Study of the Effect of Multiple Writes on jpg Files

Dwight Sipler, small farm, New Canaan, CT

2021-03-10

Abstract

Many photographers try to avoid using jpg images. They feel that since the jpg is created using a lossy compression algorithm, successive re-writes of the image file will degrade the image significantly. Little has been written about the degree of degradation or how bad the degradation becomes and how many re-writes it takes to produce significant image degradation. For that reason I decided to do the experiment to evaluate the rate of change as a function of number of re-writes. The results of this study were not what I expected, given the handwaving arguments that abound in the various forums where this has been discussed, basically assuming that the degradation is a continuing process as many times as an image file is rewritten.

This study shows that much, if not most, of the degradation occurs in the first writing of the jpg file. The degree of degradation, as expected, will depend on the quality setting specified in the program doing the rewriting, since that determines the degree of lossy compression. However, instead of finding the image changes continuing unabated for as long as the re-writes continue, the degree of change saturates at some point. In other words, the image changes will increase with re-writes to a point, after which there will be no further change. That point will depend on the quality setting used for the re-writes as well as the details inherent in the image.

This study is a revision of an earlier study. The initial study considered only the luminosity comparisons, while this revision also includes color changes. This study repeated the application of the software with the color changes added. I used IrfanView to generate the sequentially re-written image files, and in this study I found a quirk in IrfanView that resulted in anomalous results in the earlier study. I was able to mitigate the problems produced by this quirk.

Another problem is evaluating the image, trying to compare it to the original image. Placing two images side by side is not the best way to see differences, particularly using limited resolution on computer screens or on printed copies. In this paper I describe an easy way to do a “blink test”, where a pair of images can be rapidly switched back and forth to spot areas of difference.

Writing a jpg

Before going on we should clarify what “write” means in this context. Also, the term “degraded” implies a decrease in quality. That is a subjective determination so I will use the word “changed” instead.

The data in a jpeg (or jpg as it will be referred to from here onward) is compressed to reduce the size of the file. The amount of compression can be determined by a parameter fed to the software doing the compression. The compression system used for a jpg is lossy, i.e. the compression discards some information. The compression is designed to minimize the effects of the loss on the quality of the image, but there is nevertheless some change to the image occurring on compression. In the context of this study, re-writing a jpg means that the data are written to a new file after the compression is done. The quality of an image is taken to indicate the degree to which an image matches the original image.

If a program uses a jpg, the image the program sees is slightly different from the original image before the compression was done. When the program finishes with the image, the data can just be discarded or, if so commanded, the program will produce a new copy of the image. Assuming the program is commanded to produce a new copy in the jpg format, then if the program has made no changes to the image there is no need to recompress the image so that the jpg that was read in can just be copied bit for bit to produce a file that is identical to the initial jpg. In such a case there is no change to the image and that will not fall under our use of the word “re-write” in this paper. If there has been a change in the data the program will produce a new jpg, and that will require a new compression of the data and we will call that “re-writing” a new file.

Using the computer's operating system to copy a jpg will produce identical copies and no changes to the data are made, so although technically a new file is written to memory that will not fall under our use of the term "re-write" here. And, of course, jpg files residing in the computer's memory will not change with time, barring system malfunctions.

Jpg is the most widely used format for images. The format is compact and is handled by all significant programs that use images: viewers, editors, word processors, browsers, etc. Many people prefer image formats that do not use a lossy compression algorithm, e.g. tif or png. I find that jpg is sometimes necessary. For example, iCloud for Windows will not accept any image formats except jpg. This study was undertaken to try to quantify the changes in the image resulting from multiple re-writes (re-compressions) of the jpg.

Measurements

Since we are considering only jpgs, the red, green, and blue color axes are represented by 8 bit numbers so the color channels are limited to a range of 0 to 255. The possible colors in a 3-component 8 bit numeric system will be contained in a cube with sides 255 units in length. Figure 1 shows the outer surfaces of the cube on the three axes Red, Green, and Blue. The left image shows the cube looking inward toward the origin (Red=0, Green=0, Blue=0), which is hidden behind the cube. The closest point is marked "White" and represents (255,255,255). On the right the view looking outward through the origin is shown so the closest point is black (0,0,0). These views show only the highly saturated colors possible. Many other colors are possible, but are contained inside the cube where they are not visible in this figure.

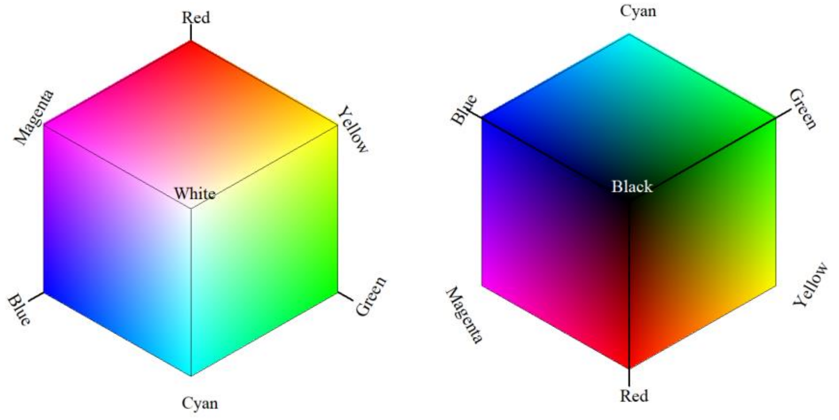


Figure 1: Two views of the color cube.

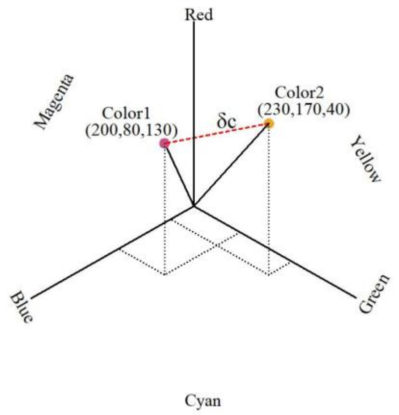


Figure 2: Two Color Vectors and Difference Vector δc

Each possible color is represented by a three-dimensional vector from the origin of the cube and contained within the cube. Removing the colored faces of the cube from Figure 1, we see in Figure 2 two arbitrary colors along with a difference vector, δc (dashed red line). The length of the color difference vector is given by:

$$\sqrt{(R_1-R_2)^2 + (G_1-G_2)^2 + (B_1-B_2)^2} \ . \qquad \text{Equation 1.}$$

The luminosity of a given color is just the length of the color vector. Luminosity differences between images for a given pixel are given by the difference between the individual luminosity values for color1 and color2 since luminosity is a scalar rather than a vector (i.e. it has no direction). The luminosity difference is given by the absolute value of:

$$\sqrt{R_1^2 + G_1^2 + B_1^2} \ - \ \sqrt{R_2^2 + G_2^2 + B_2^2} \ . \qquad \text{Equation 2.}$$

If the color of a pixel is changed it will be a new vector. It is possible that the luminance would not be changed, but the direction of the color vector will have changed. Conversely it is possible that the direction of the color vector will have changed but not the luminance. Subtracting vectors will result in difference vectors which are limited to be within the numeric color space. Looking at the changes in the image, we will consider the luminance and color changes as scalars. Although the color change is a vector quantity, we will disregard the direction of the vector and use only the length of the difference vector as given by equation 1.

I chose four images from my photopile to use for testing. All the images started as raw files and were converted to tif in Photoshop for use as reference images. The four images are shown below.

