

Gregory James Gosian
526 Thornell Rd.
Pittsford, NY 14534

A Non-Probabilistic, Compact Compression Algorithm Suitable for Deep Space Solar System Mission Image Transmission.

Abstract

Image acquisition on space missions is a memory intensive activity, due to high-resolution focal planes and hyperspectral data collection. Data compression is required to accommodate on-board data storage and data transmission rate limitations due to distance from Earth. Current algorithms employed on spacecraft are probabilistic and require the processing of entire images to achieve high compression ratios, either in frequency or wavelet domains.

Twenty-five test images from prior space missions were subjected to a newly developed compact, non-memory intensive, non-probabilistic, lossy compression algorithm, and achieved PSNR's of approximately 30 dB. On pixel gray-level step-discontinuities, the algorithm achieved tracking of the original signal within 2 digital counts for all discontinuities from 0 to 255 digital counts, after only 6 pixels. The algorithm (written in IDL) requires less than 1 kilobyte of memory, including workspace for computation. It can function as a stand-alone compressor and achieves approximately a 2.7: 1 compression ratio for 8 bit images, and 5.3: 1 for 16 bit images. It can function as a pre-processor when paired with a lossless encoder (e.g., Huffman, Arithmetic) for further compression.

Nomenclature

DC = digital count
 f_{ij} = pixel DC in original image
 \hat{f}_{ij} = pixel DC in decompressed image
 N_c = number of pixel columns
 N_r = number of pixel rows
 m = bits per pixel in original image
 n = bits per pixel encoded

Introduction

Interplanetary space probes have been launched to all points in the solar system since late 1959 when the Union of Soviet Socialist Republic's Luna 3 took the first photo of the far side of the moon. Though doing so in the proximity of their targets, these probes still are essentially remote sensing missions. As such, transmission of imagery back to Earth is one of the primary means of relaying data to the Mission Science Teams. But data transmission over such long distances remains problematic. Greater distance from the Earth demands slower bit rates in transmission to minimize errors. Any way to simplify or shorten the data stream enhances the probe's performance, and allows broader mission objectives to be achieved over the lifetime of the spacecraft. Image compression is one such method to shorten the bit stream.

Since the advent of digital image acquisition, compression algorithms such as Huffman encoders, Integer Cosine Transforms (a variant of JPEG compression), Integer Wavelet

Transforms (a variant of JPEG 2000 compression), Arithmetic Encoders, or combinations thereof, have been employed.

A family of newly developed lossy compression algorithms is presented that will prove useful in speeding initial image transmission, to facilitate selection of images and places of interest to be further examined with higher resolution and alternative longer data stream/lossless compression algorithms. These compression algorithms also have the advantage of being non-probabilistic, non-computational and non-memory intensive.

Background and Motivation

Image compression doesn't just enable faster image downloads to Earth, in one instance it was called upon to save an entire mission: NASA's Galileo Project. Galileo was designed to place an orbiter around Jupiter to study the planet, and predominantly its four major moons, Io, Calisto, Ganymede and Europa. The spacecraft as launched was conceived of in the late 1970's, and built with early 1980's technology. It was originally scheduled to be launched with a cryogenic high-energy Shuttle Centaur upper-stage, from the Space Shuttle cargo bay. The launch failure of the Shuttle Challenger and the subsequent grounding of the entire Space Shuttle fleet delayed Galileo's launch an additional three years. It's a very time consuming, money consuming problem to unpack a payload that's been configured for launch, so Galileo sat in storage with its main antenna folded away for that entire wait period on the ground. When finally launched from Shuttle Atlantis's cargo bay, and after well on its way to Jupiter, the high-gain antenna was commanded to deploy but did not do so properly, and as such was useless. The high gain antenna was to have supported data transmission speeds of up to 134 kilobits/second. This would have allowed an 800 x 800 pixel, 8 bits/pixel image to transmit to earth in just under one minute [1].

The only alternative was to utilize the low gain antenna, which had originally been designed to send back numerical data on spacecraft subsystems. The low-gain antenna had an original data rate of 16 bits/second. For an 800 x 800 pixel, 8 bits/pixel image, the required data stream was 5120000 bits and at 16 bits/second, a single image would require 89 hours to transmit.

The first subsequent fix was to send up new software that enabled the data stream rate of the low-gain antenna to be increased to 160 bits/second, but this still required 9 hours per image. Next, Galileo's ICT image compressor (which allowed compression rates of up to 50:1 for images that could withstand such high compression rates) was utilized to bring that number down further to about 10 minutes of transmission time for each image. In actual practice, most images were compressed such that it took one to two hours to transmit. The mission proceeded with images acquired being stored using the on-board magnetic tape system. Magnetic tape offered superior performance with respect to the density of bits per unit pound/per unit volume, as well as better protection against radiation induced data loss, than did the solid state memory systems of the day. These stored images were then transmitted during the cruise phases as Galileo transited between the target moons.

Over time, the magnetic tape system's performance degraded, as mechanical systems are prone to do. Even though subsequent missions are utilizing solid state memory systems with no moving parts, the potential for loss of a memory storage unit or of computational ability remains a real possibility. Weight considerations imply that no space probe can carry sufficient redundancy to compensate for all major malfunctions.

It is precisely this example that prompted the investigation into a non-probabilistic compression algorithm that would be robust, maintain high peak signal to noise ratio (PSNR), and would minimize on-board storage needs, data stream and transmission time.

Algorithm Description

This compression algorithm is designed to reduce the transmission of an m -bit word to describe digital counts for a panchromatic image, to a fixed n -bit code word. For an m of 8, and an n of 3, a reduction of 62.5% of the data would be achieved, yet it is required that a peak-signal-to-noise-ratio (PSNR) of at least approximately 30 decibels (dB) be maintained.

Any algorithm created could be a lossy type, as long as it reproduced image features that would remain recognizable by an astronomer, without creating artifacts that would detract from the true scientific information content.

The algorithm chosen is a differencing schema – information will be transmitted to additively modify the preceding pixel's digital count, to arrive at the current pixel's digital count in the decompression of the image.

The transmitted information is series of n -bit words. One bit determines the sign: 0 indicates a minus, 1 indicates a plus. The remaining bits describe the scalar values of the difference. As such, here are the details for the 8-bit to 3-bit algorithm:

- A scalar difference DC of 1 is dismissed as not required, since adjacent constant value pixels can be “dithered” up by a DC of 2, and down by a DC of 2 and the human visual system (HVS) will smooth out the contrast over this short spatial distance. For a constant value DC, the algorithm oscillates in a very narrow digital count range around any particular DC value.
- A scalar difference DC of 256 is dismissed. An image of the white limb of a planet against a black background is typically the only locus in an image requiring so great a step. For all practical purposes, this can be accomplished (as well as for *any* such higher difference between adjacent pixels) with one step of 128, followed by whatever successive value is required and available in the codebook.
- Selected are 128 at one end of the scale for difference, and 2 at the other. This consumes one bit with two states (0 or 1):
- Another bit of the 3-bit code word is assigned to the values of 32 and 8 – so as to equally space out the assignment of bit states. The values selected are as such:
 - Binary:
 - 128 – 10000000
 - 32 – 00100000
 - 8 – 00001000
 - 2 – 00000010
 - These four values are encoded among 2 bits.
- As this is a differencing algorithm, and it will add or subtract, a third bit is required to carry sign. One state to represent a “+”, the other state to represent a “-.”

The 3-bits are encoded into words according to the codebook in Table 1.

Value (Delta)	-2	-8	-32	-128	+2	+8	+32	+128
Encoded Decimal	0	1	2	3	4	5	6	7
Encoded Binary	000	001	010	011	100	101	110	111

Table 1 – 3-bit encoding codebook

The binary representation of a 256 gray-level image can be represented by an 8-bit word. The values of the differences between adjacent pixels ranges from 0 to 255, also represented by an 8-bit word. Consider the 8-bit word to be a bank of 8 “switches.” This algorithm will only allow one of those 8 bits to be “switched” at a time, and will allow only every other switch as a selection.

