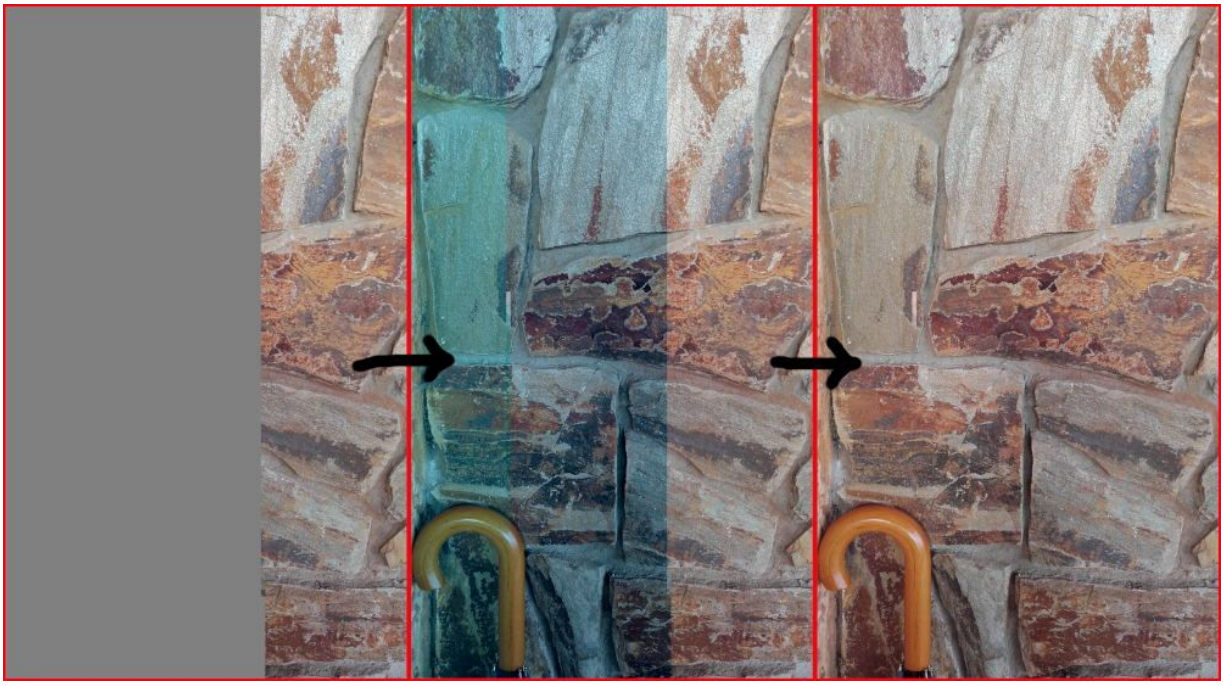


Repairing corrupted JPEG images

by Alberto Maccioni

This tutorial describes a workflow used for correcting JPEG images that exhibit various corruption artifacts.

1 (Essential) Introduction to the JPEG format

JPEG images are generated from bitmap images by removing some amount of detail according to a quality setting. This operation is performed on elementary blocks of 8x8 pixels.

Essentially, pixel data is transformed into frequency coefficients in a process known as 2D DCT; these coefficients are reduced in size in a way that preserves the perceived quality, then stored in a compressed form.

How exactly this is done is not essential for our purpose, however we have to understand a few additional details:

- pixels are first transformed from RGB to YCbCr; this means that luminance (Y, i.e. the gray scale representation) is coded separately from chrominance (Cb&Cr, i.e. color information).
- Y and C can be (and often are) sampled at a different resolution; that's because the human eye is more sensitive to luminance than chrominance. So the image is further subdivided in Minimum Coded Units comprising a certain amount of Y and C blocks, usually with more Y than C.

  Example: YYYYCbCr; this has 4 times as many Y as Cx; so the MCU is 2x2 blocks or 16x16 pixels, with full Y data and Cb/Cr sub-sampled at ½ the resolution in width and height. There are also other schemes possible.

- Y and C blocks are stored as a list of coefficients, with the first one, the DC coefficient, depending from the DC of the previous block; in fact it is coded as the increment from the previous one. Note that DC levels are chained per-component: all Y depend from each-other, all Cb together, all Cr together as well.
- Coefficients are compressed using Huffman coding (also rarely Arithmetic coding), resulting in strings of bits of any size. This means that it's not easy to understand where one coefficient ends, and also that editing cannot be done byte by byte. That is the reason why in general a hex editor is not useful for modifying or correcting an image.
- The layout of a JPEG image is divided into sections, separated by markers that are byte-aligned and begin with 0xFF. Each describes an aspect of the image, like size/organization, EXIF data, quantization tables, compression tables, etc. Stream data (the variable bit coefficients that represent the image) could also contain bytes at 0xFF, in which case a 0x00 byte is appended (this is called bit stuffing).
- In order to increase robustness against data loss, some images make use of Restart Markers. These markers are inserted every N MCU (a configurable number), and signal a break in the DC level chain; the subsequent block will code its DC level as a pure number and not as a delta with respect to the preceding block. In this way, any DC error will propagate at most up to the next restart marker.

There are may more details that are outside the scope of this guide and not essential for our purpose; they are nonetheless very interesting; here some links:

https://en.wikipedia.org/wiki/JPEG

www.impulseadventure.com/jpeg-compression (from WebArchive, as it recently went offline)

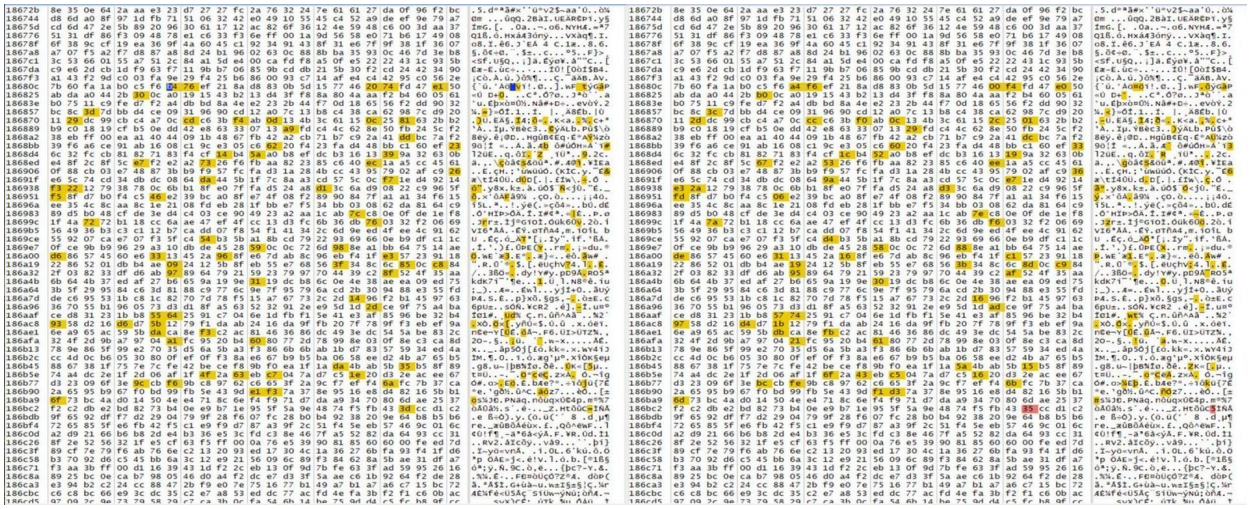
2 JPEG image corruption

Image corruption has always been present, but lately it is becoming more common for a number of reasons: a lot more pictures are generated today; pictures are getting larger; storage media is often a memory card, which is inherently less reliable than an HDD or SSD.

