image and the left mouse button is clicked. We can then drag and fit a window to the *region we wish to keep*, clicking the `Crop` function, and saving the cropped image. The steps are these:

a. left mouse click on the image to reveal the GUI;
b. `Transform -> Crop`;
c. put the mouse over the top left corner and drag until the bottom right corner to enclose the region of interest;
d. Click again on `Crop`; and
e. `File -> Save` with a different name.

[3] These images are in the public domain and covered by the CC0 licence.

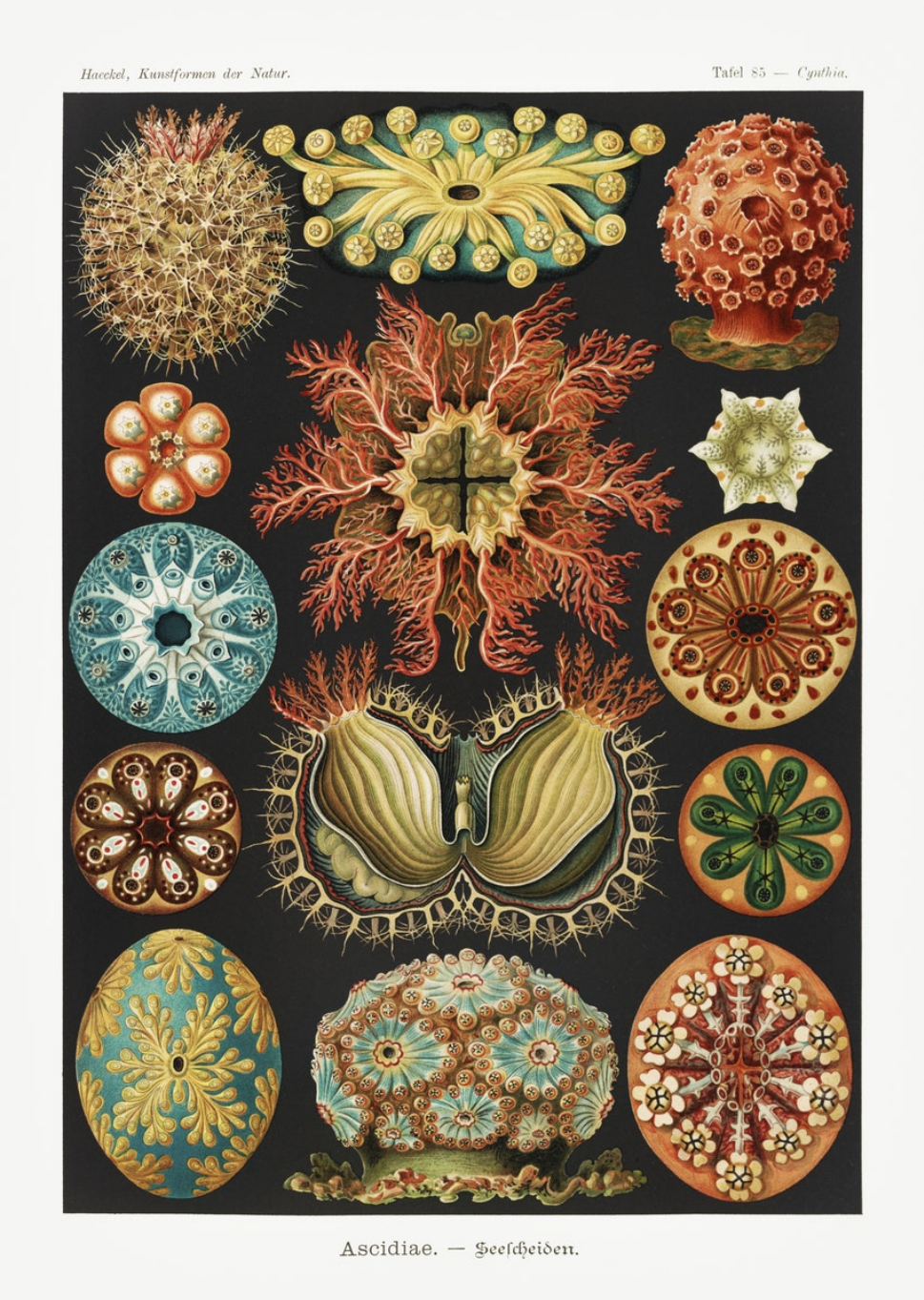

Figure 5: Non-text, graphically rich `animals-original.jpg` image in JPEG format.[3]

Figure 6: `ImageMagick` interactive GUI.

Alternatively, we may just position the cursor on the top left and bottom right corners of the region we wish to *retain*, noting the co-ordinates in each case. If these coordinates are (x_t, y_t) and (x_b, y_b), respectively, we have $w = x_b - x_t$ and $h = y_b - y_t$. We may then invoke the convert command with crop as the option so:

```
convert -crop 'wxh+x_t+y_t' animals-original.jpg animals.jpg
```

In our case, $(x_t, y_t) = (60, 84)$ and $(x_b, y_b) = (795, 1119)$ giving $w = 735$ and $h = 1035$, leading to

```
convert -crop '735x1035+60+84' animals-original.jpg animals.jpg
```

The resulting cropped image, `animals.jpg` is shown in Figure 7 below.

File sizes The sizes of the original and cropped files are shown below in human friendly numbers:

```
ls -Xsh animals*.jpg | awk '{print $1 "\t" $2}'
---
200K    animals.jpg
312K    animals-original.jpg
```

As expected, the original file `animals-original.jpg` is larger than the cropped full-size version, `animals.jpg`, and all is well.

Raster to raster conversion

We now perform a sequence of image manipulations, including raster to raster format conversions.