Algorithm Methodology

This compression algorithm for 8-bit to 3-bit encoding utilizes the following steps:

1. The first pixel in the row is assigned an encoded DC value of **127** (regardless of what the actual value is), for the encoding and decoding stages. The value is irrelevant as long as the transmitter and receiver agree on the starting value.
2. For the next and each successive pixel in the sampled image, the residual difference between the original image's $n+1^{\text{th}}$ pixel's DC value and the encoded image's n^{th} pixel's DC value is calculated, according to the following formula:

$$\text{Residual difference} = DC(\text{original})_{n+1} - DC(\text{encoded})_n$$

3. A determination is made as to which of the code book values available is closest to the residual difference, and can be added to the n^{th} encoded image pixel's DC to bring it as close as possible to the original image's $n+1^{\text{th}}$ pixel's DC value:

$$DC(\text{encoded})_{n+1} = DC(\text{encoded})_n + \text{codebook value}_n$$

4. The encoded word representing the added value is transmitted in binary according to Table 1 to the receiver.
5. Steps 2 through 4 are repeated for each pixel across every pixel row of the image.
6. Steps 1 through 5 are repeated for all rows of the image

Here is an example for one particular row in an image:

ENCODING and TRANSMITTING

Pixel	0	1	2	3	4	5	6	7	8	9	10	11
Original Image DC	180	180	190	189	189	188	160	22	21	18	18	19
Encoded Image DC	127	159	191	189	187	189	157	29	21	19	17	19
Subtract Encoded Image DC from Original Image DC	53	21	-1	0	2	-1	3	-2	0	-1	1	0
Closest code book value	+32	+32	-2	-2	+2	-32	-128	-8	-2	-2	+2	-2
Encoded image DC plus code book value (becomes $n+1^{\text{th}}$ Encoded Image DC)	159	191	189	187	189	157	29	21	19	17	19	17
Transmitted word (coded codebook value)	110	110	000	000	100	010	011	001	000	000	100	100
Difference (between Encoded Image DC and Original Image DC)	21	1	0	2	1	3	7	0	1	1	0	2

Table 2 – Encoding scheme

KEY -



Move sum of Decompressed Image DC and closest Codebook value up to this cell.

DECODING and DECOMPRESSION

The DECOMPRESSION algorithm is applied as follows for the first pixel row:



1. The receiver sets the first pixel (in each row) DC count in the row to be decoded to 127.
2. The receiver adds the value of the decoded binary word to this first pixel's DC value, and makes this the next pixel's DC value.

$$DC(decoded)_{n+1} = DC(decoded)_n + codebook\ value_n$$

3. The receiver continues to add the value determined by each successively received encoded binary word received to the n^{th} pixel DC in his image, to reconstruct the $n+1^{th}$ pixel DC.
4. Steps 1 through 3 are repeated for the remaining pixels in the row.
5. Steps 1 through 4 are repeated for all rows.

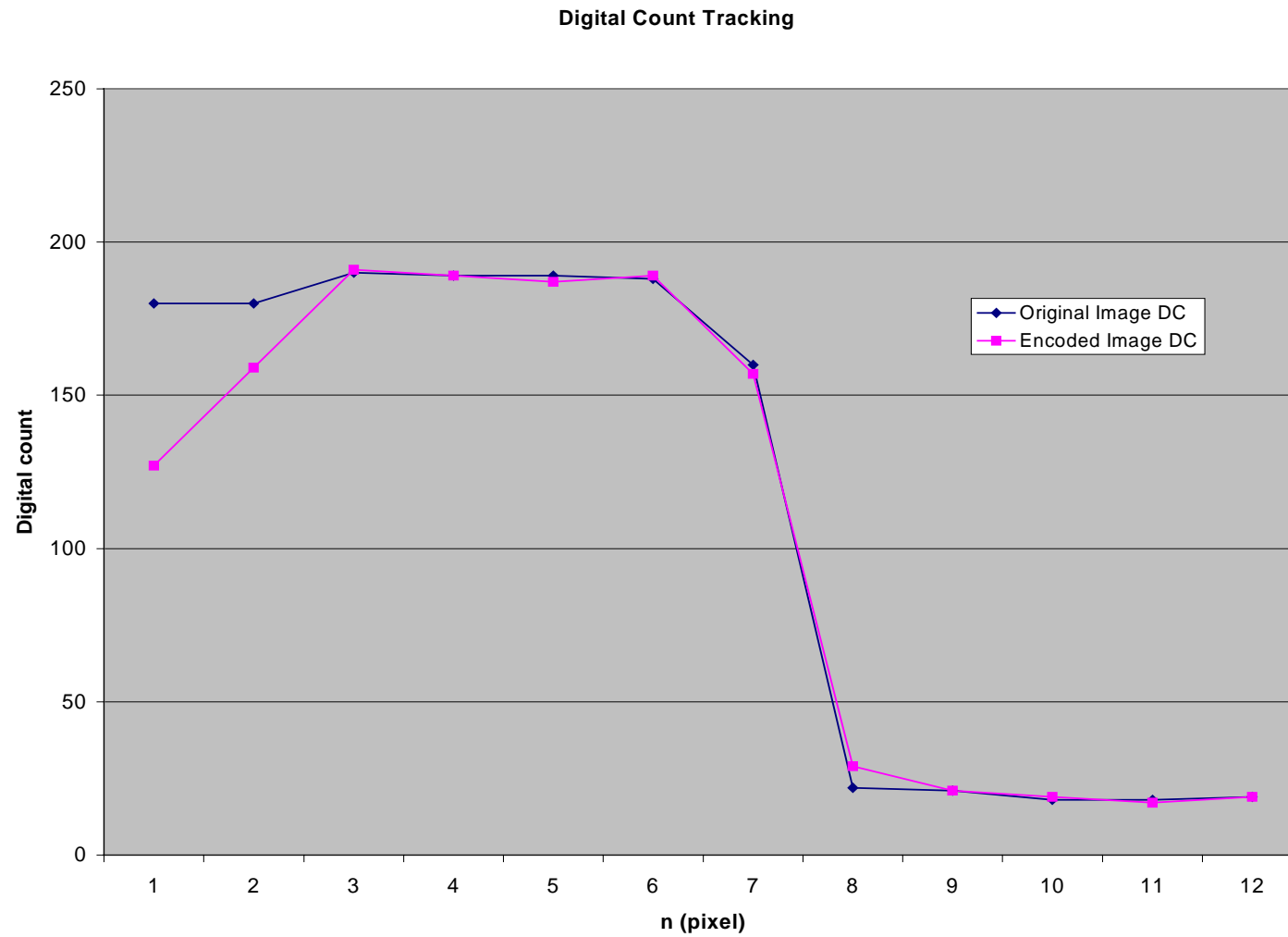
Pixel	0	1	2	3	4	5	6	7	8	9	10	11
Received word		110	110	000	000	100	010	011	001	000	000	100
Decoded value		+32	-32	-2	-2	+2	-32	128	-8	2	-2	+2
Decompressed Image DC	127	159	191	189	187	189	157	29	21	19	17	19
Original Image DC	180	180	190	189	189	188	160	22	21	18	18	19
Difference between Original Image DC and Decompressed Image DC	53	21	1	0	2	1	3	7	0	1	1	0

Table 3 – Decoding scheme

KEY -  Add two values (at head and tail).
 Put sum of n^{th} pixel DC and $n+1^{th}$ pixel decoded value here.

The actual DC level, and the encoded tracking DC levels for this example are shown in Figure 1:

Figure 1 – Digital Count tracking



Analysis

PEAK SIGNAL to NOISE RATIO (PSNR)

In order to evaluate this algorithm, 25 gray scale images (available in Appendix) of selected NASA space missions (Galileo, Shoemaker-NEAR, Voyager, Mars Reconnaissance Orbiter, Mars Exploration Rovers A and B) were subjected to the described compression algorithm. An IDL implementation is available in the Appendix at the end of this document.