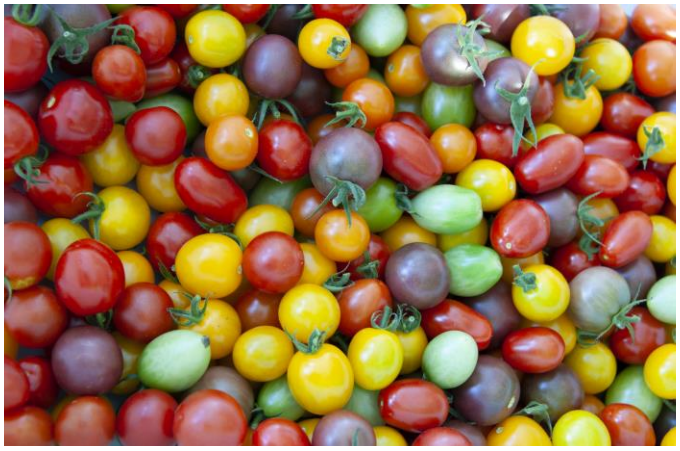


Image 1 3600x2400

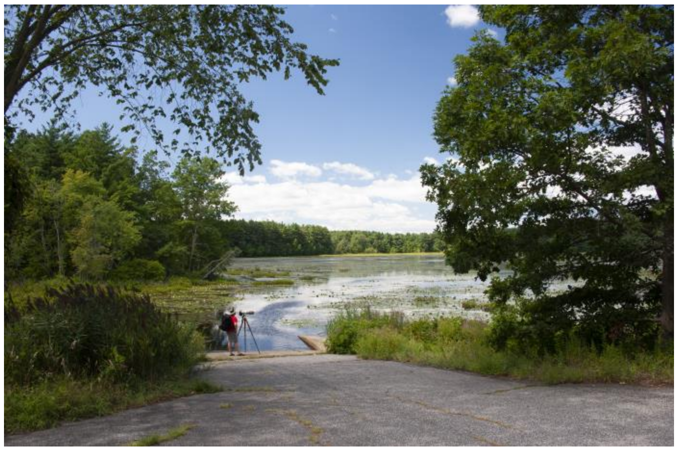


Image 2 4250x2800

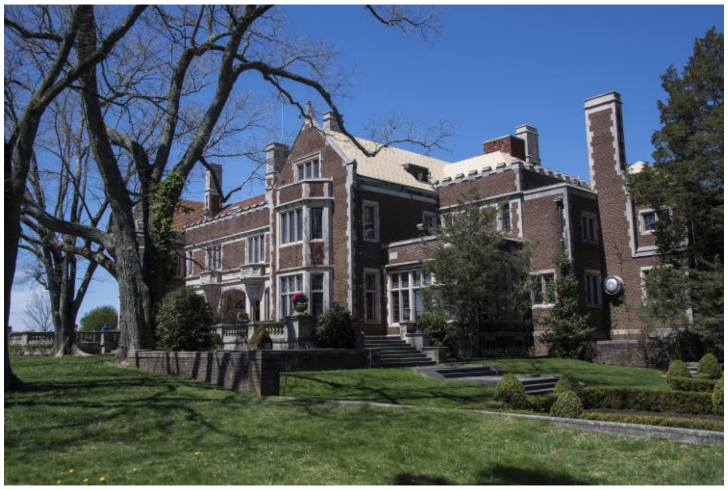


Image 3 5550x3700

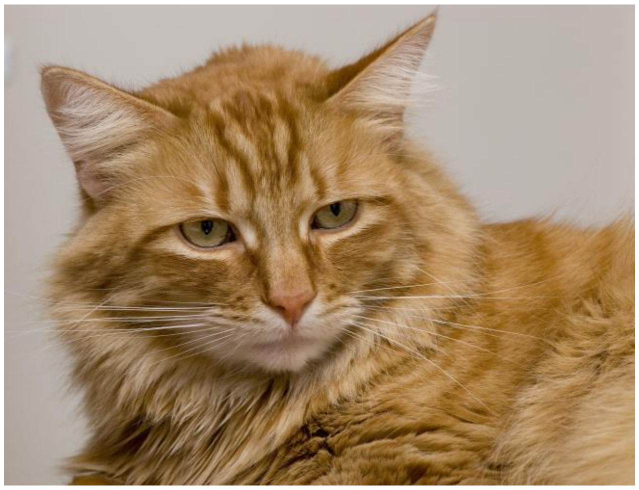


Image 4, 3550x2550

IrfanView is a popular freeware image viewer/editor (Windows only) that can write an image as a jpg with varying degrees of compression, specified by the IrfanView quality setting which ranges from 1 to 100. IrfanView also has the ability to run from a Windows command line, so I wrote a program to generate a batch file consisting of a sequence of command lines that write a jpg from the tif file over a full range of quality settings. That gave me 100 jpg files of varying quality. Applying a comparison program to find the RMS differences between the different jpg files and the reference tif, I generated a graph of RMS changes as a function of save quality from IrfanView (Figure 3).

FastStone is another popular freeware viewer/editor (Windows only) that can write a jpg with varying degrees of compression, specified by the FastStone quality setting which ranges from 0 to 100. Unfortunately FastStone does not have the ability to run from a Windows command line so I had to produce several jpg files by hand. To minimize the effort, I chose a limited selection of quality settings rather than produce all 101 jpg files (included in Figure 3).

Another program that can be used to generate a sequence of newly written jpgs is ImageMagick. This program is available for both Windows and Mac. It can be used from a command line in Windows so I generated a sequence of files using it and compared them to IrfanView. I chose some images for testing that had a range of colors, a fair amount of detail, and a small area of uniform color with a slight gradient.

I produced 100 images using each of the above images at 100 different quality settings using IrfanView, and 19 images using selected quality settings for each of the images using FastStone. I then compared the resulting images to the original tif file. A program was developed that loaded the original tif and the subject jpg as numeric arrays of color triplets. From the RGB data I compared each pixel from the tif with the corresponding pixel of the jpg, using equations 1 and 2 to measure the difference. I used these numbers to find the RMS difference over the image for both the luminance and the color changes.

Then, since I have three other choices for jpg production I generated a number of images for ImageMagick, Lightroom, and Photoshop and ran them through the comparison program as well. Results for Image 4 are shown in Figure 3 and represent the luminance and color changes between the original tif file and the initial jpg. The Y axis represents the RMS percentage change in the luminance and the color. The percentage change is obtained by dividing the observed RMS change by the largest luminance possible, i.e. white. Since white is given by (255,255,255) the luminance equals $255\sqrt{3}$, or about 441.7. Note that the color changes are larger than the luminance changes. The eye's sensitivity to color changes is smaller than its sensitivity to luminance changes[1] so the luminance changes will be more important. Figure 3 shows that for quality factors that are widely used (80 or larger) the luminance changes produced by converting a tif to a jpg are around 2% or less. Note that Photoshop only allows quality settings between 0 and 12 so the quality settings in Figure 3 are multiplied by 8.333 to plot them on the same scale as the other programs.