Examples of JPEG image corruption: false colors, misalignment, missing sections.

Notice that corruption starts at a precise point in the XY scan, and proceeds from there until the end; frequently more than one corruption point are present.

Bit errors are often localized in a small area of the image file; not all bits and bytes are affected; as an example, below a comparison between a corrupted file and the original version.



There is a certain probability that changing even a single bit in the stream, a compression code prefix (Huffman code) is changed; the resulting code may signal a different bit length than the original one, therefore not only that coefficient is affected, but also the subsequent one, and so on.

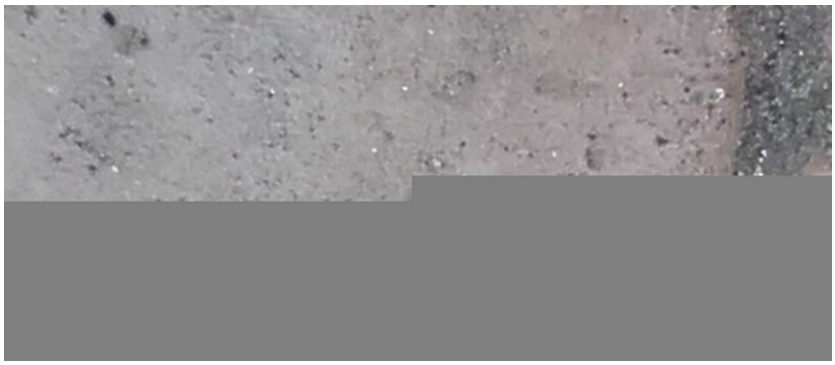
Fortunately it is very common that at some point after the error, and it may be many blocks down the line, the decoding process recovers the correct coefficient ordering; I don’t know the reason, it’s probably some kind of mathematical property of the Huffman coding.

Anyways, between the error occurrence and the recovery point we are left with a certain number of blocks that show more or less random coefficient values; the interpretation of this data by the decoding algorithm may lead to various artifacts.

2.1 DC level errors

These are due to the fact that a DC coefficient is expressed as variation with respect to the previous one; therefore, once a DC coefficient is corrupted, all subsequent blocks, even when free of errors, will result into wrong values.

For example, let’s suppose that as a consequence of a bit error the DC coefficient for a Y block is interpreted as 1500 instead of 10; this means that the luminance value will be extremely high and possibly will saturate the output range; regardless of the other (AC) coefficients in the same 8x8 block, the block will be rendered with a uniform color (exactly what color is codec-dependent).



The subsequent block will start from the same brightness level (plus its own DC value); it will likely be rendered as out of range as well. The same happens from now to all remaining blocks, hence the image that appears of a uniform color from this point on.

It is interesting to note that, save for the damaged block, all pixel information is still present but is simply rendered incorrectly.

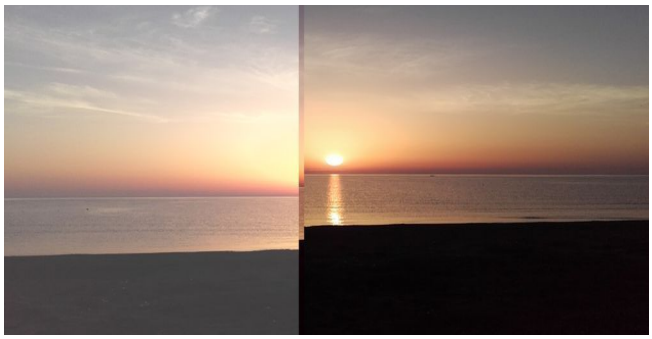
The same type of corruption can result into various degrees of high or low brightness or, if the error happens to Cb/Cr blocks, weird colors can appear.

It is actually very common that more than one block is affected at the same time, so both Y and Cb/Cr get corrupted.

2.2 Alignment errors

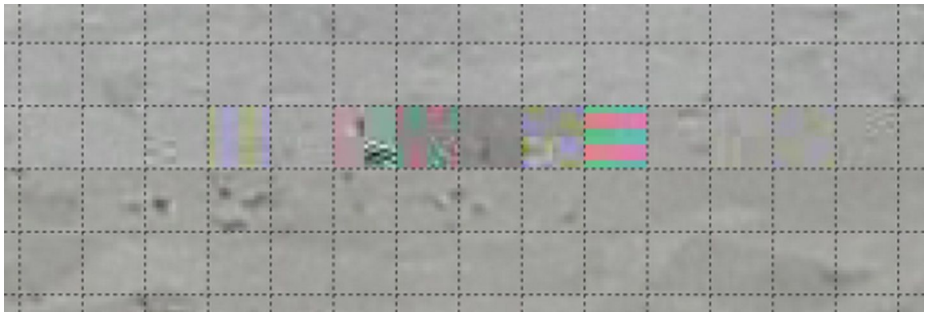
A possible and frequent effect of bit errors is that blocks are terminated early; this is somewhat codec-dependent, as the JPEG standard does not describe the behavior in case of impossible situations like an undeclared Huffman code or more than 64 coefficients. Anyways, it is often the case that in place of a number of original blocks, the decoder produces more of them, also disrupting the sequence between Y and C (note that there is no way to distinguish them other than counting).