This code was modified for 4 bit compression using the codebook described in Table 4, and also applied to the 25 images, to see if any significant improvement in the algorithm's performance was gained as measured by the Peak Signal to Noise Ratio (PSNR) values.

Value	-2	-4	-8	-16	-32	-64	-128	-235	2	4	8	16	32	64	128	235
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Table 4 – 4-bit encoding codebook

The PSNR values of the encoded images were computed using the original bitmap images as the reference, and were calculated (Equations 1 and 2). They are presented in Table 5. The same data is presented graphically in Figure 2:

$$\text{Equation 1} \quad PSNR (dB) = 10 \cdot \log \frac{(2^n - 1)^2}{MSE}$$

$$\text{Equation 2} \quad MSE = \frac{\sum_{i=1}^{N_R} \sum_{j=1}^{N_C} (f_{ij} - \hat{f}_{ij})^2}{N_R N_C}$$

For this particular set of images, where $m=8$ and $n=3$, a 62.5% reduction in information content yields an average PSNR of 31.79dB [25.30dB, 39.02]. Where $m=8$ and $n=4$, utilizing the additional bit in the codebook achieves a 50% reduction in information content, and yields an average PSNR of 33.58dB [25.69, 39.66].

The 4-bit compression performs an average of 5.8% better in terms of PSNR over the 3-bit compression. However, both algorithms yield PSNRs in excess of 30dB for nearly all images. The plot in Figure 2 shows that the PSNR values for both algorithms track each other well, with little difference in visual performance for any particular image. In no case does the 3-bit algorithm perform better than the 4-bit version.

Sample in presentation	3 bit PSNR (dB)	4 bit PSNR (dB)	% difference
1	30.614	33.019	7.856
2	35.843	37.825	5.531
3	25.296	29.689	17.364
4	31.214	32.894	5.383
5	33.396	34.072	2.025
6	30.123	31.703	5.245
7	31.018	33.532	8.107
8	29.702	31.351	5.553
9	32.601	34.018	4.346
10	39.016	39.655	1.636
11	31.867	33.645	5.579
12	29.931	30.276	1.153
13	32.558	34.070	4.642
14	34.687	36.596	5.502
15	27.704	31.185	12.565
16	31.546	32.180	2.009
17	32.465	33.687	3.763
18	30.213	31.669	4.818
19	30.188	33.388	10.599
20	33.903	34.955	3.104
21	27.660	28.660	3.616
22	31.275	33.262	6.353
23	34.627	36.817	6.325
24	33.775	36.000	6.586
25	33.448	35.359	5.715
Average PSNR	31.787	33.580	
	Average % difference		5.815

Table 5 – 3-bit and 4-bit encoded PSNRs

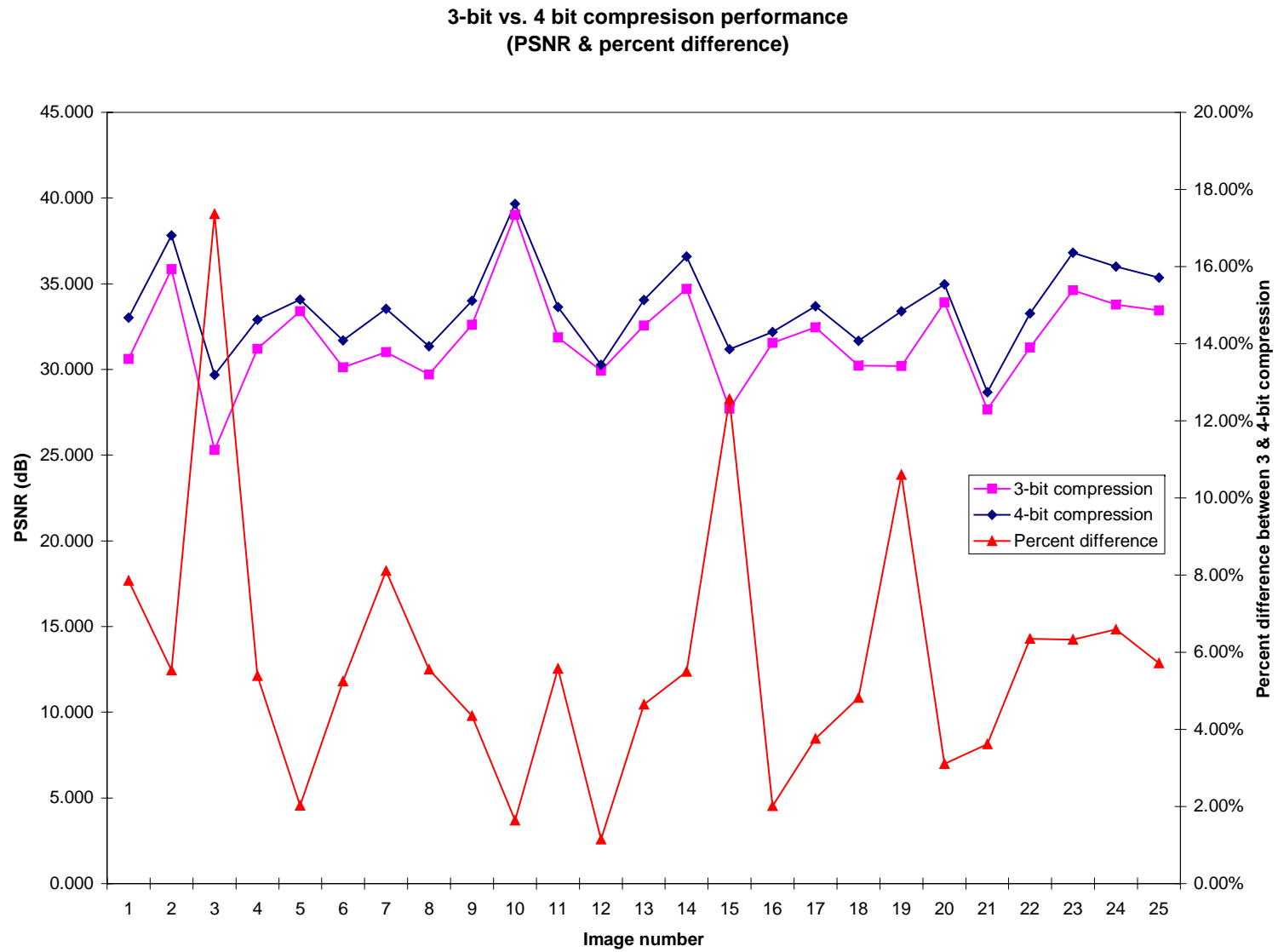


Figure 2 – PSNRs of images utilizing 3-bit and 4-bit version of compression algorithm

SIDE LOBE to PEAK RATIO

One of the qualitative checks created during the development of the algorithm was the creation of a difference image between the original image and decompressed encoded image. It was created by taking the decompressed image and subtracting the original image from it, on a pixel DC by pixel DC basis. Figure 3 illustrates several histogram exemplars for these difference images:

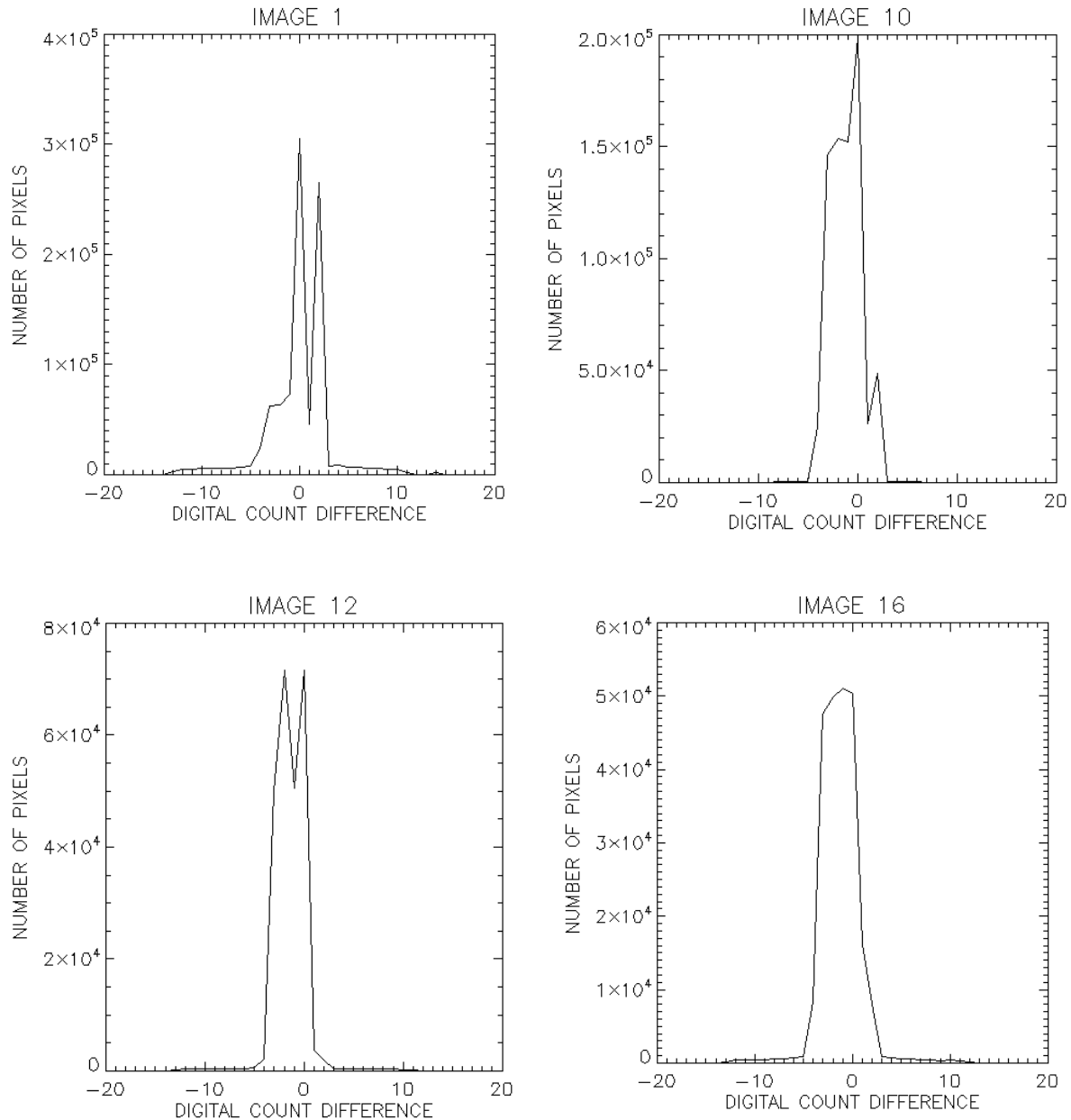


Figure 3 – Histograms of the differences of Original and Encoded Images. L-R, top to bottom: Image 1, Image 10, Image 12, Image 16

Of interest is what the distribution of the ratio side lobe to peak ratios for all difference image histograms, and the rate at which this ratio falls to zero. Ideally, each pixel should numerically match between images. So the peak of the histogram curves should be at 0. One could ask how much that ratio drops if 1 digital count to either side of 0; 2 digital counts on either side of 0, or 3, etc., can be determined.

Of the 25 images, the plots of the ratios are shown in Figure 4.

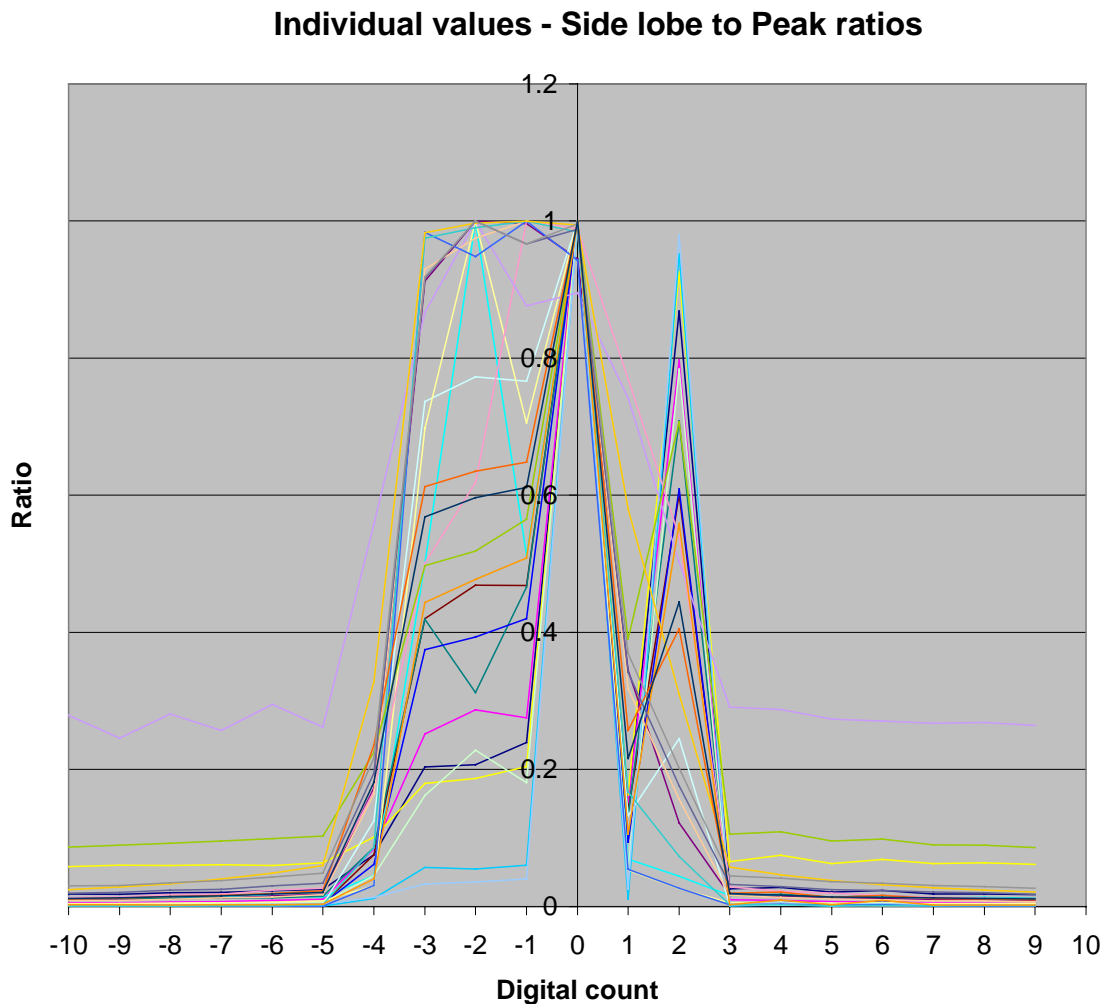


Figure 4 – Individual Side lobe to Peak ratios

Several features are noteworthy. First is the expected peak at a digital count of 0. This represents the pixels where DC values matched up and a difference of zero was obtained.

A strong peak at +2 was also observed. For images with a lot of black content (not at all unexpected), a DC value of 0, 1 or 2 would be expected for the black. The encoding would have tried to match the 0, 1, or 2 as closely as possible with a DC level of 1. Since the starting pixel DC in any row is 127, and all codewords are even numbered, then the lowest DC for any encoded image that could be achieved would be 1. In order to represent extended pixel runs of 0, 1 or 2,

the encoding would have added DC values of 2, and subsequently subtracted DC values of 2. So in order represent black scene content, the encoded image values would have values oscillating between 1 and 3. This leads to the strong peaks at 0 and +2.