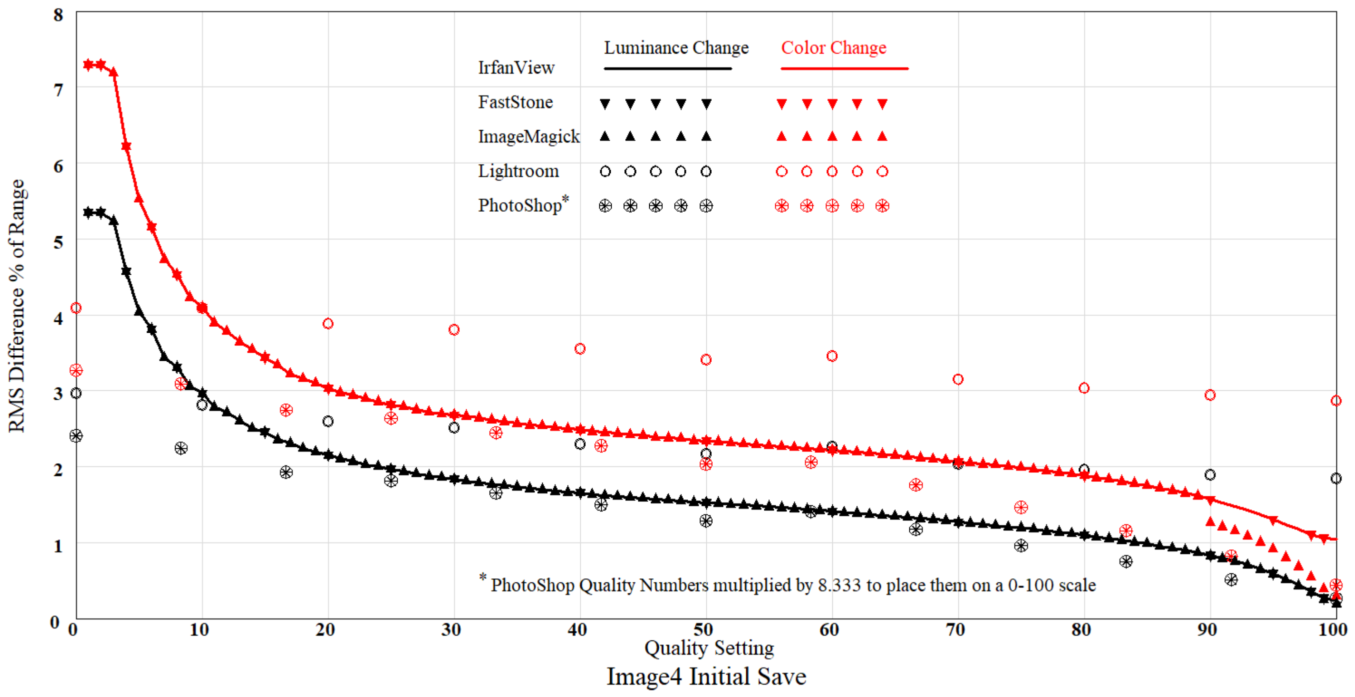


Figure 3: Image changes for the initial compression from tif.

The results from different program sources are denoted by different symbols and the IrfanView results are denoted by lines. The IrfanView and FastStone results are identical, so the different viewers are probably using the same algorithm to write the jpg at a given quality setting. The ImageMagick results are also largely identical until you reach a quality setting of 90, at which point they diverge from the IrfanView and FastStone results, but only in the changes in the color. ImageMagick does not subsample the chroma channels when the quality factor is at 90 or above[2]. I will be using IrfanView for further study since it has command line operation available, which FastStone does not. That makes it easier to produce a script to generate large numbers of re-writes. IrfanView also has a smooth curve for the color changes, which ImageMagick lacks. Lightroom and Photoshop appear to have limited the lower limit of the quality available in their settings, since the changes do not show the large rise in RMS differences seen at the lower settings of the other programs. Since the Adobe quality settings do not correspond to the quality settings used by the other programs. I tried to find a fit that would allow conversion of the Adobe quality settings to the other programs but was unable to find a fit that would overlay the curves.

As expected, the more compression is applied (smaller quality settings), the smaller the file will become. The file size for image 4 is shown in Figure 4. The tif file size is indicated at the top of the figure. Since it's a log scale, it is hard to compare the jpg sizes with the tif sizes but for quality settings below 90, the jpg sizes are generally more than an order of magnitude smaller than the tif. The interesting feature of Figure 4 is the difference in file size depending on the program used to perform the compression. The differences are around a factor of 3 in the middle range of quality settings, slightly lower at low compression settings. Since IrfanView, FastStone, and ImageMagick give identical changes with quality settings, the difference in image sizes probably results from non-image data in the file.

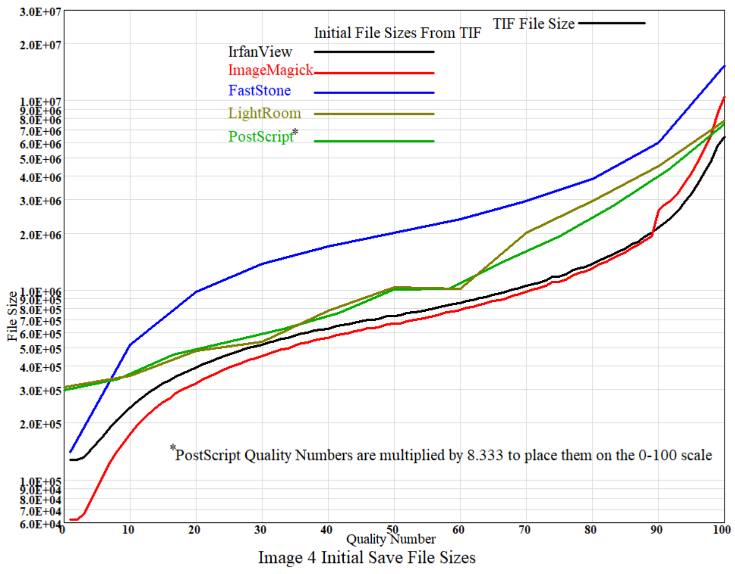


Figure 4: Image file sizes for the initial compression from tif.

Sequential re-write tests

Since IrfanView can be run from a Windows command line using a DOS command, a program was prepared that generated a sequence of command lines and wrote them to a batch file. The program started with a tif file and wrote a jpg with a given quality setting. The resulting file was named "ImageNSaveQ-nnn.jpg" where N is the image number (1-4), Q is the quality setting, and nnn is the number of the write starting with 000 (the initial jpg converted from the tif). The next iteration of the write took file ImageNSaveQ-000.jpg, loaded it into IrfanView, and wrote it as ImageNSaveQ-001.jpg. In this way a sequence of writes were produced with nnn ranging from 001 to some large number. Since the commands used the IrfanView command "convert" instead of "save", the exported jpg was produced with the compression algorithm applied. I used the resulting batch file to generate a large number of sequential files with a given quality setting.

Here I find things I did not expect.

The files so produced were compared with the original tif file using the method described above to determine the RMS changes in the image data for each file. While there is initially an increase in the RMS change of the image data with number of writes, the RMS change reaches a point where no further change occurs (saturation). The saturation point varies with quality setting, with lower quality settings saturating after fewer writes. The saturation takes three forms: (1) the RMS change increases asymptotically until it reaches a number that does not change with additional writes; (2) the RMS change rises at a moderate rate until it reaches a limit, after which it is constant; or (3) the RMS change reaches a number that then decreases a bit, but dithers around some value with additional writes. The dithering values seen here are small, generally in the 6th significant figure or further out. I would attribute this effect to rounding errors. Example forms of the saturation of the change are seen in Figure 5.

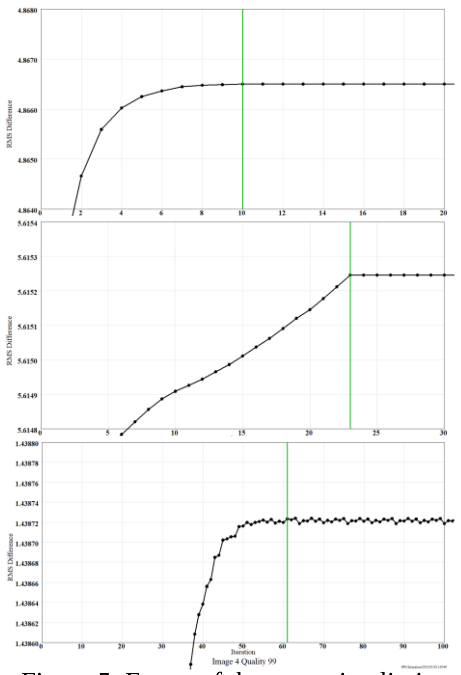


Figure 5: Forms of the saturation limit

The saturation of the change was not something I expected. But given the existence of the saturation, I would expect it to be dependent on the quality setting and that is what does happen. I would have expected that the higher quality settings would produce higher quality images all the way through the process. That appears to be the case generally for the four images I used for testing, although for images 1 through 3 the quality 99 and 100 runs show the results to be quite close.