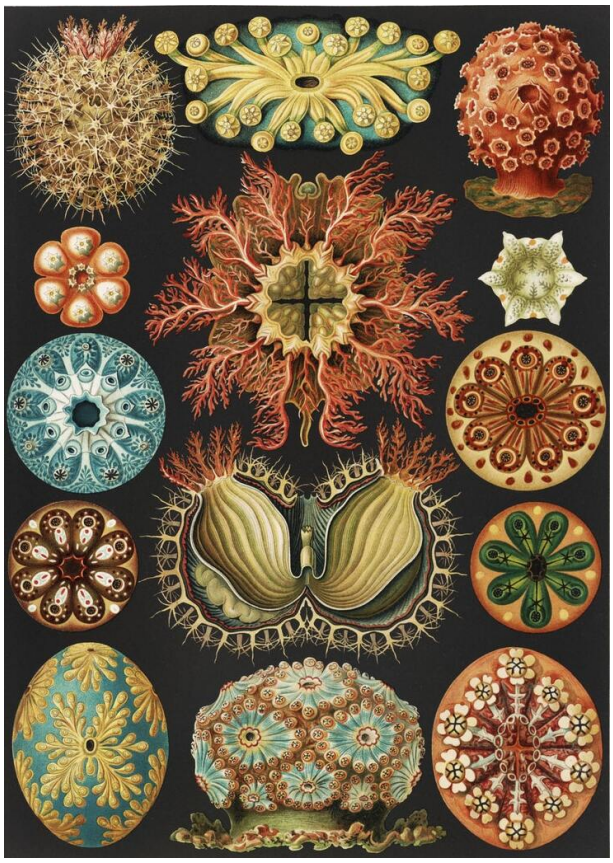

Figure 7: Cropped version of the original image in Figure 5. This is the `animals.jpg` image.

Resizing, format-conversion, and montaging

We may invoke the `convert` function of `ImageMagick` not only to convert from one format to another but also to accomplish cropping (as we have already seen), image-resizing, making the background transparent, and montaging, etc.

Suppose we want to reduce the dimensions of the cropped image to half their original values, and display the full-size and half-size images side by side, we could run the following command:

```
convert animals.jpg -resize 50% animals-halfsize.jpg

# Composite the two images by aligning their bottoms
convert +append -gravity south \
animals.jpg animals-halfsize.jpg animals-both.jpg
```

Background transparency

Notice that there is a coloured white rectangle atop the half-size image on the right in Figure 8. We could remove it by rendering the background transparent. However, because JPEG does not support transparency (through an alpha channel) we have to convert the composite image to the PNG format, which does support transparency. This is an example of why we need to convert from one format to another.

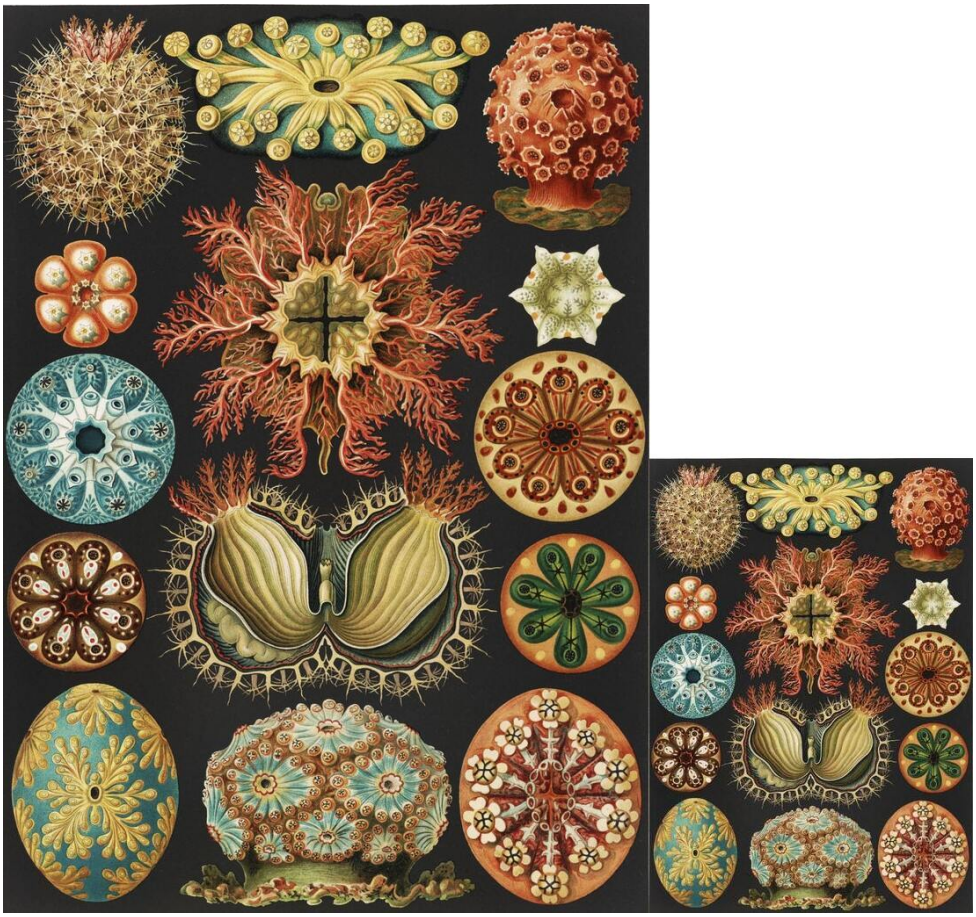

Figure 8: Full-size cropped image on the left and half-sized image on the right in JPEG format.

```
# Non-transparent composite in PNG
convert +append -gravity south \
animals.jpg animals-halfsize.jpg animals-both-non-transparent.png

# Transparent composite in PNG
convert +append -gravity south -background transparent \
animals.jpg animals-halfsize.jpg animals-both-transparent.png
```