By the time the decoder recovers the correct sequence and block order (if it does!), the rows will certainly be misaligned, like in the image below:



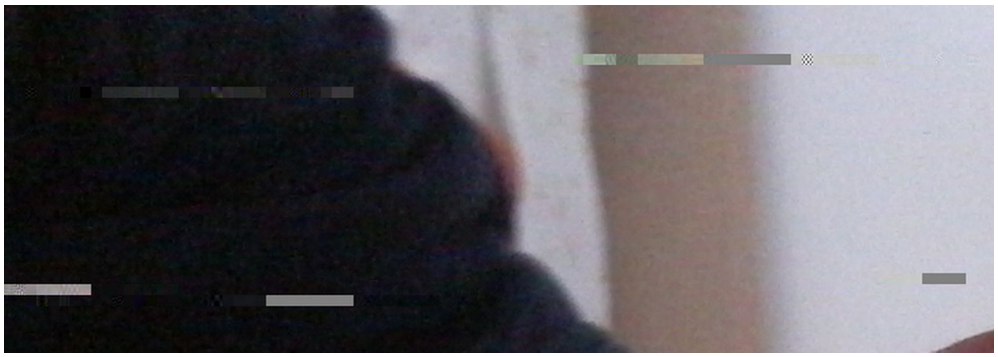
2.3 AC errors

AC coefficients in a block define the shapes present, i.e. the fine detail. A corrupted AC coefficient may be visible or not, depending on its value. A small level of error checking is given by the hard limit of 64 coefficients per block. However, the highest impact of an AC coefficient error is that it can affect all subsequent coefficients if its length is decoded incorrectly; this can even extend to subsequent blocks, and theoretically to a whole image (although it is very improbable).



2.4 Errors with Restart markers

When present, restart markers limit the extent of artifacts to a finite number of MCU; however it is always possible that the restart marker itself is corrupted. In these cases, most codecs ignore data until the next correct marker is found, so the only visual effect is that parts of lines are of a uniform color; the overall shape is still correct and also colors are unchanged.



2.5 Header corruption

The first part of a JPEG image contains various sections that specify how to interpret stream data. Errors in this area will generally result in unreadable images.

Copying the header from a similar but readable image should fix the issue.

3 Tools

Several tools are needed in order to correct JPEG images according to the proposed flow; they are all free and open source.

3.1 Jpeg-decomp

https://github.com/albmac/jpeg-decomp

This command-line utility is able to extract all stream data into a text file with several tags specifying various kinds of information. Once in text form, it is easy to edit individual component blocks in order to correct different kinds of image artifacts. The -encode command can re-generate an image from the edited text file.

The utility supports Huffman coding, restart markers, standard APP sections; does not work with progressive-scan images or arithmetic coding. The supported features correspond to the majority of images produced by cameras and smartphones.

Command option	Description
-fin <filename>	Input file
-fout <filename>	Output file
-decode	Decode JPEG image into text format
-encode	Encode text format into JPEG image

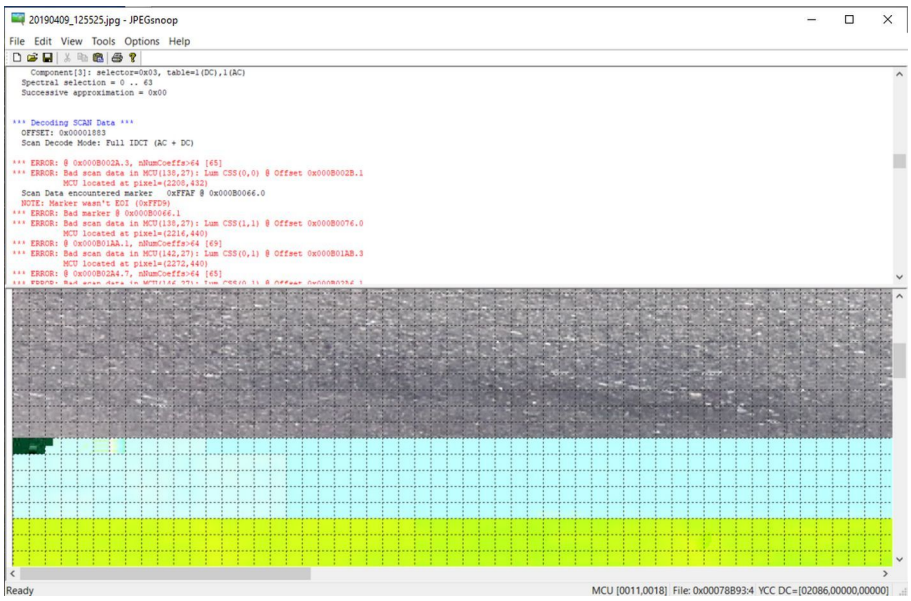
Text file format:

Tag	Description
// or #	Comment; lasts up to the end of the line. Useful information is produced by the -decode command, such as MCU coordinates and the decimal value of DC and AC coefficients, along with error warnings.
<raw>...</raw>	Raw data in hex form; each line starts with 0x. This is non-interpreted data such as headers. It is written directly in the output file when using -encode
<dht>…<dht>	Define Huffman table. First parameter is the table type: YDC, YAC, CDC, CAC; following is a list of elements in brackets [], each holding 3 hex values. Only non-standard Huffman tables are generated by -decode.
<y>...</y>	Luminance block. The first number is the DC level expressed as decimal number; following is a bit string starting with 0b that represents AC coefficients. Missing DC or AC coefficients are replaced with the code representing 0. It uses YDC and YAC Huffman tables.
<c>...</c>	Chrominance block. The first number is the DC level expressed as decimal number; following is a bit string starting with 0b that represents AC coefficients. Missing DC or AC coefficients are replaced with the code representing 0. It uses CDC and CAC Huffman tables.
<restart>N</restart>	Restart marker. N is between 0 and 7.
<eoi></eoi>	End of Image marker: not necessary as -encode inserts it anyways.

3.2 JPEGsnoop

https://github.com/ImpulseAdventure/JPEGsnoop

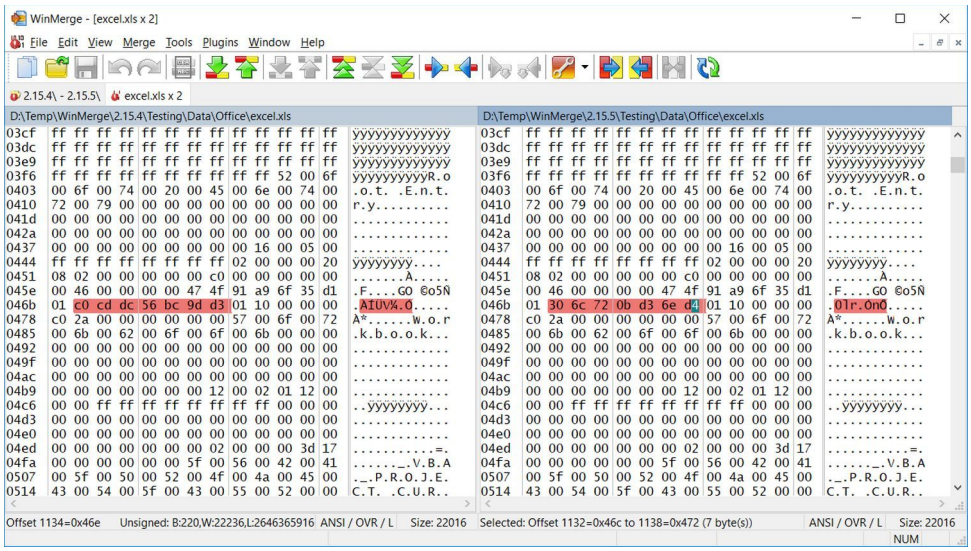
This invaluable tool can display and decode most information about a JPEG image. It is essential to locate artifacts in terms of MCU coordinates, as well as visually match brightness levels, as explained further down.