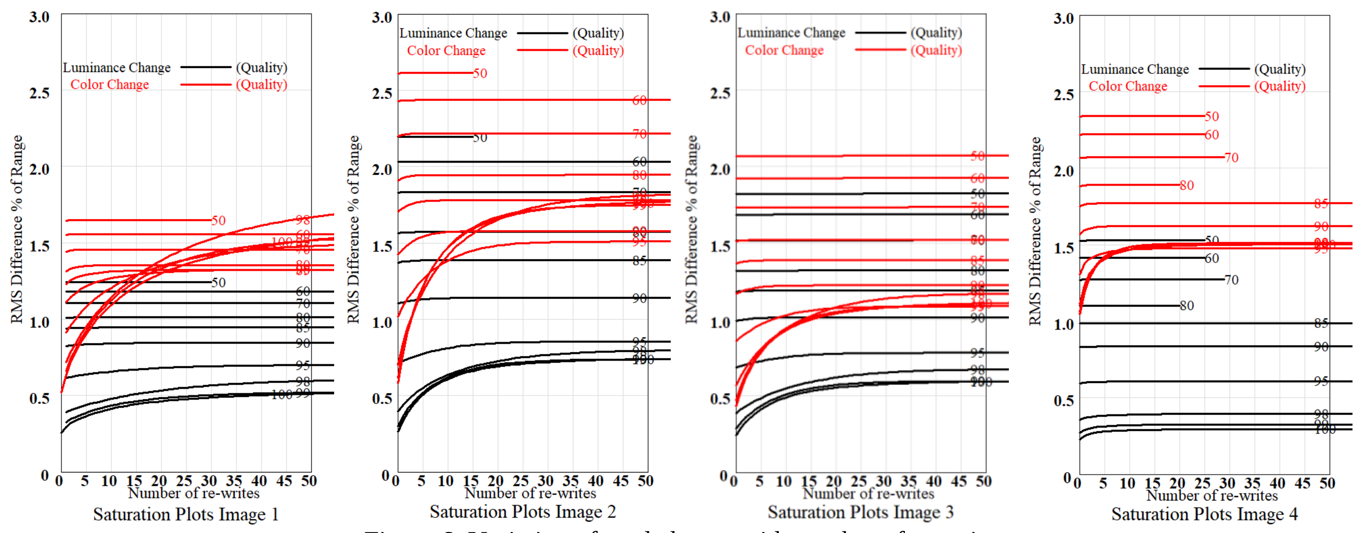


Figure 6: Variation of total change with number of re-writes

Figure 6 shows that for moderate quality jpgs (quality settings 50-80) the luminance change curves are fairly flat with the number of re-writes. The curves do in fact rise to a saturation point, but the change with number of re-writes is quite slow, so the initial compression produces most of the change and the slope of the curve is too low to be apparent in the graph. For higher quality jpgs (quality settings above 80) the luminance change curve show a rise at the beginning, and saturation is reached after a large number of re-writes. Figure 7 shows the saturation points for the above runs.

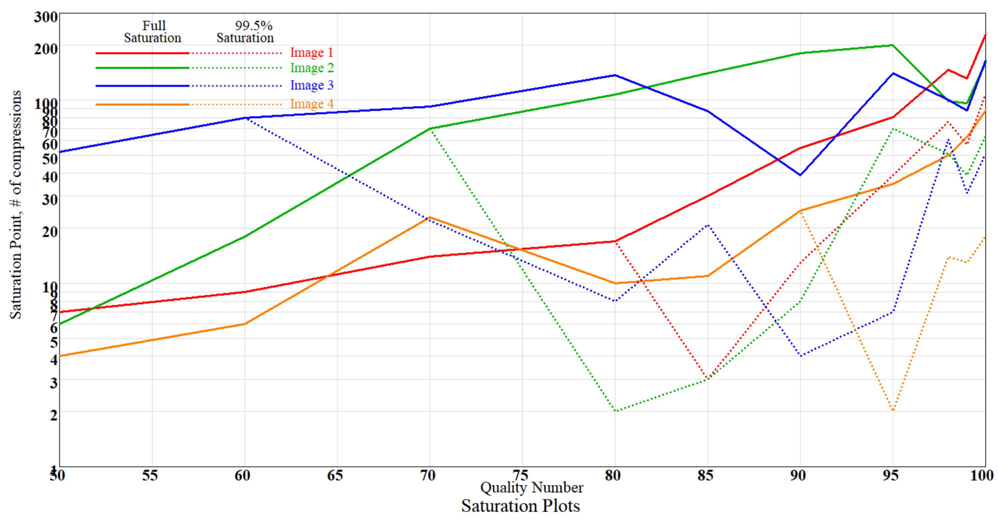


Figure 7: Saturation Point Plots

The results for different images are shown in different colors as labeled on the figure. Note the logarithmic scale to compensate for the large range of saturation points. In general, the lower quality settings reach saturation sooner than the higher quality settings. Since many of the plots approach the saturation point very slowly, I have also included the points at which the image reaches 99.5% saturation, i.e. the full saturation point differs from the dotted lines by only 0.5%. These “Effective Saturation” points are shown by the dotted lines on the plot. Most images reached 99.5% saturation in fewer than 25 recompressions. Higher quality (lower compression) plots may take more re-writes to reach that point.

Figure 8 shows the range of the change resulting from different quality settings for the 4 images. At quality settings below 80, the RMS change in the image does not change much from the initial jpg to the final re-write at the saturation point so the plot just shows a point. At higher quality settings the image takes more re-writes to reach saturation and the change is shown as a vertical line from the RMS change at the initial jpg to the final change at the saturation point. The points are offset slightly on the quality axis to show the range which would otherwise overlap for the various images.

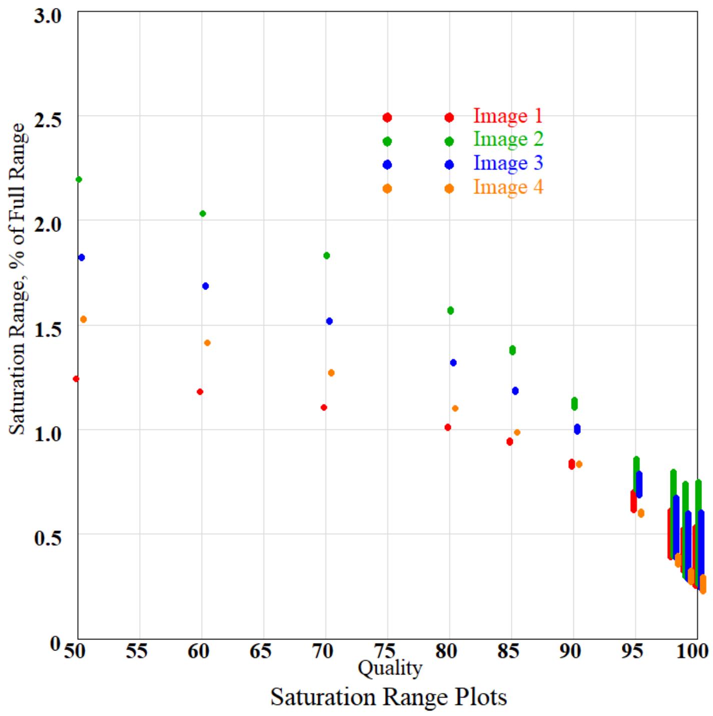


Figure 8: Saturation Point Range

From Figure 8 it can be seen that the first write at the lower quality settings produces most of the change. If that were not the case, the points would be stretched to vertical lines. The saturation occurs after only a few re-writes and the change measured by the RMS change in the signal does not increase that much. In contrast, the first write of image 1 at a quality setting of 99 produces an RMS signal change of only about 0.3%, but the RMS change increases to a bit more than 0.5% after 200 re-writes.

Since I had a program that looked at changes in the image, I added the ability to create a histogram of the changes. This was interesting to look at so I made a plot of the histograms from a sequence of re-writes (Figure 9). The plot shows the change histograms for all four images at the 10 quality settings [50, 60, 70, 80, 85, 90, 95, 98, 99, 100]. The lower quality settings are on the left. The red line shows the histogram for the initial jpg. It can be seen that changes increase with successive re-writes until the saturation point is reached (represented by the green histogram). As noted previously the saturation point is reached early in the lower quality settings so the green histogram is overlaid by the red initial histogram and is mostly not visible. The initial histogram is pretty much logarithmic (a straight line on the log scale) while the higher quality sequential histograms start to diverge around 2% of the full scale change (Figure 9).