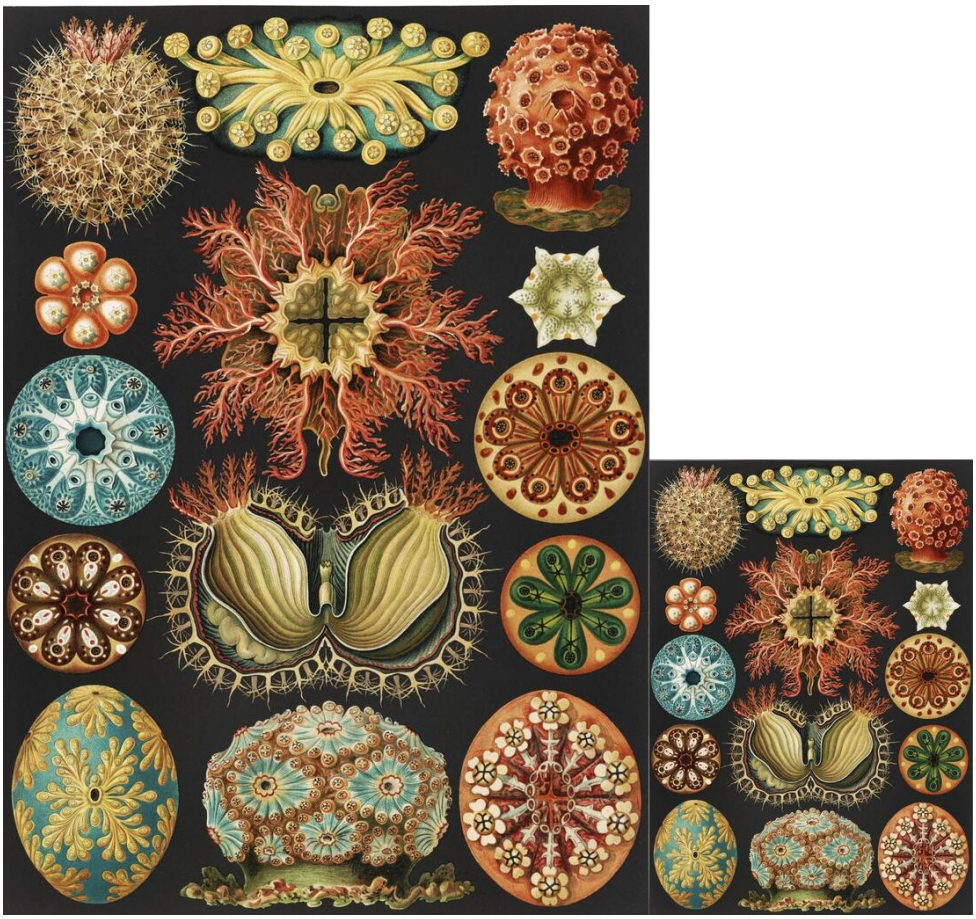

Figure 9: Composite image converted to PNG format with transparent background.

File sizes again How do the file sizes of the three composite images compare? How does the non-transparent JPEG compare with the non-transparent PNG? Also, how high a price have we paid for the transparent background?

```
ls -Xsh animals-both*| awk '{print $1 "\t" $2}'
---
264K    animals-both.jpg
2.0M    animals-both-non-transparent.png
2.2M    animals-both-transparent.png
```

The PNG composite image is *more than seven times larger* than its JPEG counterpart, *even without transparency*. And transparency makes the PNG file size ten percent larger.

Compression levels and file sizes The image compression level used above is the default compression level in `ImageMagick`. Getting the right combination of image format, image dimensions, image compression, and image quality so that the image loads fast and looks good is quite an art. [4]

To get an idea of the range of file sizes involved, let us try generating a composite image with extremes of the compression level, which can range from 0 to 9.

```
# Least compression
convert -define PNG:compression-level=0 +append -gravity south \
-background transparent \
animals.jpg animals-halfsize.jpg \
animals-both-compressed-0.png

# Most compression
convert -define PNG:compression-level=9 +append -gravity south \
-background transparent \
animals.jpg animals-halfsize.jpg \
animals-both-compressed-9.png
```

The file sizes are:

```
ls -Xsh animals-both* | awk '{print $1 "\t" $2}'
---
264K    animals-both.jpg
4.4M    animals-both-compressed-0.png
2.2M    animals-both-compressed-9.png
2.0M    animals-both-non-transparent.png
2.2M    animals-both-transparent.png
```

It appears that the default compression used by `ImageMagick` gives a file size that is the same as the highest compression level. Indeed, the uncompressed version—with a compression level of zero—gives a file *twice* the size of the uncompressed version and *sixteen times* the size of the JPEG. And we have not even used two other related attributes: filter and strategy [5]. Getting the best tradeoff of image format, image size, file size, loading time, and image quality is still more of an art to be mastered than an algorithm to be applied.

We may conclude from the above that non-textual, detail-rich images are better stored and displayed as JPEGs than PNGs.

Results with `text-only`

Recall that `text-only` was originally generated as a PDF. Previously, we briefly touched upon how we converted `text-only` from PDF to PNG and JPEG.

To get a PNG version of the image, we need to run a little ahead of ourselves and convert from PDF to PNG.

From that PNG, let us do a simple *no quality loss* conversion from PNG to JPEG for `text-only`, and compare appearances and file sizes.

```
# Lossless JPEG with a 'quality' of 100 from PNG
convert -quality 100 text-only-600-dpi-cairo.png \
text-only-600-dpi-cairo-IM.jpg

# Composite both images into one PNG with a transparent divider
convert text-only-600-dpi-cairo.png text-only-600-dpi-cairo-IM.jpg \
-background transparent -splice 20x0+0+0 +append -chop 20x0+0+0 \
text-only-both-600-dpi-cairo.png

# Composite both images into one JPEG with a transparent divider
convert text-only-600-dpi-cairo.png text-only-600-dpi-cairo-IM.jpg \
-background transparent -splice 20x0+0+0 +append -chop 20x0+0+0 \
text-only-both-600-dpi-cairo.jpg

ls -Xsh text-only-600-dpi-cairo.png \
text-only-600-dpi-cairo-IM.jpg \
text-only-both-600-dpi-cairo.* \
| awk '{print $1 "\t" $2}'
---
148K    text-only-600-dpi-cairo-IM.jpg
148K    text-only-both-600-dpi-cairo.jpg
40K     text-only-600-dpi-cairo.png
120K    text-only-both-600-dpi-cairo.png
```

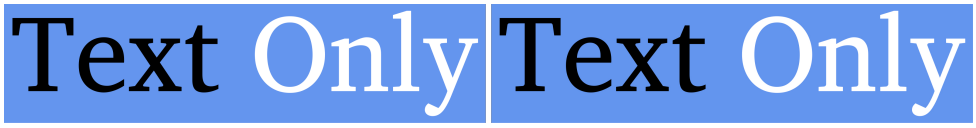

Figure 10: Composite PNG image of the PNG on the left, and JPEG on the right.

The right sub-image of Figure 10 does not reveal noticeable degradation in quality after conversion from PNG to JPEG, and back to PNG again. Note also that the file size of the *composite* PNG image is smaller than the file size of the *single* JPEG image.

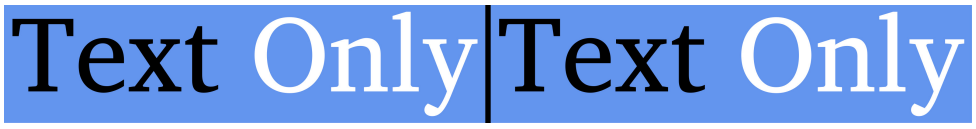

Figure 11: Composite JPEG image of the PNG on the left, and JPEG on the right.