3.3 WinMerge

https://winmerge.org/

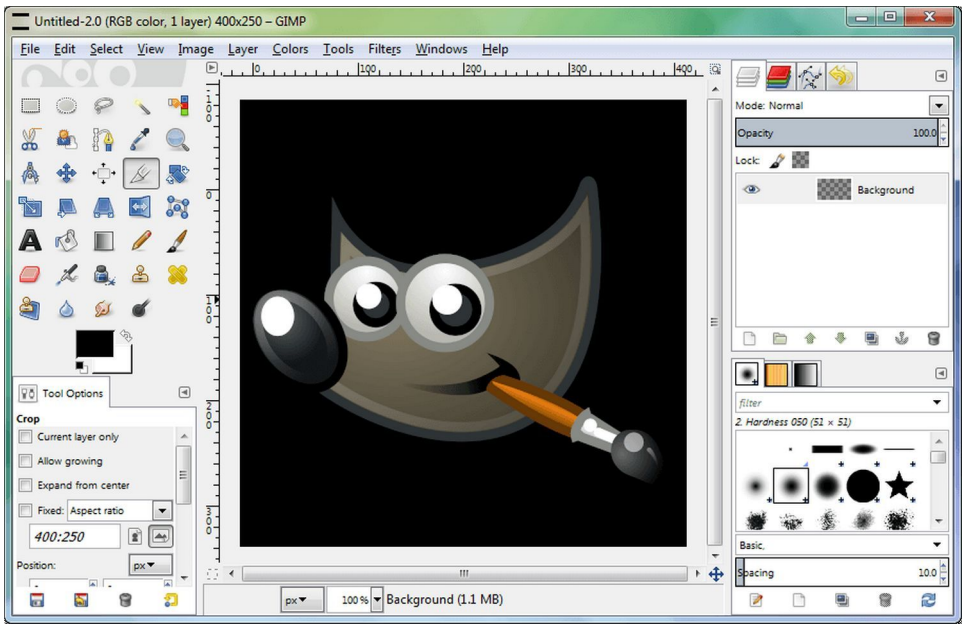
File comparison utility. Useful (in addition to text) to compare two images as binary files; in order to do so it is necessary to add ".jpg" to the file pattern for binary files, under
edit → options → compare → binary



3.4 GIMP

https://www.gimp.org/

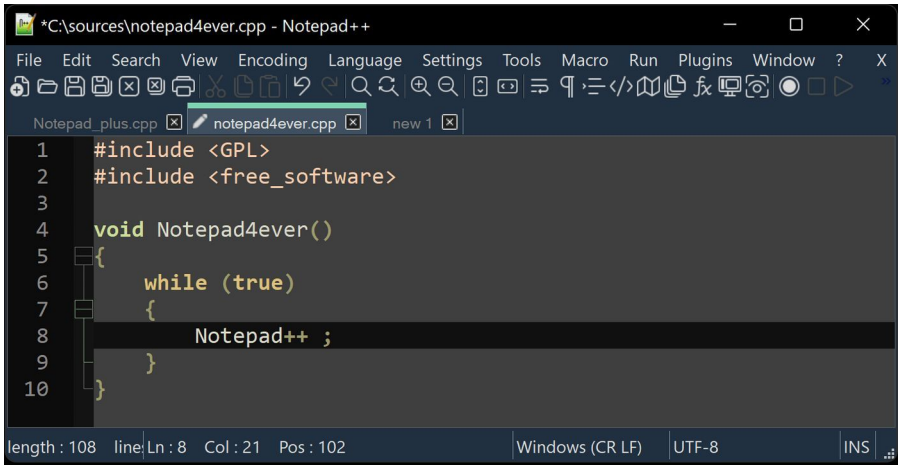
Image editing software; used to manually correct unrecoverable artifacts.



3.5 Notepad++

https://notepad-plus-plus.org/

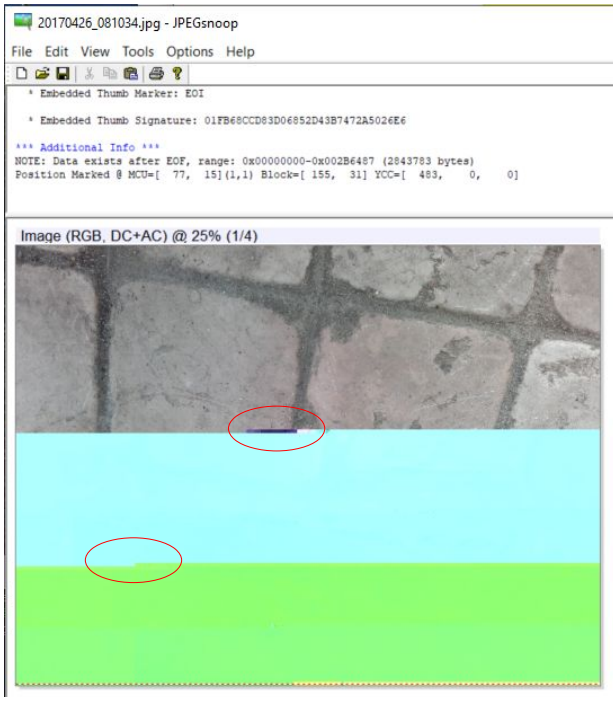
Text editing program with nice features.



4 Repairing an image

An example of repair flow is described here, applied to the following image:

- Load the image into JPEGsnoop and examine errors. There are likely two error locations:



  Scrolling the message window you can see a lot of info about the image and many messages reporting stream data errors (in red).