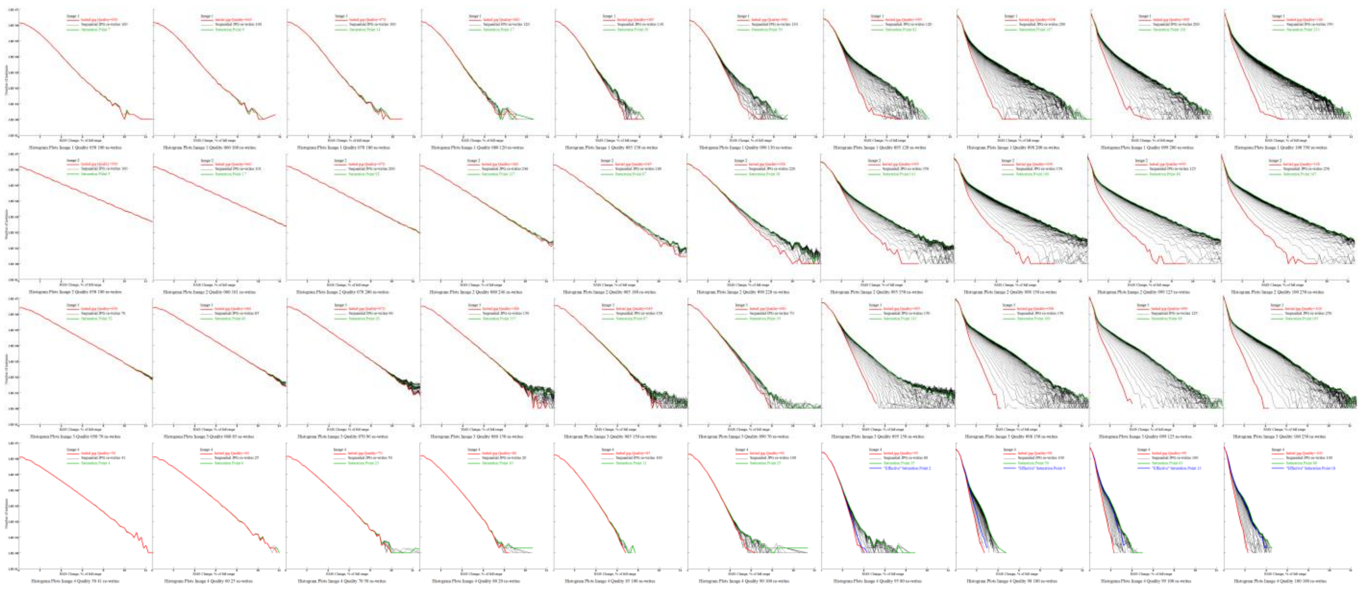


Figure 9: Histogram sequences.

As the quality setting increases, the slope of the histogram for the initial jpg increases, showing that the changes to the image decrease, as expected. However, when you get to a quality setting of 90 or above, larger changes start to show up earlier in the histogram. Also note that some of the images at quality settings above 90 show a lot of noise increasing with re-writes.

Since it is a bit hard to see this in the small plots I prepared animated gifs to show the sequences: http://small-farm.org/Test/Histograms-Image1.gif, http://small-farm.org/Test/Histograms-Image2.gif, http://small-farm.org/Test/Histograms-Image3.gif, http://small-farm.org/Test/Histograms-Image4.gif.

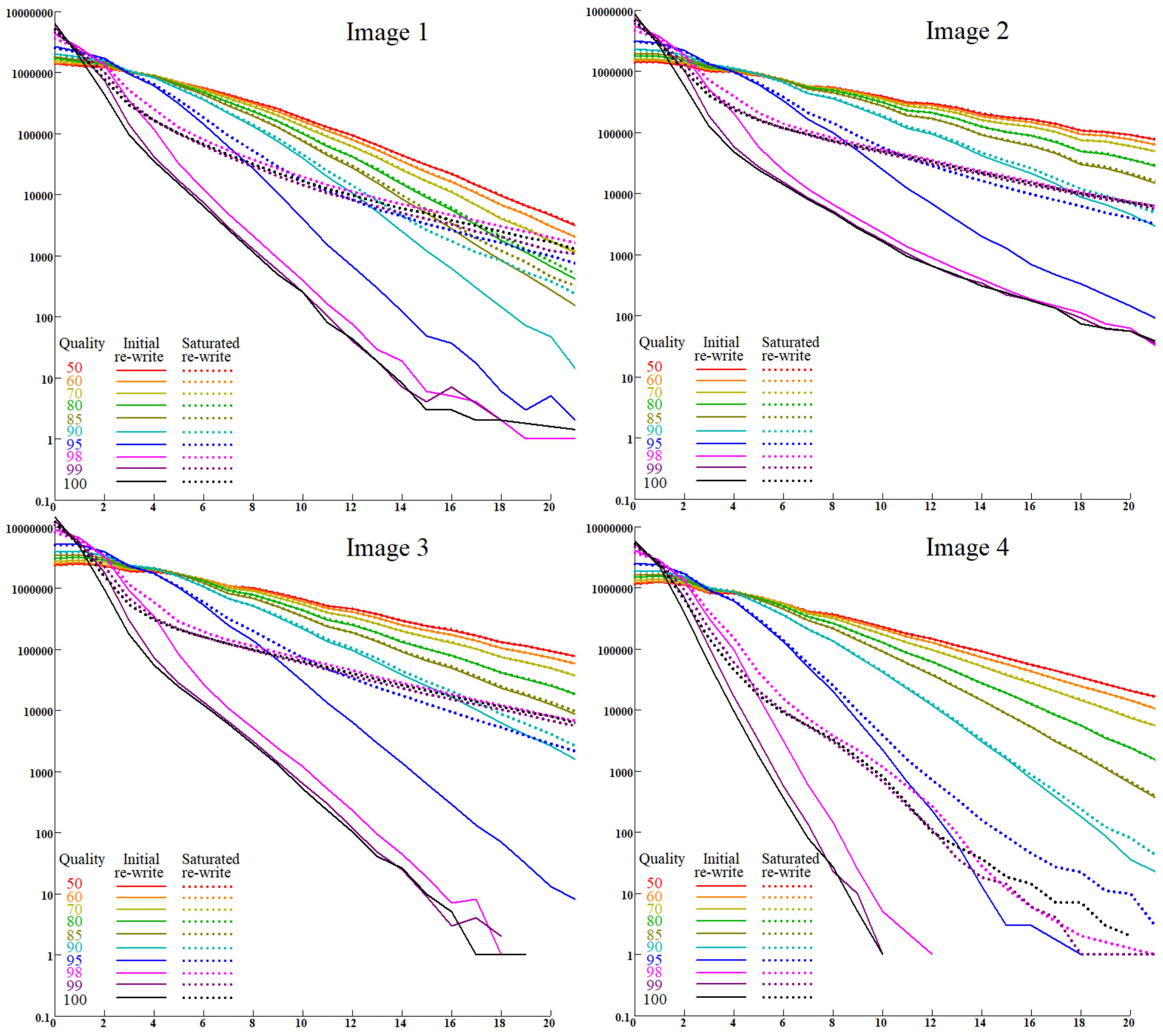


Figure 10: Comparison of histograms at initial compression and after the number of re-writes required to reach saturation.

In an attempt to show the value of the quality setting I plotted just the initial histogram and the histogram at the number of re-writes that saturated the changes. The four plots in Figure 10 show that the low quality settings lead to fairly low slopes of the histogram, implying that the changes are large. However, the saturated histogram shows little difference from the initial histogram, so the changes mostly occur in the first jpg compression. As the quality settings increase, the slope of the histogram increases, showing that the changes are less as the quality setting increases. However, when the quality setting rises above about 90, the changes start to diverge from the initial histogram. This implies that if the number of re-writes is expected to be small, the image should be compressed using the highest available quality setting. But if large number of re-writes are possible, the quality setting should be left below 90.

To this point the analysis was performed using my own software. There exist a few Python modules that will facilitate similar analyses, but the documentation is not always sufficiently detailed to determine the methods used for comparison. It is worth presenting the results of the publicly available software to compare with my above results.