Likewise, notice that when the composite image, Figure 11 is a JPEG, there is no noticeable degradation in image quality. Moreover, the file size of the composite is the same as for the singleton JPEG image. Note also that because JPEG does not support transparency, the sliver of "transparent background" sandwiched between the two images now appears black.

PNG for text and JPEG for non-text

We conclude from the `text-only` images that PNG is better suited for textual images and provides a smaller file size for the same quality.

Conversely, we know from the `animals` images that JPEG is more suited to non-textual detail and yields good quality images at far smaller file sizes than PNG.

Can `cairo` and `poppler` do all this?

It is all a question of what format do we start with?

Both `cairo` and `poppler` are designed for PDF input images. They stand out as tools of choice when we start off with PDFs.

The `animals` image, on the other hand, is scanned from an illustration. Our input is a JPEG raster image. The forte of `ImageMagick` is the display, manipulation, format conversion, and processing of raster images. So, we run with the utilities provided by `ImageMagick` in the latter case.

Raster to vector conversions

Let us say that we have a logo, designed and available as a raster image in some format. To use it on the Web, we could, if necessary, reformat it as a JPEG or PNG file. But as we zoom into the page, the raster images will start becoming less sharp and more blocky as shown in Figure 1.

However, if the graphic were in SVG format, supported by most web browsers, the logo would scale without visual degradation as we zoom into the page.

How do we convert a raster image to a vector format like PDF or SVG?

Raster to PDF with `convert` for `animals`

The `convert` utility of `ImageMagick` comes to our rescue again. For example,

```
# Convert JPEG image to PDF
convert animals.jpg animals-IM.pdf

ls -Xsh animals.jpg animals-IM.pdf | awk '{print $1 "\t" $2}'
---
200K    animals.jpg
204K    animals-IM.pdf
```

Web browsers, while they may feature PDF viewers on separate tabs, are still unable to display PDFs as part of a web page. The converted image, `animals-IM.pdf`, may be viewed on a browser tab from the given link. If the converted PDF is magnified by zooming, it will be seen to reveal remarkable detail. And the difference between the JPEG and PDF file sizes is negligible.

What happens, though, if the half-sized image is used to generate the PDF? It is smaller and accordingly embodies less information than the original, again commensurate with the respective file sizes.


```
convert animals-halfsize.jpg animals-halfsize.pdf

ls -Xsh animals-halfsize.jpg animals-halfsize.pdf | \
awk '{print $1 "\t" $2}'
---
64K     animals-halfsize.jpg
64K     animals-halfsize.pdf
```

Increasing detail demands larger file sizes: there is no free lunch. But the conversion from PDF to JPEG does not cost us much, if at all, in file size.

Raster to SVG with `convert`

Will `convert` cater for a JPEG to SVG conversion?

```
# Original `animals.jpg` to SVG using `convert`
convert animals.jpg animals-IM.svg

ls -Xsh animals.jpg animals-IM.svg | awk '{print $1 "\t" $2}'
---
200K    animals.jpg
436K    animals-IM.svg
```

The SVG file is more than *twice* the size of the original JPEG. The question arises whether there is an alternative route to the SVG that could give us smaller file sizes but comparable fidelity. What if we did not convert from raster to SVG but from raster to PDF and thence to SVG?

Since PDF to SVG conversion is really part of vector to vector conversion, we will revisit this question later.

Round tripping from PDF through PNG to PDF

The `text-only` image first saw life as a PDF. We then converted it to a 600 dpi PNG. What happens if we convert that image back to a PDF?

```
# Generate PDF from 600 dpi PNG using `convert`
convert text-only-600-dpi-cairo.png text-only-from-600-dpi-PNG.pdf

ls -Xsh text-only*600*.pdf text-only.pdf | awk '{print $1 "\t" $2}'
---
40K     text-only-from-600-dpi-PNG.pdf
16K     text-only.pdf
```

Not surprisingly, the round trip has resulted in a fatter file for the PDF the second time around. Compare for yourself `text-only.pdf` and `text-only-from-600-dpi-PNG.pdf` by viewing each on a separate browser tab and zooming.

What would you expect if the initial PDF to PNG image conversion had been done at 150 dpi, or 96 dpi, or 75 dpi? The command sequence is explained later but the results and their consequences are noteworthy here:

```
# Generate 75 dpi PNG from `text-only.pdf` using `pdftocairo`
pdftocairo -png -r 75 -singlefile text-only.pdf text-only-75-dpi

# Round trip convert the 75 dpi PNG back to PDF using `convert`
convert text-only-75-dpi.png text-only-75-dpi.pdf

ls -sh text-only.pdf text-only*75* | awk '{print $1 "\t" $2}'
---
8.0K    text-only-75-dpi.pdf
8.0K    text-only-75-dpi.png
16K     text-only.pdf
```

View the PDF generated from the 75 dpi PNG on a separate browser tab and zoom in on the image as before. How does it compare with the one generated previously from the 600 dpi PNG?

This is one reason why conversion from a PNG to a PDF might result in a PDF which looks like a raster image when zoomed in close. The source image resolution was not high enough to generate a PDF that does not degrade on zooming, *on the monitor being used for display*. *The visual quality of the original raster image is what the output PDF will embody.* Just because a PDF image scales does not mean it cannot exhibit blockiness. It will, if a low resolution raster image was used as source.

Vector to raster

The `poppler` utilities, with the `cairo` backend are the primary resource for vector to raster conversions, specifically when the source image is a PDF.

PDF to PNG and JPEG: `poppler` and `cairo`

It was mentioned here and here that `text-only` was originally generated as a native PDF, vector graphics image, and subsequently converted to the PNG and JPEG formats. We explain how that was done and also why the `ImageMagick` suite is not used for this purpose.

The `poppler` suite contains utilities to convert from PDF to several raster formats. Two versatile utilities called `pdftocairo` and `pdftoppm` are available for our purpose. One may view their usage by typing the name of the utility prefixed by `man` or suffixed by `-help`, although the former is more exhaustive.