- Decompile the JPEG file into text

```
        >jpeg-decomp -decode -fin image.jpg -fout image.txt
Addr            Marker  Type
@0x0000         0xFFD8  SOI
@0x0002         0xFFE1  APP1 (4732 bytes)
@0x1280         0xFFE9  APP9 (4 bytes)
@0x1286         0xFFDB  DQT (132 bytes)
@0x130C         0xFFC4  DHT (418 bytes)
@0x14B0         0xFFC0  SOF0 (17 bytes)
@0x14C3         0xFFDA  SOS (12 bytes)
@0xF641B        0xFF80  ??
@0x16BE47       0xFF01  ??
@0x2B2C5D       0xFFFA  JPG10 (6305 bytes)
```

```
@0x2B6485       0xFFD9  EOI
Precision=8 2592x1944 3 components:
ID:1 [22] Dest:0
ID:2 [11] Dest:1
ID:3 [11] Dest:1
MCU: YYYYCC (16x16 pixel)
[162x122=19764 MCU]
HT standard @0x1310
found 19804 MCU (79216 Y + 39608 C)
```

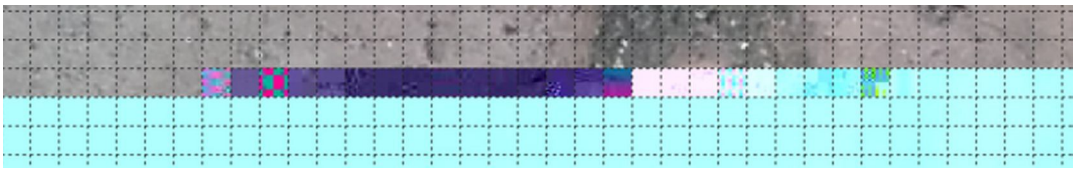
Notice the sections present at the beginning; you can read more about them in various online guides. After SOS (start of scan) there should only be one EOI (end of image) marker; instead there are three, 0xFF80, 0xFF01, 0xFFFA; this is the effect of data corruption.

19804 MCU blocks were found, instead of 19764; it means that some blocks were generated as a consequence of data corruption.

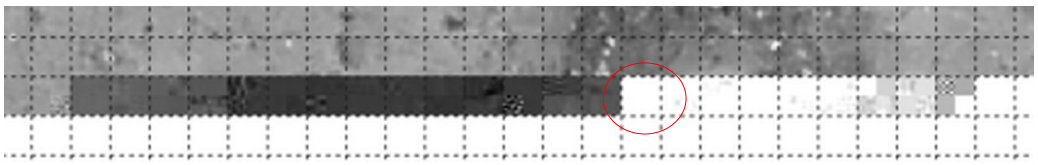
- Compile the text file back into JPEG; often this step is sufficient to reveal additional details, because the compiled file will be (more) JPEG compliant.

```
       >jpeg-compiler -encode -fin image.txt -fout image.1.jpg
1 raw segment
79216 y segments
39608 c segments
```

- Load image.1.jpg into JPEGsnoop; add the MCU grid (view → overlays → MCU grid) and zoom-in to the first discontinuity.



Notice that as the mouse cursor passes above the image, MCU coordinates are shown in the status bar. Now switch to luminance view (alt-6):



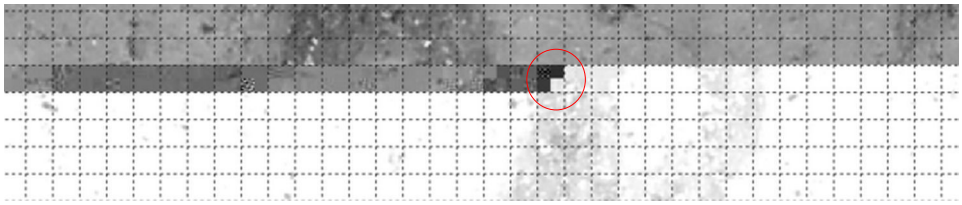
The first white MCU is (78,51); here a high DC value in a Y block is setting luminance to the upper limit. It has to be corrected in order to reveal more details of the image.

- Edit image.txt in order to decrease brightness; any text editor is fine (I use notepad++). Find MCU (78,51):

```
//************ MCU 8340 (78,51) (@0xF67D6.6):
<y>
//[Y@0xF67D6.6] DC:1444 AC: 3 2
1444 0b011101101010
</y>
<y>
//[Y@0xF67DA.6] DC:-18 AC: 0 -3 -10 1
-18 0b1101100101101010011010
</y>
...
```

Indeed we have a Y block with a DC level of 1444. Let's change it to 0 and see what happens.

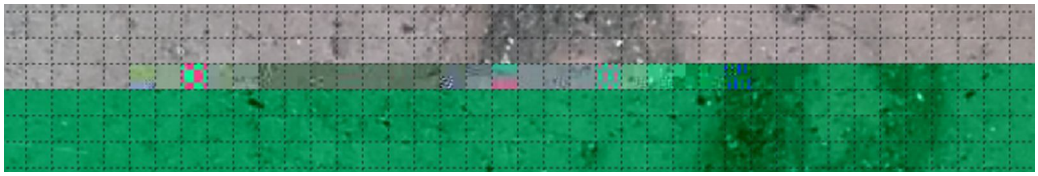
- Save image.txt, compile again (same command as before) and reload in JPEGsnoop (ctrl-r):



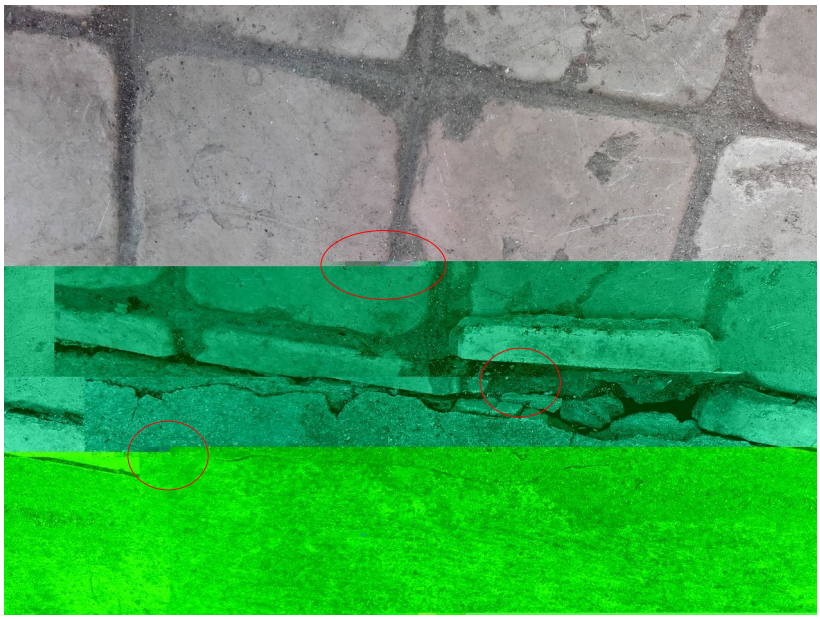
The situation is a little better, but there's another big shift at MCU (86,51):