The first module is ImgCompare[3], from an MIT project. This module has functions that take two images and return the difference and percentage difference. The function calculates the difference by looking only at luminance. I did something similar when looking at the luminance of the image, but I used equation 2 to sum the squares of the RGB channels to produce the luminance. The MIT program appears to convert to monochrome using a perceptual model, which weights the R, G, and B channels according to a standard model that I believe is based on the relative sensitivity of the eye to those colors. There will be some difference in the numbers obtained by this method compared to my above analysis. There are several different gray conversion coefficients[4]:

Gray = 0.2989 * R + 0.5870 * G + 0.1140 * B	(CCIR 601)	Equation 3A
Gray = 0.2126 * R + 0.7152 * G + 0.0722 * B	(ITU Recommendation BT 709)	Equation 3B
Gray = 0.2120 * R + 0.7010 * G + 0.0870 * B	(SMPTE 240M)	Equation 3C

I have seven different ways to find the changes to the image. To compare the different methods I looked at the MIT project and tried to determine how it was arriving at the numbers it gives. Documentation and online descriptions gave some clues but I was unable to reproduce its results exactly. Documentation states that it uses MSE (Mean Squared Errors) but this appears not to be correct. Imgcompare.image_diff gives a large number that is more consistent with the sum of the squared errors, but it does not divide by the total number of pixels to obtain the mean value. Accordingly, I compared the ImgCompare results with other methods of determining the sum of the square errors rather than the MSE:

1. My method using Equation 2 on full color images to determine the Luminance;
2. ImgCompare.image_diff from full color images;
3. My method using Equation 3A to convert the images to monochrome;
4. My method using Equation 3B to convert the images to monochrome;
5. My method using Equation 3C to convert the images to monochrome;
6. My method using Python's Luminance conversion to convert to monochrome;
7. ImgCompare.image_diff from images converted by Python's Luminance conversion.

I chose Image 2, quality setting 95 for the comparison since it shows a nice rising curve at the beginning and saturates after a long sequence of re-writes. The result is shown in Figure 11.

The curves are different but several of them cluster at the bottom of the graph. The two that are most different are my original method using Equation 2 and the ImgCompare method using the full color images. The method producing the lowest error sums is the ImgCompare method using the Luminance conversion in Python.

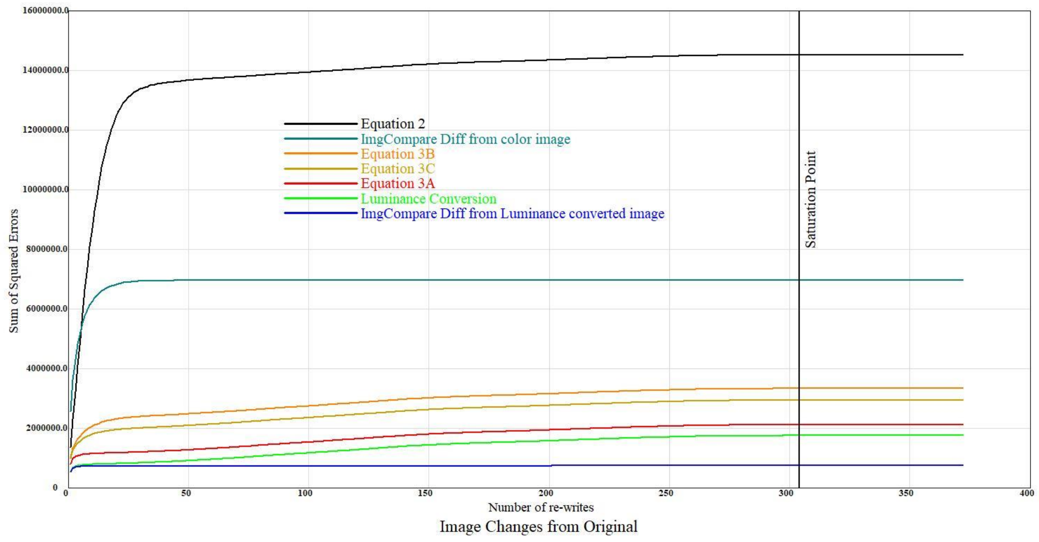


Figure 11: Comparison of Summed Squared Errors.

There is a factor of 20 between my initial method and the ImgCompare method using monochrome images. It appears from Figure 11 that the results are generally smaller when the images are first converted to monochrome. The conclusion that the errors reach a saturation point is unchanged, and all the curves in Figure 11 reach saturation at the same number of re-writes. Since my initial method generates the largest errors I consider it to represent a worst case scenario. I note also that my analysis generates a slightly different shape to the curves, with the ImgCompare curves being flatter overall.

Another available method of comparing images is the Structural Similarity Index Measure (SSIM) developed by Wang et al.[5,6] given by Equation 4 (see Wang et al. or Wikipedia for definition of the elements of this equation). This method compares subsamples of the images

$$\mathrm{SSIM}(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$ Equation 4

Again, this method is available in a Python module[7], but the details of how it performs the subsampling is not given. The range of the output from this program is 0-1, where 0 is no similarity and 1 is perfect similarity (identity). The first attempt to use SSIM on the image used for comparison of the ImgCompare package failed because the program wanted to allocate memory to hold a 90 MByte array of numbers in 64 bit floating point and my computer was not able to do that. I took the image and tried two things: I reduced the size of the image in Photoshop by reducing the dimensions by a factor of 2; and also tried cropping the image to a similar size. With both the smaller images I was able to run SSIM. To maximize the change I used images with a quality setting of 50. The number I got from the program was greater than 0.999 for a small set of re-written images with small changes in the 4th significant figure. Since SSIM is supposed to evaluate the structure of the images and since I was unable to see changes visually (using the Blink Test) the large similarity between the original images and the re-written images is reasonable. Since I don't know how the subsampling is handled, I don't really have a way to evaluate that number as a way to identify where the differences are. Wikipedia (op. cit.) states that there exists a way to map structural differences but I did not attempt that in this study.

This was purely an experimental study, and did not delve into the theoretical aspects of the writing of a jpg. Since the saturation was something I did not expect I looked online to see if others had observed it. Most of the articles I found just stated something like "I re-wrote the jpg 30 times and could not see any difference", after which they showed two images representing before and after. Looking at two separate images is not a good way to see differences. At the minimum you need to have a way to switch overlaid images from before to after quickly and conduct a blink test to spot differences, since the differences are usually small and occur at constrained spots in the image. However, I did find a couple articles that could be relevant[8,9,10].

The 4 test images used in this study are available at http://small-farm.org/Test/Image1.tif through http://small-farm.org/Test/Image4.tif.

Summary

Test images were used to compare the results of multiple writes of a jpg file. Numeric tests were used to produce quantitative measurements of differences produced by the multiple writes. The changes observed are dependent on the compression used in the writing of the jpg file, and in general higher compression leads to larger changes, but subjective by-eye comparisons were unable to detect differences large enough to be significant for the upper half of the range of available quality settings determining the compression. Successive writes of the jpg file did produce numerically detectable changes in the image but only up to a point at which saturation of the changes occurred. The number of re-writes required to reach saturation is widely variable and depends on the details of the image and the degree of compression, but in general, most of the changes occur in the initial conversion from a tif to a jpg. Since the amount of change rises very slowly to the saturation point, an "effective saturation" point can be defined at 99.5% of saturation. This point occurs significantly sooner than saturation for lower compression images. For the high compression images studied, the change from the initial point to complete saturation was less than the 99.5% point.

Histogram plots suggest that large numbers of re-writes can increase low level noise in the image when the compression is low.

Alternative analysis tools were not sufficiently well documented to allow a good comparison between my method and canned programs. However, the magnitude of the changes was significantly higher for methods that used full color images, implying that conversion to monochrome is better done by accepted algorithms rather than the method I used.

Conclusion: re-writing a jpg does not change the program continually. There is a limit. The changes noted for the images I used for testing were not large enough to convince me to avoid using jpg for image files. However, the noise increase shown in the histogram plots means that if it is expected that an image would be re-written many times, the quality factor should be kept below 90.

Tasks accomplished in this study:

- Polishing my Python Proficiency.
- Generating a set of sequentially re-written jpg files from the original tif file for 4 different test images and finding the RMS change in luminosity as a function of re-writes.
- Evaluation of changes observed due to multiple re-writes of jpg images and development of criteria for determining when it could be a problem.
- Defining a PostScript process to create a blink test for subjective by-eye evaluation of two images.
- Discovery of a quirk in IrfanView that affects the output from conversions when multiple instances of IrfanView are running simultaneously.