To convert from vector to raster, we invoke commands like these:

```
# `pdftocairo`: from PDF to 600 dpi PNG
# root file is the last argument
pdftocairo -png -r 600 -singlefile text-only.pdf \
text-only-600-dpi-cairo
```

```
# `pdftoppm`: from PDF to 600 dpi PNG
# root file is the last argument
pdftoppm -png -r 600 -singlefile text-only.pdf \
text-only-600-dpi-ppm

# `pdftocairo`: PDF to 600 dpi JPEG
# Options may be passed to JPEG
pdftocairo -jpeg -jpegopt "quality=100" -r 600 \
-singlefile text-only.pdf text-only-600-dpi-cairo

# `pdftoppm`: PDF to 600 dpi JPEG
# Options may be passed to JPEG
pdftoppm -jpeg -jpegopt "quality=100" -r 600 \
-singlefile text-only.pdf text-only-600-dpi-ppm

# Using `convert` from ImageMagick for PNG to JPEG
# Source PNG file was output by `pdftocairo`
convert -units pixelsperinch -density 600 -quality 100 \
text-only-600-dpi-cairo.png text-only-600-dpi-cairo-IM.jpg

# Using `convert` from ImageMagick for PNG to JPEG
# Source PNG file was output by `pdftoppm`
convert -units pixelsperinch -density 600 -quality 100 \
text-only-600-dpi-ppm.png text-only-600-dpi-ppm-IM.jpg
```

The value `-r 600` signifies a resolution of 600 pixels per inch (ppi), or alternatively, dots per inch (dpi). The default value is 150 ppi. The value of 600 is suitable for printing on laser printers to give output that will visually rival the original PDF in quality. Note that while raster images have inherent resolutions, PDF images have none: they scale without loss of quality when generated natively.

The `-singlefile` option is used because we are simply converting a single "page" of PDF rather than a numbered page sequence. In all cases, the destination filename is the "root" of the converted file sequence, which in this case is the output filename without any extension.

In addition, the JPEG version may feature lossy compression where quality is traded for file size. Since PNG is lossless, to compare the two formats on an even keel, we specify that the `-quality` of the JPEG should be the maximum of 100.

Both `pdftocairo` and `pdftoppm` are used in the first four conversions above, with appropriately named filenames.

We could also use `convert` from `ImageMagick` to convert from PNG to JPEG, and this is done in the last two commands above. Note that this is strictly not a vector to raster conversion but merely raster to raster. See below for why we cannot convert from PDF to raster with `convert`.

The files sizes that result are shown below:


```
ls -Xsh text-only.pdf text-only-600* | awk '{print $1 "\t" $2}'
---
148K    text-only-600-dpi-cairo-IM.jpg
120K    text-only-600-dpi-cairo.jpg
140K    text-only-600-dpi-ppm-IM.jpg
112K    text-only-600-dpi-ppm.jpg
16K     text-only.pdf
40K     text-only-600-dpi-cairo.png
40K     text-only-600-dpi-ppm.png
```

The numbers tell their own story. I would have expected the two sets of raster images output by `pdftocairo` and `pdftoppm` to be roughly equal in size, given their identical options during invocation. Strangely, they are not, at least for the JPEGs. This could be either because of different defaults, or different algorithms, or something else: I simply do not know.

It appears that `pdftoppm` gives marginally smaller file sizes for JPEG than `pdftocairo`. Moreover, when `pdftoppm` is used to convert *directly* from PDF to JPEG, the file size is smaller than when PNG is used as an intermediate file format and conversion to JPEG is by `convert` from `ImageMagick`.

One other takeaway is that text-rich images are better rendered in PNG than JPEG, as we have already noted. The PDF and PNG image file sizes are of the same order of magnitude, whereas the JPEGs are an order of magnitude larger.

Why is `ImageMagick` disallowed for PDF to raster?

If you try to `convert` a PDF to any raster image format, you will get an error:

```
# Try to convert from PDF to PNG using `convert`
convert text-only.pdf text-only.png
---
convert: attempt to perform an operation not allowed by the security policy 'gs' @ err
convert: no images defined 'text-only.png' @ error/convert.c/ConvertImageCommand/3304.
```

Such a conversion was commonplace some years ago, but is now disallowed, for reasons explained in the appendix.

SVG to PNG and JPEG

There was a time when SVGs were ill-supported by browsers and therefore, PNGs and JPEGs dominated the logo and icon landscape on the web. Now that most mainstream browsers support SVGs out of the box, there is healthy support for tools that convert to and from SVGs.

Some SVG basics An SVG file is a text file *describing* the points, curves, and shapes as they are rendered on a page. Such a description is unshackled from the rectangular array of pixels that typify a raster image. So, what is the *natural size* of an SVG image? We touched upon this at the very beginning of this blog.

Generating an SVG from a PDF First, we need to generate an SVG version of the file `text-only.pdf`. We are running ahead of ourselves because we need to convert from PDF to SVG. We use `pdftocairo` for this because we cannot use `convert` as explained in the appendix. Moreover, `pdftoppm` does not support SVG as an output format. So, `pdftocairo` is our preferred option.

```
# PDF to SVG using `pdftocairo`
pdftocairo -svg text-only.pdf text-only-pdftocairo.svg

ls -Xsh text-only.pdf text-only-pdftocairo.svg | \
awk '{print $1 "\t" $2}'
---
16K     text-only.pdf
12K     text-only-pdftocairo.svg
```

It is instructive to open the file `text-only-pdftocairo.svg` and look at the first block of text:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" wid
```

The *width*, *height*, and *viewBox* of the SVG image are all stated in *points*, abbreviated as pt where, as we have already seen, 1 pt equals $^{1}/_{72}$ inch [6]. At a screen setting of 96 pixels per inch, 1 pt equals $^{4}/_{3}$ pixels. So, the native size of the `test-only` image as a PNG should be about 286 pixels by 70 pixels on a 96 ppi monitor. When`convert` is applied on the SVG to yield the PNG, the latter is faithful to these unit conversions:

```
# SVG to PNG using `convert` at 96 ppi
convert -units pixelsperinch -density 96 \
text-only-pdftocairo.svg text-only-96-dpi-svg-convert.png

# Display basic information about an image using `identify`
identify text-only-96-dpi-svg-convert.png
---
text-only-96-dpi-svg-convert.png PNG 286x70 286x70+0+0 8-bit sRGB 6548B 0.000u 0:00.000
```

And, not unexpectedly, the numbers returned by `identify` match those we derived above by simple unit conversion. ☺

Regarding file sizes, the PNG is actually smaller than the SVG:

```
ls -Xsh text-only-pdftocairo.svg text-only-96-dpi-svg-convert.png | \
awk '{print $1 "\t" $2}'
---
8.0K    text-only-96-dpi-svg-convert.png
12K     text-only-pdftocairo.svg
```


Figure 12: Fuzzy PNG image from SVG using `convert` at 96 dpi.

The PNG image lacks the sharpness of definition the original exhibited, which is not surprising because an increase in image resolution has not been achieved, and moreover, the file size has been reduced. Note the white border that has been added to three sides of the image, perhaps to arrive at an integer number of pixels for the image dimensions.

Let us repeat the above process but with a higher ppi for the PNG, say 600:

```
# SVG to PNG using `convert` at 600 dpi
convert -units pixelsperinch -density 600 \
text-only-pdftocairo.svg text-only-600-dpi-svg-convert.png

# Display basic information about an image using `identify`
identify text-only-600-dpi-svg-convert.png
---
text-only-600-dpi-svg-convert.png PNG 1786x435 1786x435+0+0 8-bit sRGB 43439B 0.000u 0
---
ls -Xsh text-only-pdftocairo.svg text-only-600-dpi-svg-convert.png | \
awk '{print $1 "\t" $2}'
---
44K     text-only-600-dpi-svg-convert.png
12K     text-only-pdftocairo.svg
```

The PNG image is now 1786 by 435 pixels as expected, and the file size is also modestly larger. The PNG image, displayed on a standard desktop screen or even a mobile phone display should now appear sharper, as shown in Figure 13. But the white border on three sides still shows up for reasons unclear to me. My guess is that it might have to do with getting an integer number of pixels in the PNG output.

Figure 13: Sharper PNG image from SVG using `convert` at 600 dpi.

It is important to know which options to use with `convert` to get the desired results at economical files sizes and acceptable visual quality. `ImageMagick` could use either its builtin SVG renderer MSVG or `rsvg-convrt` from `librsvg`. Knowing the intermediate delegates involved in the conversion chain helps optimize the process.

Font support in SVG is not widespread, and format conversions might result in non-optimal font rendering after conversion.

Conversion tools There is a growing number of tools that can convert an SVG to a PNG image. Among these are:

a. `convert` from `ImageMagick`;

b. `Inkscape`;

c. `cairosvg`;

d. `rsvg-convert` from `librsvg`.

We know that `convert` will produce a minimally sized PNG that faithfully converts pt to pixels at 96 dpi. Other tools might use different default dpi values. Moreover, different programs might insert borders, transparent backgrounds, etc., modifying the aspect ratio of the PNG slightly. We will gloss over such details in the tool comparison below and simply focus on the command line invocations and resulting file sizes. Bear in mind also that some of the tools could use identical backends and therefore give identical output images.

The tool faceoff We will convert `text-only` from SVG to PNGs at 600 dpi, using different tools, and compare results, starting with `inkscape`.

```
inkscape  -d 600 -o text-only-600-dpi-inkscape-svg.png text-only-pdftocairo.svg
---
Background RRGGBBAA: ffffff00
Area 0:0:285.747:69.5467 exported to 1786 x 435 pixels (600 dpi)
---
identify text-only-600-dpi-inkscape-svg.png
---
text-only-600-dpi-inkscape-svg.png PNG 1786x435 1786x435+0+0 8-bit sRGB 43268B 0.000u
---
ls -sh text-only-600-dpi-inkscape-svg.png | awk '{print $1 "\t" $2}'
---
44K     text-only-600-dpi-inkscape-svg.png
```

The explicit conversion from points to pixels at 600 dpi is clear, as is the image itself. We would expect the PNG to have a size of 1786 by 435 pixels, and it does. No surprises there. ☺

CairoSVG is designed to parse well-formed SVG files, and draw them on a Cairo surface. Cairo is then able to export them to PDF, PS, PNG, and even SVG files. If we use `cairosvg`, we get these results:

Figure 14: PNG at 600 dpi of `text-only` from SVG using `inkscape`.

```
cairosvg -d 600 -f png -o text-only-600-dpi-cairosvg.png text-only.svg

identify text-only-600-dpi-cairosvg.png
---
text-only-600-dpi-cairosvg.png PNG 1785x434 1785x434+0+0 8-bit sRGB 42999B 0.000u 0:00
---
ls -sh text-only-600-dpi-cairosvg.png | awk '{print $1 "\t" $2}'
---
44K     text-only-600-dpi-cairosvg.png
```

Figure 15: PNG at 600 dpi of `text-only` from SVG using `cairosvg`.

The image and file sizes are within a whisker of each other for `inkscape` and `cairosvg` and they are comparable in ease of use and fidelity. Indeed, it could very well be that `inkscape` uses the `cairo` backend.

Now for the final tool, `rsvg-convert`.

```
rsvg-convert -a -d 600 -p 600 -f png -o text-only-600-dpi-rsvg-convert.png text-only.sv

identify text-only-600-dpi-rsvg-convert.png
---
text-only-600-dpi-rsvg-convert.png PNG 1786x435 1786x435+0+0 8-bit sRGB 43171B 0.000u
---
ls -sh text-only-600-dpi-rsvg-convert.png | awk '{print $1 "\t" $2}'
---
44K     text-only-600-dpi-rsvg-convert.png
```

Note that the `-a` option preserves aspect ratio, and the x and y resolutions have to be specified separately using `-d` and `-p` respectively. Though more verbose, it also offers options to shear or zoom the image and is more versatile.

All of `inkscape`, `cairosvg`, and `rsvg-convert` produce PNG files of the same size and visual quality, as is apparent from Figures !Figure 14, !Figure 15, and !Figure 16.

Figure 16: PNG at 600 dpi of `text-only` from SVG using `rsvg-convert`.

Non-textual, scanned mages like `animals` are best displayed as JPEGs, given their compact file sizes in that format. There is no obvious need to convert such images from JPEG to SVG and back again, simply to view how the quality and file size changed during the roundtrip. Accordingly, we will not consider the `animals` image here.

Vector to vector

We have now reached the *last* of the four types of format conversion when both input and output files are in vector formats. 😄 😊

There are principally two cases here:

a. PDF to SVG; and
b. SVG to PDF.

The `cairo` and `poppler` libraries and their utilities are our "go to" resource if the source image is PDF. `cairosvg` and `rsvg-convert` are our resources when the source image is an SVG. Also, the `Inkscape` GUI-based vector graphics editor supports SVG as its native format, and allows export of the generated SVG graphics both as PDF and as PNG.

PDF to SVG with the `animals` image

When PDF is the source format, the `poppler` standalone utilities `pdftocairo` and `pdftoppm` are the tools of choice.

We have already generated `animals-IM.pdf`. Let us now convert it SVG and view it.