Similarly for saturated bright areas, no value above 255 would have been encoded, so a peak at DC values of 255 could be achieved. The algorithm would create runs of encoded DC values oscillating between 253 and 255. The fact is that for any particular DC value in the original image that is constant over a long run of pixels, the algorithm has to accommodate the constant DC value by adding and subtracting encoded DC values of 2. This accounts for the strong peaks at plus and minus 2.

If the curves in Figure 4 were averaged, we would obtain curve shown in Figure 5 :

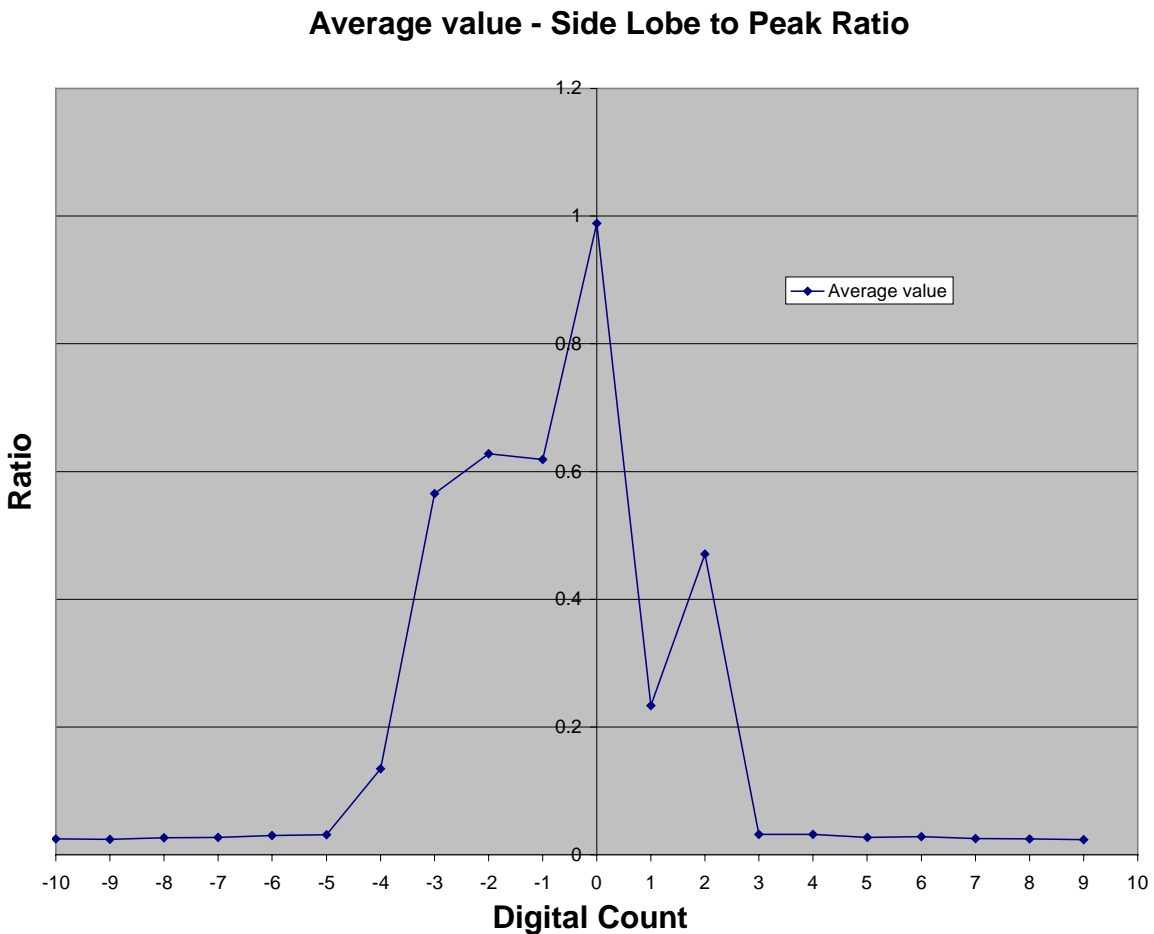


Figure 5 – Average Side lobe to Peak ratio

The item of note in this plot is that at plus or minus 5 digital counts, the value of the ratio drops to less than .032. It would appear that most of the pixels cluster within plus or minus 5 digital counts of where they should be valued. Even so, most lie within 3 digital counts of their true value.

PSNR and IMAGE ORIENTATION

The algorithm as written into code relies on scanning an image, row by row, from left to right. As such, it would appear intuitive that the code favors images that have horizontally continuous structure.

To investigate whether this is true, two images were fabricated. Each 512 x 512 pixel image was composed of alternating black (digital count=0) and white (digital count=255) lines, 4 pixels wide. One image had the stripes oriented vertically, one horizontally, and each was subjected to the algorithm. The images, their PSNRs after encoding, and the histograms of the difference of the encoded and original images are shown in Figures 6a and 6b. The PSNRs seem to acknowledge that the algorithm favors images with horizontal structure