<y>
//[Y@0xF6B03.0] DC:1417 AC: 3 -2 2 0 -2 -28 5 -2 12 0 -7 -11 -4 -5 1 6 1 3 5 4 -4 -4 -1 0 0 1 2 0 0 0 0 0 -2 -1 0 -1 0 0 0 -1 0 0 1 1
1417
0b0111010101101101101110100001110010101011011110011110010001011010010001110001000110011000101111001011001001000111000110
0011100101101111111011101000110001110100111001001110
</y>

- Let's put 0 again, recompile and refresh:



Much better, all details are visible. Examine the whole image with a normal viewer:



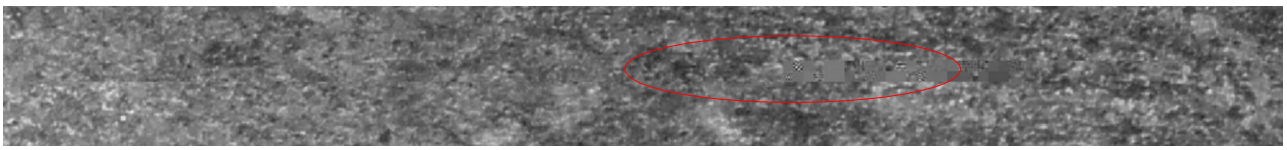
There are at least 3 discontinuities, plus the number of MCU is not correct; this causes alignment issues.

- Before fixing brightness and colors it is necessary to align the image by removing corrupted MCUs where they occur (or potentially adding empty ones, but it is rarely the case). That's because removing an MCU also affects the color/brightness of the following ones (DC coefficients are all linked together); so even if the colors were already fixed, the alignment process would cause additional shifts.

It is easier to proceed from the end backwards; in this way the coordinates in the text file

will be consistent even after removing MCU blocks.

Looking in detail, the first discontinuity from the bottom is at (64,105); it is barely visible in color; much better on luminance channel (alt-6) and without MCU grid:



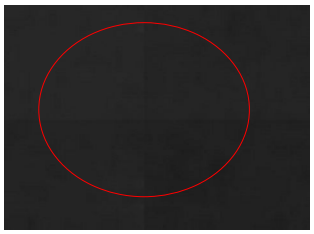
or Cb channel (alt-7):

The aim here is to align the rows before and after the error region; vertical features are a great help; in this case the only feature is the original image border.

- Remove a couple of MCU from image.txt starting from (64,105), then recompile and refresh in JPEGsnoop:

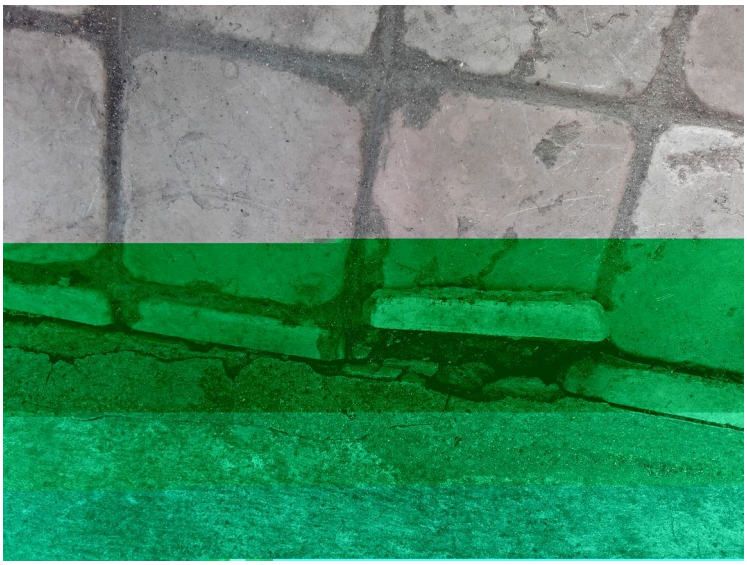
Still not aligned; after a couple of tries I had to remove 6 MCU:

```
...
//************* MCU 17074 (64,105) (@0x24443A.4):
//************* MCU 17075 (65,105) (@0x244484.2):
//************* MCU 17076 (66,105) (@0x2444E7.6):
//************* MCU 17077 (67,105) (@0x2444F5.0):
//************* MCU 17078 (68,105) (@0x244570.4):
//************* MCU 17079 (69,105) (@0x2445AA.7):
//************* MCU 17080 (70,105) (@0x2445F3.6):
<y>
//[Y@0x2445F3.6] DC:5 AC: -3 -36 13 -1 -2 -1 0 -2 0 -2 4 -1 1 1 0 -3 2 3 -3 -1 -3 0 1 -1 0 0 0 0 1 -2
5 0b0100111100001101110111101000010100011011011101101100100000001001110110001100111010000001001100100011101110101011010
</y>
...
```



Now it’s aligned.

- Repeating this sequence for all discontinuities required removing 5 MCU @(42,89), 6 @ (28,88), 6 @(108,73), 10 @(69,51). Here the result so far:



- Next is color/brightness correction; first it's necessary to re-calculate all MCU coordinates from the latest image:

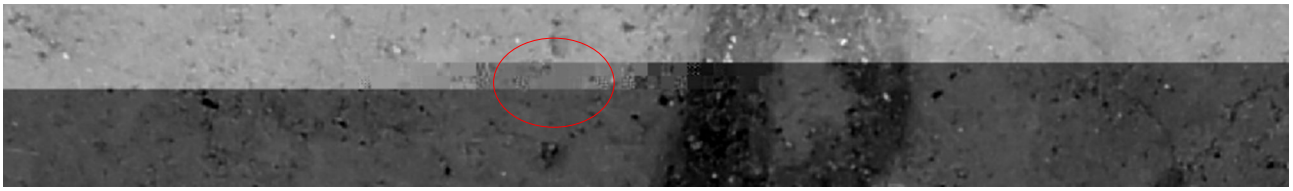
```
>jpeg-compiler -decode -fin image.1.jpg -fout image.1.txt
```

And generate a new image:

```
>jpeg-compiler -encode -fin image.1.txt -fout image.2.jpg
```

Now load it into JPEGsnoop; at this point we are sure that coordinates on the screen match with those in image.1.txt.