```
# PDF to SVG using pdftocairo
pdftocairo -svg animals-IM.pdf animals-pdftocairo.svg
```

The images appear the same visually and do not seem to have lost definition in the conversion from the original JPEG through the two vector formats.

Let us collate and view the file sizes of all the `animals` images in the PDF and SVG formats, including those previously converted using `convert` from `ImgeMagick`.

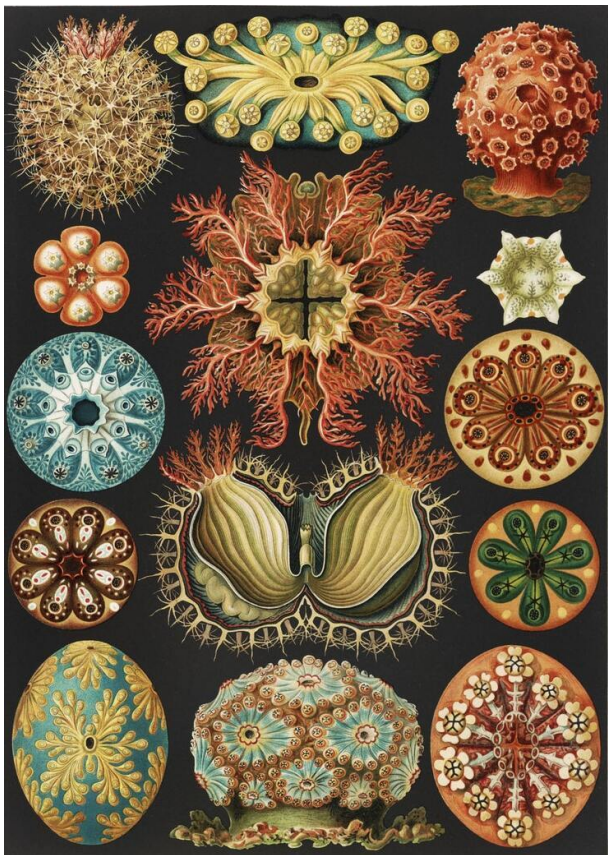

Figure 17: SVG version of the `animals` image.

```
ls -Xsh animals.jpg animals-{I,p}*.pdf animals*.svg | \
awk '{print $1 "\t" $2}'
---
200K    animals.jpg
204K    animals-IM.pdf
436K    animals-IM.svg
268K    animals-pdftocairo.svg
```

We may infer that, for conversion to SVG:

1. Direct conversion from JPEG to PDF, or from PDF to SVG, does not substantially degrade image quality or increase file size for visually rich, non-textual images like `animals`.
2. Direct conversion from JPEG to SVG using `convert` results in a much larger file size than if we converted from JPEG to PDF and used `pdftocairo` to convert that PDF to SVG. Two stage-conversion may therefore be preferable, at least in some cases.
3. `pdftoppm` is not set up to convert *to* SVG, and hence cannot be used.

The standalone `pdf2svg` utility There is a utility called `pdf2svg` that has been available for some time now. It may be used to accomplish the same PDF to SVG conversion as `pdftocairo`:


```
# PDF to SVG using pdf2svg
pdf2svg animals-IM.pdf animals-pdf2svg.svg

ls -Xsh animals-IM.pdf animals-pdf2svg.svg animals-pdftocairo.svg | \
awk '{print $1 "\t" $2}'
---
204K    animals-IM.pdf
268K    animals-pdf2svg.svg
268K    animals-pdftocairo.svg
```

The file sizes are identical to those from the `pdftocairo` conversion. Drew Barton, the author of `pdf2svg`, has written:

> Note: since this utility was written, the maintainers of Poppler have written a utility that works on the same principle: `pdftocairo`. I recommend that you use their utility since it is better maintained than mine.

So, it appears that `pdftocairo` is sufficient for converting from PDF to SVG.

PDF to SVG with the `text-only` image

We have already converted from PDF to SVG using `pdftocairo` previously. We repeat below the identical command used before to generate the SVG image:

```
pdftocairo -svg text-only.pdf text-only-pdftocairo.svg

ls -Xsh text-only.pdf text-only-pdftocairo.svg | awk '{print $1 "\t" $2}'
---
16K     text-only.pdf
12K     text-only-pdftocairo.svg
---
pdfinfo text-only.pdf | grep pts
---
Page size:      214.31 x 52.16 pts
---
less text-only-pdftocairo.svg | grep viewBox
---
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" wid
```

The PDF *page size* and the SVG *viewBox* are identical at 214.31 by 52.16 pts, as we would expect. For comparison, the `text-only.pdf` image may be opened on a separate browser tab and compared with the SVG shown in Figure 18.

PDF to SVG to PDF roundtrip We could open up `text-only.svg` in `Inkscape` and save it as a PDF, `text-only-inkscape.pdf` completing the round trip. Or we could use the command line so:

Figure 18: SVG version of `text-only` image converted from PDF by `pdftocairo`.

```
inkscape text-only.svg -o text-only-inkscape.pdf

ls -Xsh text-only*.{svg,pdf}| awk '{print $1 "\t" $2}'
---
8.0K    text-only-inkscape.pdf
16K     text-only.pdf
12K     text-only.svg
```

What exactly has been gained or lost in this round trip is a little too recondite to consider here, and covers issues such as PDF version, font embedding or its absence, default borders, etc.

SVG to PDF

When SVG is the source image, there are four routes to format conversion:

1. `cairosvg`, which parses well-formed SVG files, draws them on a Cairo surface, and then uses `cairo` to export them to PDF, PS, PNG, and even to SVG again.
2. `rsvg-convert`, which is based on `librsvg` and which uses `libxml` and `cairo` to render and convert SVG files into other formats.
3. `inkscape`, which opens SVG files natively and can export them as a PDF, PNG, etc.
4. `convert`, which can convert an SVG to PDF and also to any raster format like PNG.

SVG to PDF:`text-only`

Let us use the file `text-only-pdftocairo.svg` as the input and convert it to PDF using all four means, and compare the resulting file sizes:

```
cairosvg -f pdf -o text-only-cairosvg.pdf text-only-pdftocairo.svg

rsvg-convert -f pdf -o text-only-rsvg-convert.pdf text-only-pdftocairo.svg

inkscape text-only-pdftocairo.svg -o text-only-inkscape.pdf

convert text-only-pdftocairo.svg text-only-IM.pdf
```



```
ls -Xsh text-only-pdftocairo.svg text-only-{c,r,i,I}*.pdf | \
awk '{print $1 "\t" $2}'
---
8.0K    text-only-cairosvg.pdf
8.0K    text-only-IM.pdf
8.0K    text-only-inkscape.pdf
8.0K    text-only-rsvg-convert.pdf
12K     text-only-pdftocairo.svg
```

The PDF file sizes are the same from the different conversion methods. Since the *viewBox* of the input SVG is the same and the file content is the same, the PDF *Page size* is the same for all PDFs.

Comparison with the original file `text-only.pdf` shows the following differences:

a. The three files `text-only-cairosvg.pdf`, `text-only-inkscape.pdf`, and `text-only-rsvg-convert.pdf` all show a white border on three sides of the converted PDF file. Zooming shows no degradation in image quality.

b. The fourth file, `text-only-IM.pdf` obviously went through an intermediate raster phase and shows blockiness when zoomed. Direct conversion from SVG to PDF vis `ImageMagick convert` should be avoided.

SVG to PDF: `animals`

We use the file `animals-pdftocairo.svg` as source to convert to PDF and list the commands and file sizes below:

```
cairosvg -f pdf -o animals-cairosvg.pdf animals-pdftocairo.svg

rsvg-convert -f pdf -o animals-rsvg-convert.pdf animals-pdftocairo.svg

inkscape animals-pdftocairo.svg -o animals-inkscape.pdf

convert animals-pdftocairo.svg animals-from-svg-IM.pdf

ls -Xsh animals-pdftocairo.svg animals-{c,r,i,f}*.pdf | \
awk '{print $1 "\t" $2}'
---
1.9M    animals-cairosvg.pdf
200K    animals-from-svg-IM.pdf
204K    animals-inkscape.pdf
1.9M    animals-rsvg-convert.pdf
268K    animals-pdftocairo.svg
```

The page sizes are similar but the file sizes differ by an order of magnitude for the different PDFs. Those output by the `cairo` library are almost ten times larger than those put out by `ImageMagick` and `Inkscape`.

You may compare the appearance of all four images by clicking on separate browser tabs for these four images and zooming in to see details:

- animals-cairosvg.pdf
- animals-from-svg-IM.pdf
- animals-inkscape.pdf
- animals-rsvg-convert.pdf

The image animals-from-svg-IM.pdf is pixellated on zooming in and we should avoid this conversion for images with fine detail. The two images weighing in at a file size of 1.9M exhibit detail commensurate with their heft. The image animals-inkscape.pdf is the preferred option as it balances good rendition of detail with a tractable file size.

Summary

1. convert from ImageMagick is the tool of choice for converting from any raster format to another raster format or to PDF or SVG.
2. When we start out with PDF as the source image format, and the destination format is either a raster format or SVG, the tool of choice is pdftocairo from the poppler utilities.
3. When the source format is SVG and the destination format is either PDF or a raster format, the tools of choice are either cairosvg or inkscape depending on image content.

Table 1 summarizes this information, which is current at the time of writing, but could change as the image utilities landscape changes with time.

Table 1: Tools for image format conversions.

Conversion Type	Tool
raster to raster	convert
raster to PDF	convert
raster to SVG	convert
PDF to raster	pdftoppm
SVG to raster	cairosvg
PDF to SVG	pdftocairo
SVG to PDF	inkscape

Appendix: Security vulnerabilities in ImageMagick

Great power exacts a commensurate price. ImageMagick's great power and ease of use does come at a great price: vulnerability to exploits by malicious remote actors.

ImageMagick uses external libraries or *backend tools* which are called via system() commands in accordance with *delegated* command strings specified in a configuration file called policy.xml.

In April 2016, it was reported that because of insufficient validation of delegated command strings, it was possible for someone to execute malicious code remotely, to the detriment of the unwitting user of `ImageMagick`. This was revealed at a website, interestingly named ImageTragick to attract sufficient attention and remedial action to the discovered bug [7].

In November 2020, another security vulnerability was discovered [8]. It was reported and promptly patched by the `ImageMagick` maintainers [9].

Recent versions of the `ImageMagick` suite, bundled with major distributions, should have correctly configured `policy.xml` files that will block known exploits. Sandboxing is another technique to quarantine the system from possible vulnerabilities. Above all, it is vital to keep system and application software up to date to avail of evolutions in performance and security.

Feedback

Please email me your comments and corrections.

A PDF version of this article is available for download here:

https://swanlotus.netlify.app/blogs/image-format-conversions.pdf

References

1. Gabriel, Sebastien. Designer's guide to DPI. Online. 15 May 2015. [Accessed 27 March 2021]. Available from: https://sebastien-gabriel.com/designers-guide-to-dpi/
2. ImageMagick Studio LLC. ImageMagick 7. Online. [Accessed 8 March 2021]. Available from: https://github.com/ImageMagick/ImageMagick
3. ImageMagick—Command-line Processing. Anatomy of the Command-line. Online. [Accessed 12 March 2021]. Available from: https://imagemagick.org/script/command-line-processing.php
4. Newton, Dave. Efficient Image Resizing With ImageMagick—Smashing Magazine. Online. 25 June 2015. [Accessed 11 March 2021]. Available from: https://www.smashingmagazine.com/2015/06/efficient-image-resizing-with-imagemagick/
5. Setchell, Mark. ImageMagick: Lossless max compression for PNG?—Stack Overflow. Online. 3 December 2014. [Accessed 12 March 2021]. Available from: https://stackoverflow.com/questions/27267073/imagemagick-lossless-max-compression-for-png/27269260#27269260
6. Bellamy-Royds, Amelia, Cagle, Kurt and Storey, Dudley. Units for Measurements Reference — Using SVG with CSS3 and HTML5—Supplementary Material. Online. 5 October 2019. [Accessed 20 March 2021]. Available from: https://oreillymedia.github.io/Using_SVG/guide/units.html
7. —. ImageMagick is on fire—CVE-2016–3714. Online. 12 May 2016. [Accessed 8 March 2021]. Available from: https://imagetragick.com/
8. Leyden, John. ImageMagick PDF-parsing flaw allowed attacker to execute shell commands via maliciously crafted image. Online. 23 November 2020. [Accessed 8 March 2021]. Available from: https://portswigger.net/daily-swig/imagemagick-pdf-parsing-flaw-allowed-attacker-to-execute-shell-commands-via-maliciously-crafted-image

9. Inführ, Alex. ImageMagick - Shell injection via PDF password. Online. 21 November 2020. [Accessed 8 March 2021]. Available from: https://insert-script.blogspot.com/2020/11/imagemagick-shell-injection-via-pdf.html