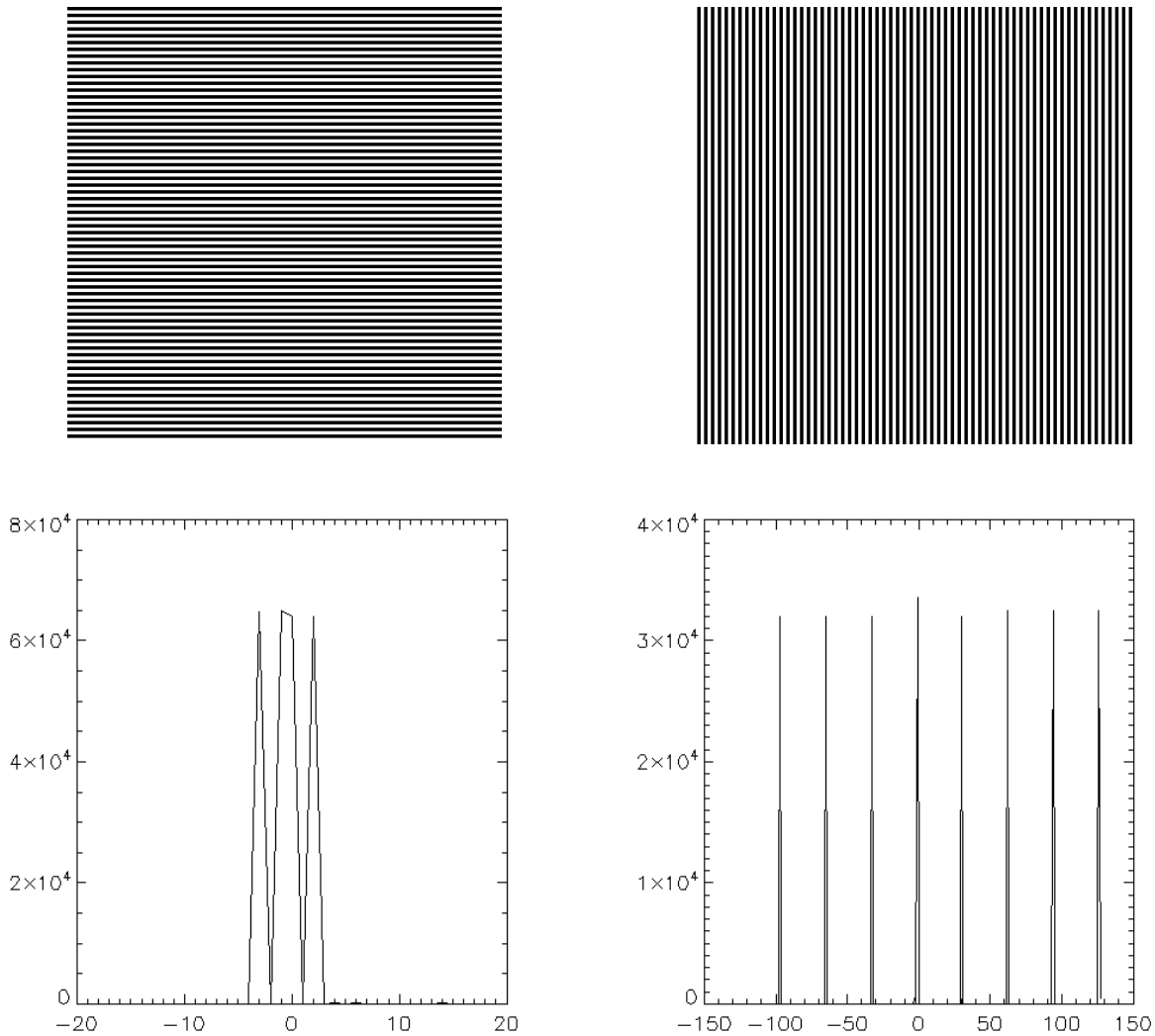


Figure 6a Horizontal Strip Images
PSNR = 27.628

Figure 6b— Vertical Strip Image
PSNR = 10.718

The histograms for the difference image created in Figure 5A is easily considered. In any row of pixels, the stripe's encoded values are quickly stabilized at or around the target DC values of either 0 or 255. The white stripe is encoded with DC values of 255 and 253, hence the difference values at 0 and 2 (in this case, +2). The black stripes are encoded as 1 and 3, hence the difference values of 1 and 3 (in this case, -1 and -3).

The histogram for the vertical stripes image case is more interesting. The image was created with a bright white stripe first (digital count value 255). So encoded, the decoded reassembled image starts with a default first pixel-in-any-row value of 127. The encoder adds a +128 code word and arrives at +255. The algorithm then dithers a few times between 253 and 255. The algorithm and resultant data for the histogram takes the form shown in Table 6. The same values, spaced at integral values of 32, appear almost uniformly (with the exception of the start and end pixels. The encoding for any row of pixels follows the pattern shown in Table 6.

Pixel	Original Image DC count	Encoded Image DC count	Difference
0	255	127	-128
1	255	255	0
2	255	253	-2
3	255	255	0
4	0	127	127
5	0	95	95
6	0	63	63
7	0	31	31
8	255	159	-96
9	255	191	-64
10	255	223	-32
11	255	255	0
12	0	127	127
13	0	95	95
14	0	63	63
15	0	31	31
16	255	159	-96
17	255	191	-64
18	255	223	-32
19	255	255	0
20	0	127	127
21	0	95	95
22	0	63	63
23	0	31	31
24	255	159	-96
25	255	191	-64
26	255	223	-32
27	255	255	0
28	0	127	127
29	0	95	95
30	0	63	63
31	0	31	31
32	255	159	-96
33	255	191	-64
34	255	223	-32
35	255	255	0
...

Table 6 – Vertical Stripe Image Histogram Pattern Data

Further confirmation was obtained with a real image as shown in Figure 7.

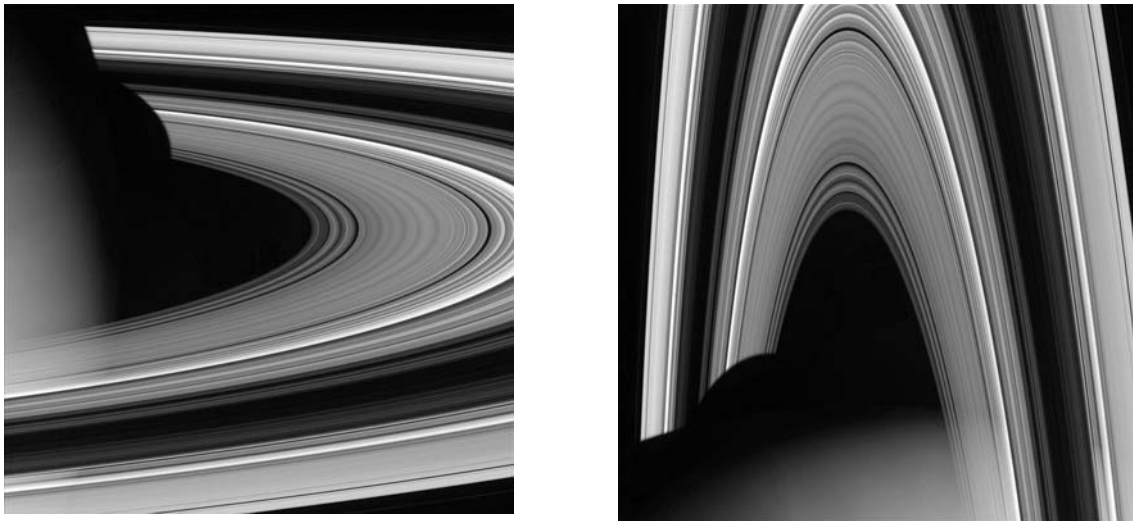


Figure 7 – PSNRs = 33.400(L) and 28.495(R)

The PSNR for the image of Saturn on the left was higher than the one for the image on the right; 33.400 versus 28.495, confirming that images with horizontal structure fare somewhat better than images with vertical structure. This suggests that flight code, if being used as a pre-processor, might have an element incorporated that determine the best orientation prior to compression. Of course, such a pre-processing element would need to be performed probabilistically, and would require computation and memory, which defeats the non-probabilistic nature of this algorithm. Either the image will have to be compressed without regard to the orientation of the majority of horizontal structure, or in the event of memory or computation failure the pre-processor would have to be removed from the image chain.

STEP DISCONTINUITY

Another aspect of performance that warrants investigation is the ability of the algorithm to track a discontinuity in DC value in one row of pixels. In this case, the interest is in parameterizing the algorithm's ability to replicate the discontinuity that occurs when one DC value that has been maintained for a consecutive number of pixels, is changed to another DC value over a one pixel increment, to a new value that will be maintained for another number of consecutive pixels. The determination to be made is how many successive applications of the algorithm are required for the encoded DC values to be within 2 digital counts of the desired value.

For example, suppose in one row of pixels the DC value of 1 has been maintained for a long run of pixels. Then a jump to a DC value of 190 occurs (which, similarly, will be maintained for a number of successive pixels). This might represent the value of the black background of space, encountering the bright limb of a planet, in one row of pixels of an image. The difference between the two values is a DC of 189. Can the question of how many successive pixel encodings are required, to bring the encoded DC value to within 2 digital counts of 190, be calculated?