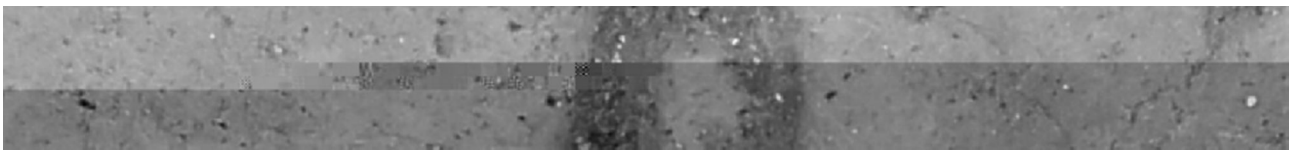
- Correct brightness; here is better to proceed from the top: switch to luminance view (alt-6) and zoom-in after the first discontinuity; the first coordinate with a noticeable shift is (74,51);



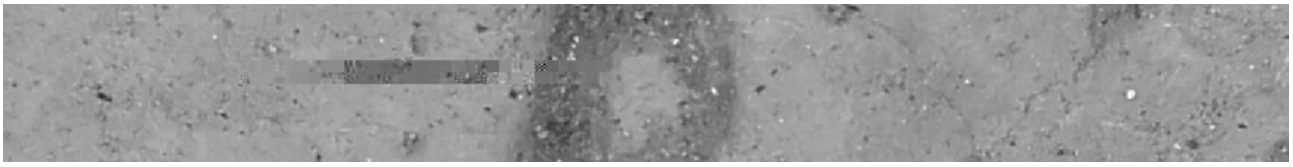
Let's find it in image.1.txt:

```
//************ MCU 8336 (74,51) (@0xF679E.5):
<y>
//[Y@0xF679E.5] DC:15 AC: -1
15 0b0001010
</y>
<y>
//[Y@0xF67A0.3] DC:-410 AC:
-410 0b1010
</y>
…
```

The level is clearly too low; let's try with 0. Save image.1.txt, generate image.2.jpg, refresh it in JPEGsnoop.



Better but not yet enough. A few trials are needed, but in the end it’s possible to find an optimal number, 280.



The eye is extremely sensitive to brightness so it’s possible to remove the discontinuity almost perfectly.

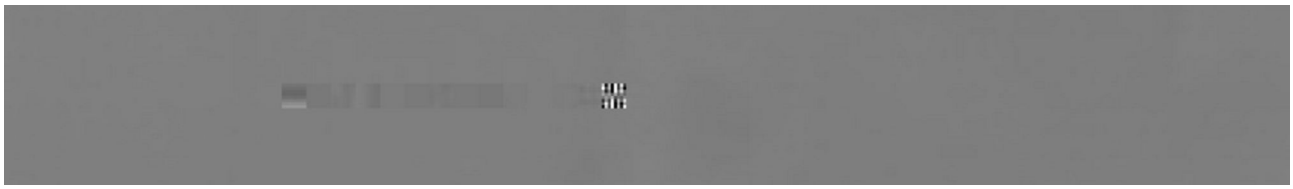
- Next channel, Cb (alt-7); notice how color details are usually less prominent:

Find MCU (72,51) in image.1.txt:

```
//************ MCU 8334 (72,51) (@0xF6689.4):
<y>
//[Y@0xF6689.4] DC:10 AC: -2 0 0 1 3 -1 -2 -6 8 -3 0 2 1 0 0 -1 -2 -4 0 0 1 0 0 0 2 0 0 0 0 1 0 -1 -1 0 0 0 0 0 -1 0 1 0 0 0 0 -1
10
0b0101111001011100001011000011011100001001101110001111000010110001111100111111011110111011111000000111101001100111101101
010
</y>
<y>
//[Y@0xF6699.4] DC:79 AC: 18 -10 3 -7 7 -5 5 -2 2 6 -4 -2 6 -5 -3 3 1 0 2 2 0 1 -4 2 0 1 0 0 0 0 0 -1
79
0b1101010010101101010111100000100111100010100101010101101001101000110101100110100010010000111001110111001101100110000110111
011001111101001010
</y>
<y>
//[Y@0xF66AC.0] DC:-65 AC: 68 -78 -28 52 5 -49 -1 31 24 22 -13 -9 -24 22 -4 24 12 -16 -27 2 1 -11 8 15 -4 -2 2 1 0 0 -3 3 2 0 -1 0 0 2 2 0 0 1 1 0 0
0 1 0 0 0 0 0 0 0 -1 0 0 0 0 -1
-65
0b1111100010001001111100001100011101000011111100011010010010111110000011100001101011111110101100011010101101011001010110
1101101000111110101011010001111010110001011110011010011111101000100011000110110100101110001011111110001101010110001111111
0010001110110110001111100110011011100100111101011111011011101101010
</y>
<y>
//[Y@0xF66D3.5] DC:-1 AC: 0 0 0 0 0 0 -1 -1 0 0 0 0 -1 2 0 3 -2 1 0 -1 1 -3 1 -18 2 -21 -40 -10 -7 -2 20 20 -6 -14 -25 14 -1 -5 -9 4 11 5 0 9 2 0 4 5 0
2 1 -4 -3 -2 5 2 0 0 0 0 0 1 0 0 1
-1
0b1111011000011101100110110111101010011100000101000011101001101011011010010101111000010111101101011000000101110101010011
0101010010000110110001110100011010111110000100010101101101001001011101110010111111011010010110111100110010010111011100001
1000110100010110010101101111010101111001
</y>
<c>
//[C@0xF66F6.5] DC:-85 AC: 0 1 1 5 0 0 0 0 0 0 0 1
-85 0b10111011101010111111010100
</c>
<c>
//[C@0xF66FB.4] DC:3 AC: 8 0 0 0 0 0 0 0 -1 0 0 0 0 -3 -1 0 0 0 0 0 0 -1
```

3 0b1100010001111010011111011000010111100100 0
</c>

Cb is the first C block in that MCU; -85 causes a drop in Cb level, as expected. Let’s proceed in the same way as for luminance: try with 0, save image.1.txt, generate image.2.jpg, refresh it in JPEGsnoop. Repeat until there is no more discontinuity:

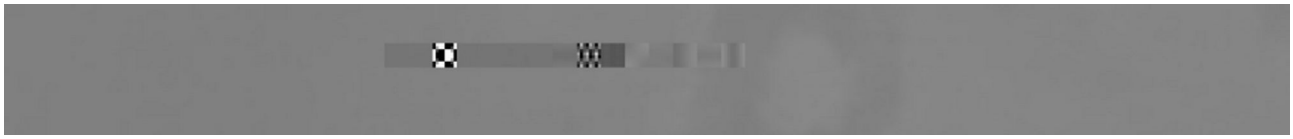


- Next channel, Cr (alt-8):

Change happens in MCU (73,51), Cr is the second C block:

...
<c>
//[C@0xF679A.1] DC:-228 AC: 0 0 1 -37
-228 0b11010111100001101000
</c>
...

Same procedure, optimized value is 350:



- After equalizing Y, Cb, Cr, the first discontinuity is removed:

A few corrupted blocks are still present, but they don’t affect other areas any more.

- The steps above have to be repeated for every discontinuity; this is the result:

The remaining artifacts can be corrected with an image editing tool, like GIMP, using a clone tool or similar.