Appendix 1: Blink Test

I took one of the 4 test images as a tif and saved it at a quality I wanted to study. I then ran a script in Photoshop (File => Scripts => Load Files into Stack) that would load the original and the compressed copy into Photoshop as layers (Figure 12). That brings up a dialog you can use to define the files you want to load (Figure 13). Clicking on "Browse" brings up a normal navigation dialog you can use to select the images you want to compare.

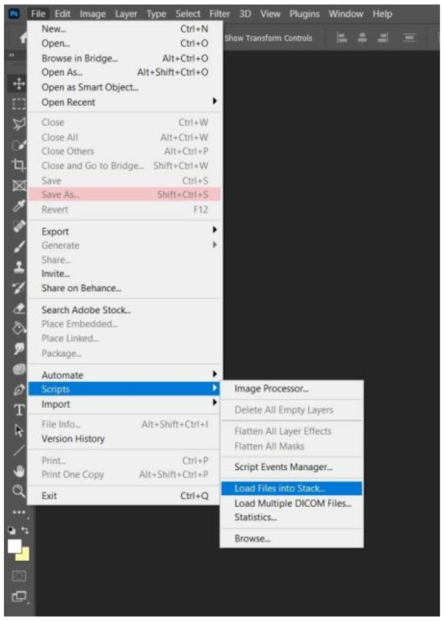
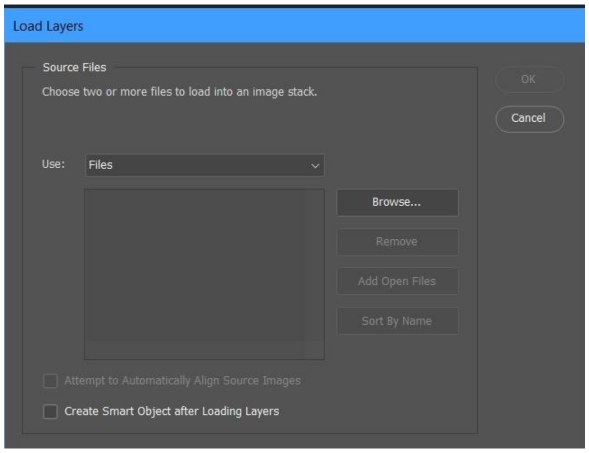


Figure 12: Postscript stack loading script.



Load Layers

Source Files

Choose two or more files to load into an image stack.

Use: Files

Browse... Remove Add Open Files Sort By Name

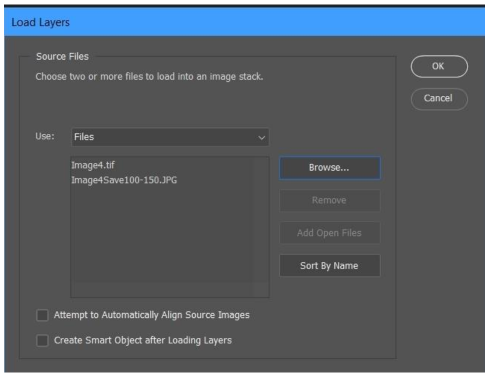
OK Cancel

☐ Attempt to Automatically Align Source Images

☐ Create Smart Object after Loading Layers

Figure 13: Load dialog.

When you select the files you want to compare, they will be shown in the load dialog (Figure 14). Clicking on "OK" will load the images into separate layers (Figure 15). You then bring up the timeline dialog with Window => Timeline (Figure 16). That is where you will define the order of the appearance of the layers in the gif and also define the duration of each image in the animation. Selecting the Timeline will bring up a panel in the Photoshop workspace (Figure 17). On the Timeline panel you will click on the button "Create Frame Animation" (circled).



Load Layers

Source Files

Choose two or more files to load into an image stack.

Use: Files

Image4.tif
Image4Save100-150.JPG

Browse... Remove Add Open Files Sort By Name

OK Cancel

☐ Attempt to Automatically Align Source Images

☐ Create Smart Object after Loading Layers

Figure 14: Load dialog.

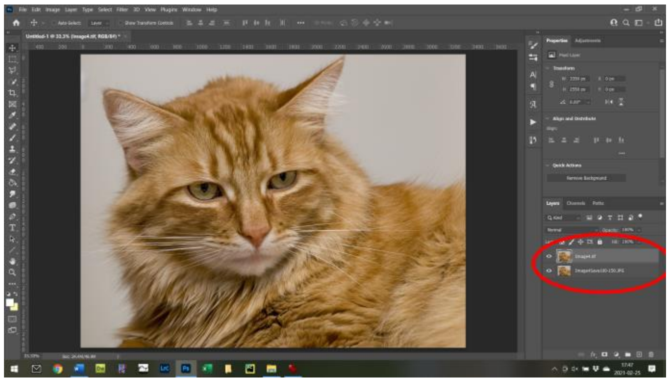


Figure 15: Two files loaded into layers.

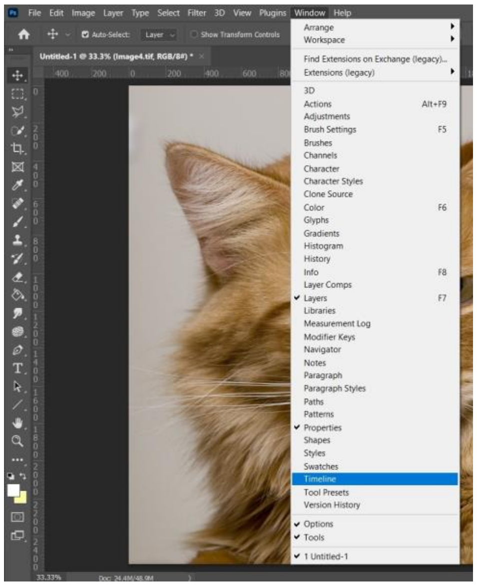


Figure 16: Select Timeline.

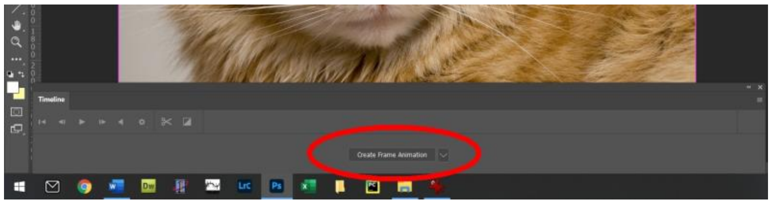


Figure 17: Timeline Panel.

Clicking on the frame animation button will show a three-bar menu icon on the Timeline panel (arrow on Figure 18).

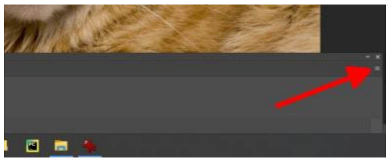


Figure 18: Timeline Menu.

The timeline menu will enable you to create animation frames from the layers (Figure 19).

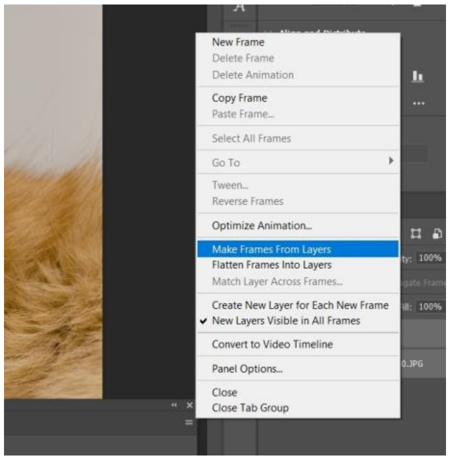


Figure 19 Define frames.

If the layers get placed in the frames so that the last frame is displayed first, you can reverse the frames using the Timeline menu (Figure 20).

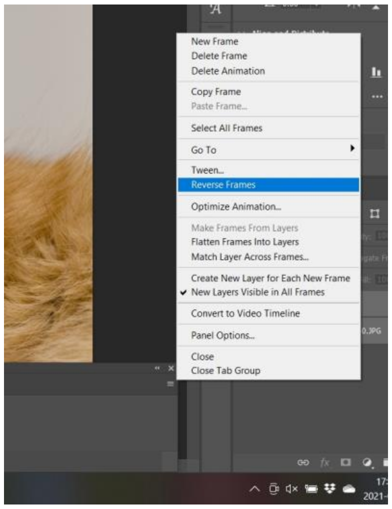


Figure 20: Define order of frames.