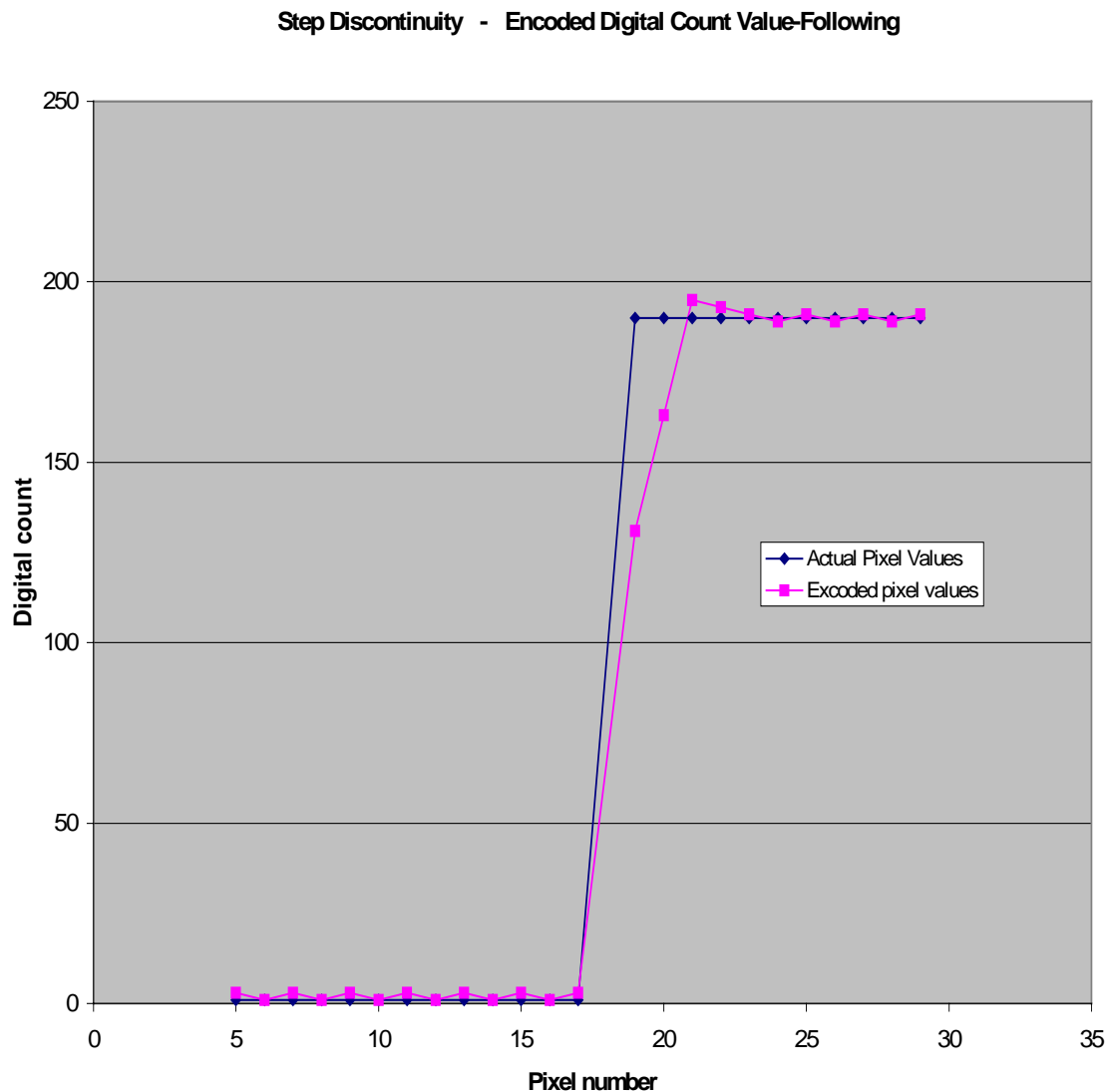


Figure 8 – Example of step discontinuity in digital counts, and algorithm tracking

Figure 8 illustrates this example, with the actual discontinuity encountered in the original image and how the algorithm would track the DC value of 0 in the original image, and how it would “home in” on the new DC value of 190:

- In an 8-bit, 256 gray-level image, the actual DC values go from 0 to 190.
 - The encoded DC value oscillates between 1 and 3 (the only allowed values close to zero since the encoding in any row starts with a value of 127, and all codewords are even numbered).
- From a value of 3, the codeword for +128 is added to get 131.
- From 131, the codeword for +32 is added to get 163.
- From 163, the codeword for +32 is added to get 195.
- From 195, the codeword for -2 is added to get 193.
- From 193, the codeword for -2 is added to get 191.
- From 191, the codeword for -2 is added to get 189.
- From 189, the codeword for +2 is added to get 191.
- From here on, the value of 2 is either subtracted or added so as to oscillate around the desired value of 190 (this is as close as we can achieve).

This particular example took 5 applications of the algorithm to arrive at a value that was steady state, within 2 digital counts of our target of 190. The question that remains is, for any arbitrary step discontinuity from 1 to 255, how many pixels would the algorithm require after the step, to arrive at the target DC value, within the tolerance band of 2 digital counts?

Numerical analysis has determined that for our 8-bit image set of cases, out of the 255 possible step values, 20 cases, 7.81%, are within our tolerance of 2 digital counts after application of the algorithms on the first pixel after the digital count discontinuity. Using the residual differences from the first application, that number jumps to 73, or 28.51% after the second pixel, and so on, as shown in Table 7 and Figure 9.

Number of pixels	Number of steps with residual DC counts within 2	% of steps whose residual DC count is within 2
1	20	7.81
2	73	28.51
3	161	62.89
4	231	90.23
5	254	99.22
6	256	100

Table 7 – Summary of how many steps are within +/- 2 digital counts of target after k application of the compression algorithm

The plots shown in Figure 8 have the step size measured in digital counts represented by the abscissa, and the remaining difference in digital counts between the step discontinuity and the accommodation made after the k^{th} pixel of the algorithm represented by the ordinate. For example, there are 0 remaining digital counts of difference for steps of sizes of 2, 8, 32, and 128, since step discontinuities of those values can be accommodated exactly by the algorithm. There is a remaining difference of 127 for a step size of 255 digital counts, since the maximum step that can be made is 128 digital counts. Successive applications of the algorithm on subsequent pixels work to make the remaining differences get smaller and smaller.

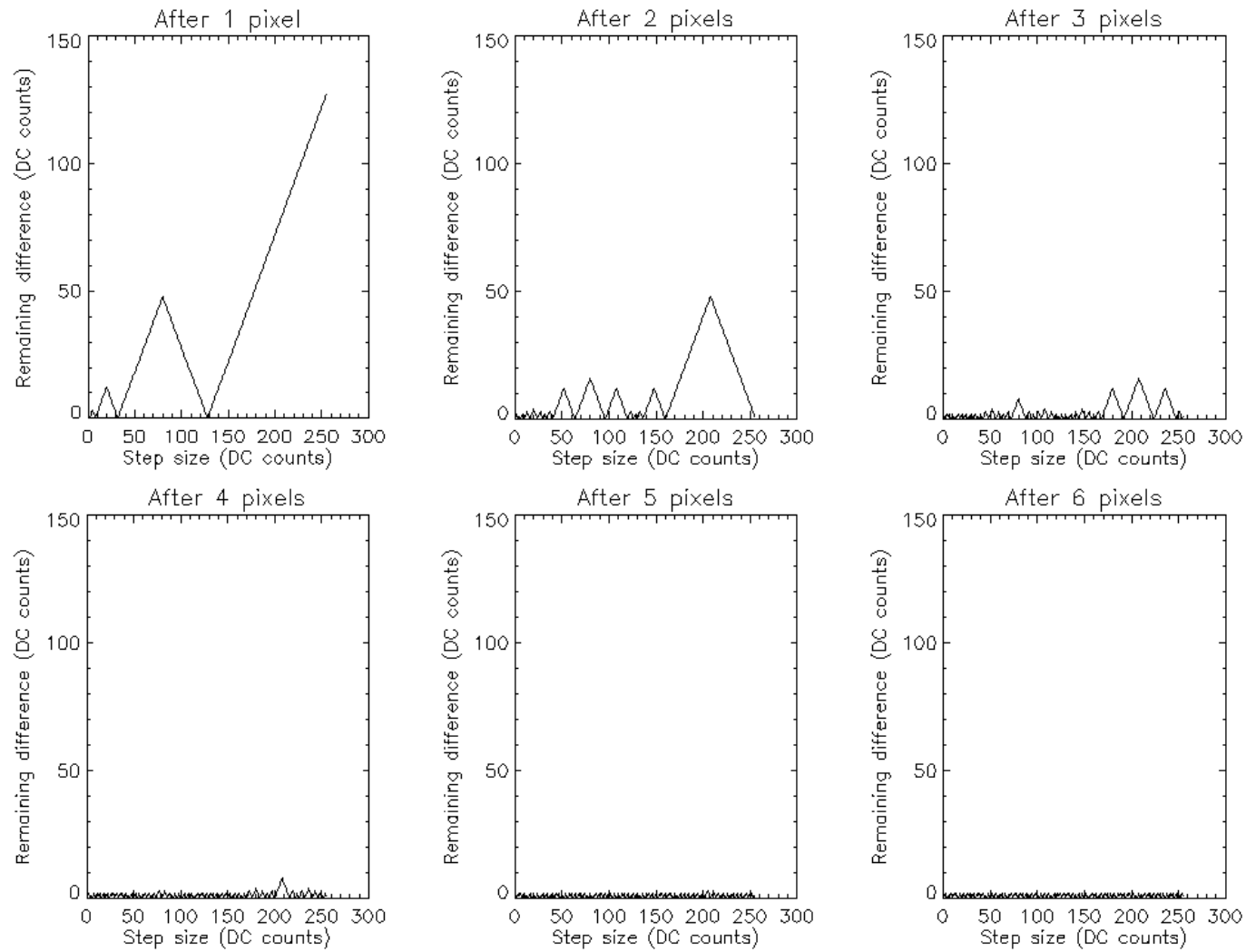


Figure 9 – Plots of residuals after k applications of the compression algorithm to various step discontinuities ranging from 1 to 255 digital counts