5 Combining multiple versions of the same image

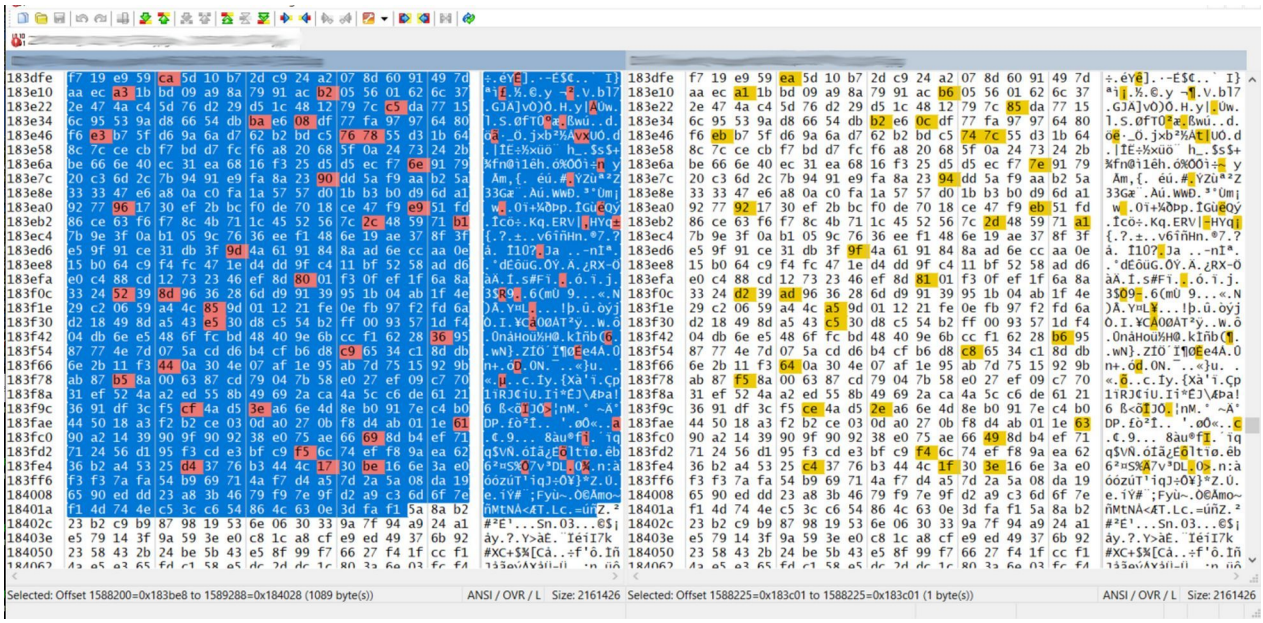
When dealing with corrupted media (e.g. SD cards) it is often useful to copy the damaged image multiple times using different methods like file manager, various data recovery software, different hardware in case of removable media.

Hopefully, those copies will show artifacts in different areas; in this case, a byte by byte copy can generate an image that combines the cleanest area of all images.

Load both images in separate JPEGsnoop instances, and in WinMerge as well (remember to specify a hex comparison view).

Move the cursor to the first region with artifacts and note the file address.

Go to the address in WinMerge (alt-up/down moves across differences); from the pane containing the source image (i.e. the one without artifacts in that position) select a region that completely overlaps all differences; copy to the other image (alt-left/right); save and examine the result: the destination should look exactly as the source in the region “repaired”.



The same procedure can be applied to all artifacts and repeated with each couple of images.

6 Repairing images with restart markers

Images that use restart markers are more resilient to artifacts, but it happens that some codecs cannot recover from extensive errors; images appear truncated or with misaligned parts.

In these cases it is often sufficient to pass through the intermediate textual representation to fix it; first decompile the image into text:

```
>jpeg-decomp -decode -fin image.jpg -fout image.txt
```

Try a simple reconstruction:

```
>jpeg-decomp -encode -fout image.1.jpg -fin image.txt
```

If the result is not satisfying, you can try correcting restart marker errors; the decoding step gives some useful information:

```
…
MCU: YYCC (16x8 pixel)
[100x150=15000 MCU]
Restart interval: 4
found 15218 MCU (30420 Y + 30335 C)
Missing restart markers
L       #
```

```
1       43
2       26
3       10
4       3
5       3
6       1
7       4
8       4
9       3
10      2
13      1
```

Note the restart interval N (=4), and the missing markers table; every row will tell how many times there was a series of MCU that extended past the restart interval. Here for example, the string of 7 MCU (N+3) without a restart marker happened 10 times.

Restart errors in the text file can be of several types:

- Interval errors:

  //Restart interval error (+1: 5 MCU instead of 4)

  this MCU was past the expected interval between restart markers; most often data corruption creates a number of extra MCU that have wrong pixel data. The length of this extra MCU chain is measured and listed in a table as seen above. Image decoders ignore extra MCU although it can happen that there are some real MCU within false ones; it's very difficult to determine which are correct.

- Incorrect restart number:

  <restart>4</restart>
  //Restart marker # error (4 instead of 0)
  it means that the last restart was #7 (scroll up in the file to check), so it was expected #0 instead of #4; it is possible that there was a bit flip and 0 became 4, or it could be that previous markers (#0-1-2-3) were deleted by data corruption; in this case there will be more than N blocks since the last restart (#7). Try to insert restart tags at the appropriate place or change the number of existing ones in order to obtain a complete sequence (0 to 7, then 0 again).

- Missing component:

  //MCU error: missing component (2 instead of 4)
  it happens when a restart marker appears before an MCU is finished; for example an MCU defined as YYCC but with only two Y blocks. Can be left unchanged, as the fix of adding empty blocks is exactly what the image decoder already does.

After fixes to the text file, encode it into a new image. It may happen that the result is nearly equivalent to the first trial; that's because the information around restart marker errors is likely corrupted, but image decoders already do a good job of finding the correct geometry even when restart markers are missing or incorrect.

Note that JPEGsnoop does not perform any restart marker correction, so it presents all blocks that a normal decoder would hide; images appear misaligned until extra blocks are eliminated.

7 Random questions

Is automatic error correction possible?

It is very difficult to replicate the capabilities of the eye/brain to detect regions separated by slightly different colors or small misalignments; also the appropriate recovery strategy may vary between images.

In short, a human is best suited to do this activity.

Is it possible to automatically adjust the DC value of image blocks?

It could be done in a graphical tool; the user would select a boundary and a target MCU. An algorithm would weigh each channel on both sides of the boundary and apply a corresponding correction to the target block.

Is there a graphical tool that does the same things?

There is a non-free tool that is supposed to correct images by allowing block editing; I never used it so I don’t know how easy it is to use and what features it has.

Designing a free tool is of course possible; maybe in the future.