When you have the sequence set up properly it is time to define the duration the frames are displayed (Figure 21). The small frames shown on the Timeline have the duration displayed just below the image. The small arrow beside the duration will bring up a dialog with choices for a display duration. You can define something other than the choices displayed there if necessary. I find it useful to use a longer duration for the last frame. This allows you to keep track of where you are in the animation. Figure 22 shows the last frame with a long duration.

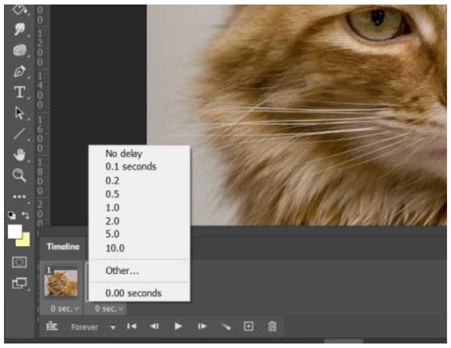


Figure 21: Define duration of frames.

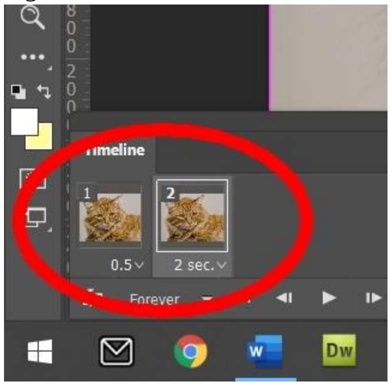


Figure 22: Asymmetric duration.

Once you have this all set up you can run the animation within Photoshop (Figure 23).

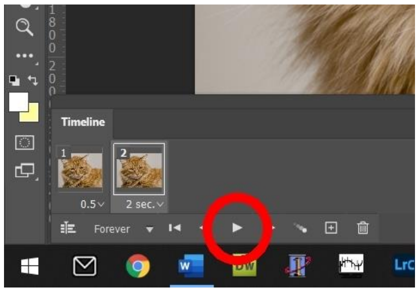


Figure 23: Run the animation.

If you want to use the animated image outside of Photoshop you can save it as an animated gif. This gives you the opportunity to modify parameters if necessary before saving the animation as a gif file. You can do the comparison within Photoshop without needing to save the file as a gif, but if you need to share the results, you can save your work as follows:

File => Export => Save for Web (Legacy) as shown in Figure 24 will bring up a dialog that you can use to define the gif parameters (Figure 25).

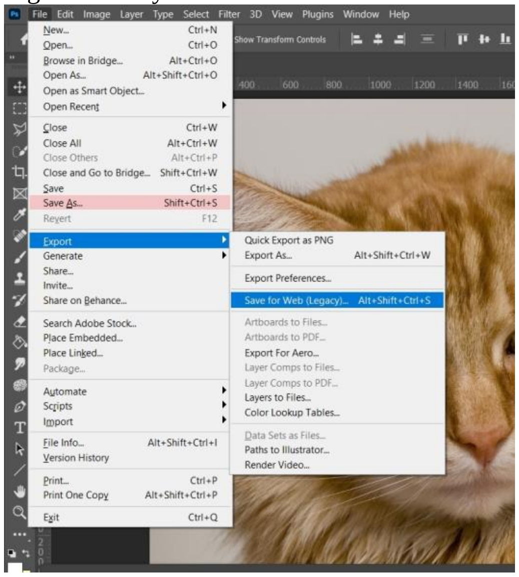


Figure 24: Export as a gif.

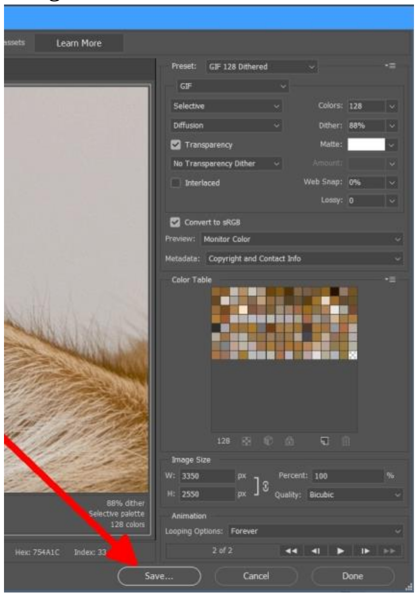

Preset: GIF 128 Dithered
GIF
Selective Colors: 128
Diffusion Dither: 88%
Transparency Matte:
No Transparency Dither Amount:
Interlaced Web Snap: 0%
Lossy: 0
Convert to sRGB
Preview: Monitor Color
Metadata: Copyright and Contact Info
Color Table
Image Size
W: 3350 px Percent: 100
H: 2550 px Quality: Bicubic
Animation
Looping Options: Forever
2 of 2

Figure 25: Save it.

I went through the above steps to create an animated gif from the two images so I could see them flip back and forth at a 0.2 second interval. It is not necessary to follow through to actually save the animated gif since the animation can be observed in Photoshop. The flip interval is something you can adjust in Photoshop. It gives you choices of 0.2, 0.5, 1.0 second, or many other intervals, but you can edit them to any value you want. It might be advantageous to use different intervals for the two images so you can tell which image has a good or bad feature. For example using 0.2 and 0.5 second would probably work well. You can use more than two images in the blink test but I think that would be more difficult to keep track of what you are looking at.

I could not see any difference by eye until I reached fairly low quality settings (Q). For Image1, Q=30; Image2, Q=40; Image3, Q=35; and Image4, Q=35. Note that you can do the blink test with the full image but in order to see changes it is useful to expand the image so you can see individual pixels. I have Photoshop set to expand or contract the image with the mouse wheel so it's easy to look at areas of interest. In Photoshop, Edit => Preferences => General => check "Zoom with Scroll Wheel".

A shortcut is available if you don't want to go through the gif animation procedure. As in the beginning of the animation process you load two images into layers. You then merely turn the visibility of the top layer on and off, thereby switching the image being displayed. The only drawback to this method is that it is manual rather than letting the process run automatically.

This is purely a subjective test. It is entirely possible that others could see differences at higher quality settings.

Statistics of the study (in case you want to try it yourself):

Test Images: 4, containing between 8.6 and 20.5 Mbytes.

Test Files: Generally 150 jpgs produced in sequential re-writes for each image at each quality setting. 10 different quality settings were used: [50,60,70,80,85,90,95,98,99,100]. Some images required over 300 re-writes to reach saturation. Along with test images from programs other than IrfanView, over 20,000 image files were generated, requiring around 80 GBytes of storage.

Computer time: Comparison of a jpg with the original tif took about 1-3 minutes on a laptop and a similar time on a desktop. Comparing 150 files took around 3-6 hours. The longest run took 11.5 hours comparing 600 files. I did not keep track of total computer time but I estimate it was more than 400 hours. Additional computer time was added trying to reconcile my algorithm with the publicly available algorithms. Many runs were done with both the laptop and desktop computers running overnight while I was asleep.

---

[1] https://vanseodesign.com/web-design/color-luminance/
[2] https://imagemagick.org/script/command-line-options.php#quality
[3] https://pypi.org/project/imgcompare/ is an MIT program for comparing image files.
[4] https://en.wikipedia.org/wiki/Luma_(video)
[5] http://www.cns.nyu.edu/pub/eero/wang03-reprint.pdf
[6] https://en.wikipedia.org/wiki/Structural_similarity
[7] https://www.pyimagesearch.com/2014/09/15/python-compare-two-images/
[8] https://photo.stackexchange.com/questions/104177/how-can-i-simulate-jpeg-quality-degradation
[9] https://photo.stackexchange.com/questions/99604/what-factors-cause-or-prevent-generational-loss-when-jpegs-are-recompressed-mu
[10] http://www.faqs.org/faqs/jpeg-faq/part1/section-10.html