STANDARD BENCHMARK IMAGES:

This paper would not be complete without subjecting the Image Science benchmark images of The Baboon and Lena, and determining the PSNRs of them:

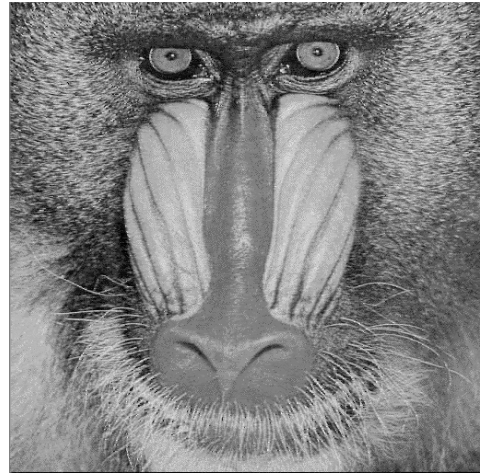
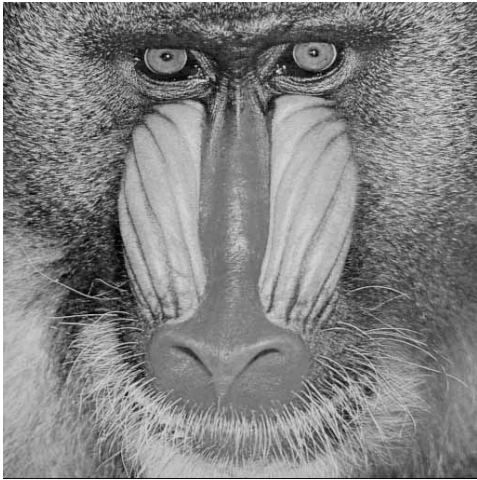


FIGURE 10 – Baboon (left – original right – compression encoded)
PSNR = 27.078 3 bit compression
PSNR = 30.715 4 bit compression



FIGURE 11 – Lena (left – original right – compression encoded)
PSNR = 29.773 3 bit compression
PSNR = 31.170 4 bit compression

QUALITATIVE EVALUATION

The 25 astronomical images compressed with this algorithm were judged qualitatively with respect to the original bitmaps. JPEG and JPEG2000 compressed versions, compressed to approximately the same size as the encoded images, were presented randomly as well, to act as controls. Six Image Scientists, whose work involves remote sensing and Earth observation, were the evaluators. They were asked to judge the compressed images with respect to information content utilizing the following instructions:

Instructions for astronomical image analysis:

1. 75 set of images will be presented. Each set consists of an original bitmap image (either on the left or above), and the same image compressed with some particular algorithm.
2. Assume you are judging the compressed image's information content with respect to the reference image, from the point of view of an astronomical image analyst.
3. Assign each compressed image a score of 0 (unusable) to 100 (good) based on your perception of how much useful qualitative information content, true to the original, that you note. Use the associated score sheet to write down the compressed image's score.

Artifacts may detract from a high score if they impact information content, but remember to judge primarily on useful information to be able to be derived, not on image recreation perfection.

The results were sorted grouping the all of the JPEG images together, all of the JPEG2000 images together, and all of the 3-bit compressed images together. The thought was that though the scales may be different between observers, but perhaps we would see the same relative scalings for a particular image. In other words, perhaps each observer would score the same image either lower on his or her respective scale (or higher), and a trend could be observed. The results are in the following graphs:

With respect to JPEG compression, it would appear that agreement was among nearly all observers with respect to images 5, 7, 9, and 25. Most ranked these JPEG compressed images to be not as good as the original images when making the comparisons (see Figure 12).

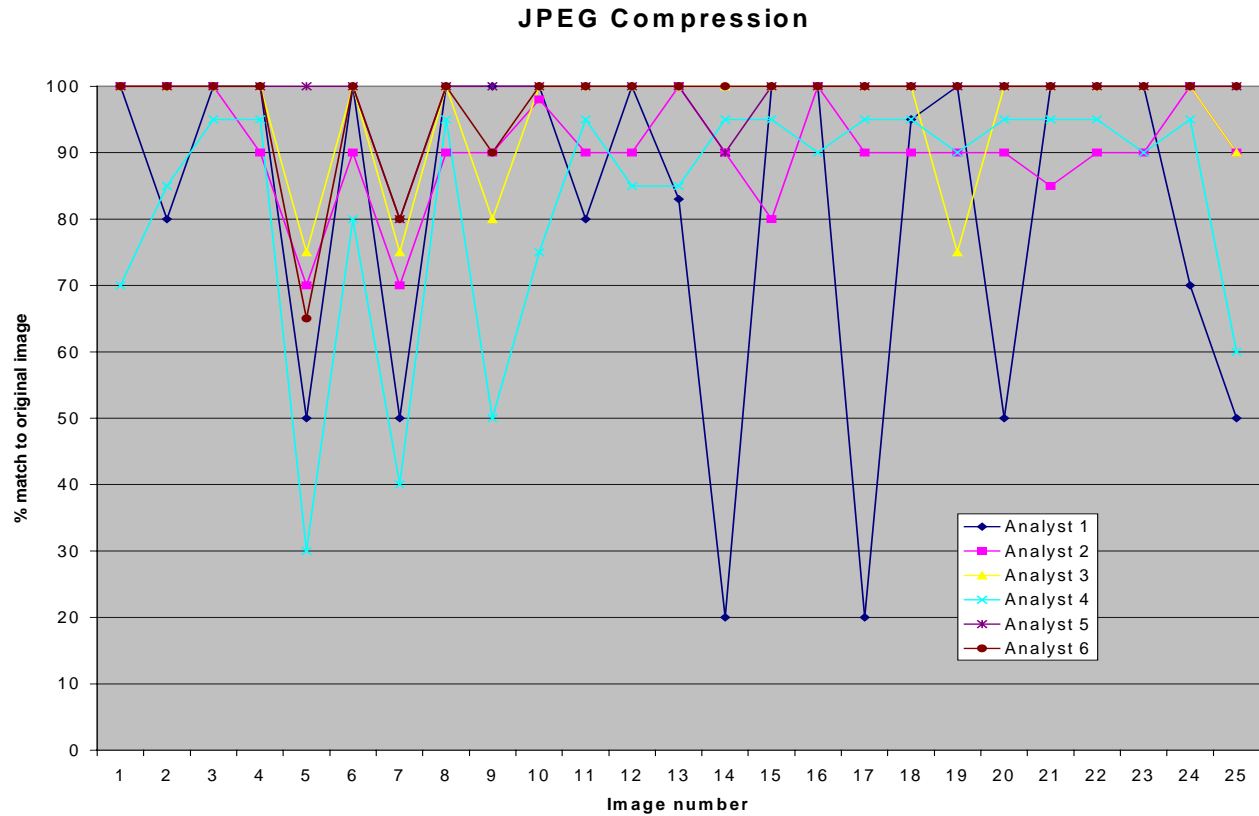


Figure 12 – JPEG Image Evaluations

With respect to JPEG 2000 compression, the same images as with the JPEG compression, 5, 7, 9, and 25, ranked lower in information content when compared to the original images (see Figure 13). Additionally, four found image 1 to be lower in information content with JPEG 2000.

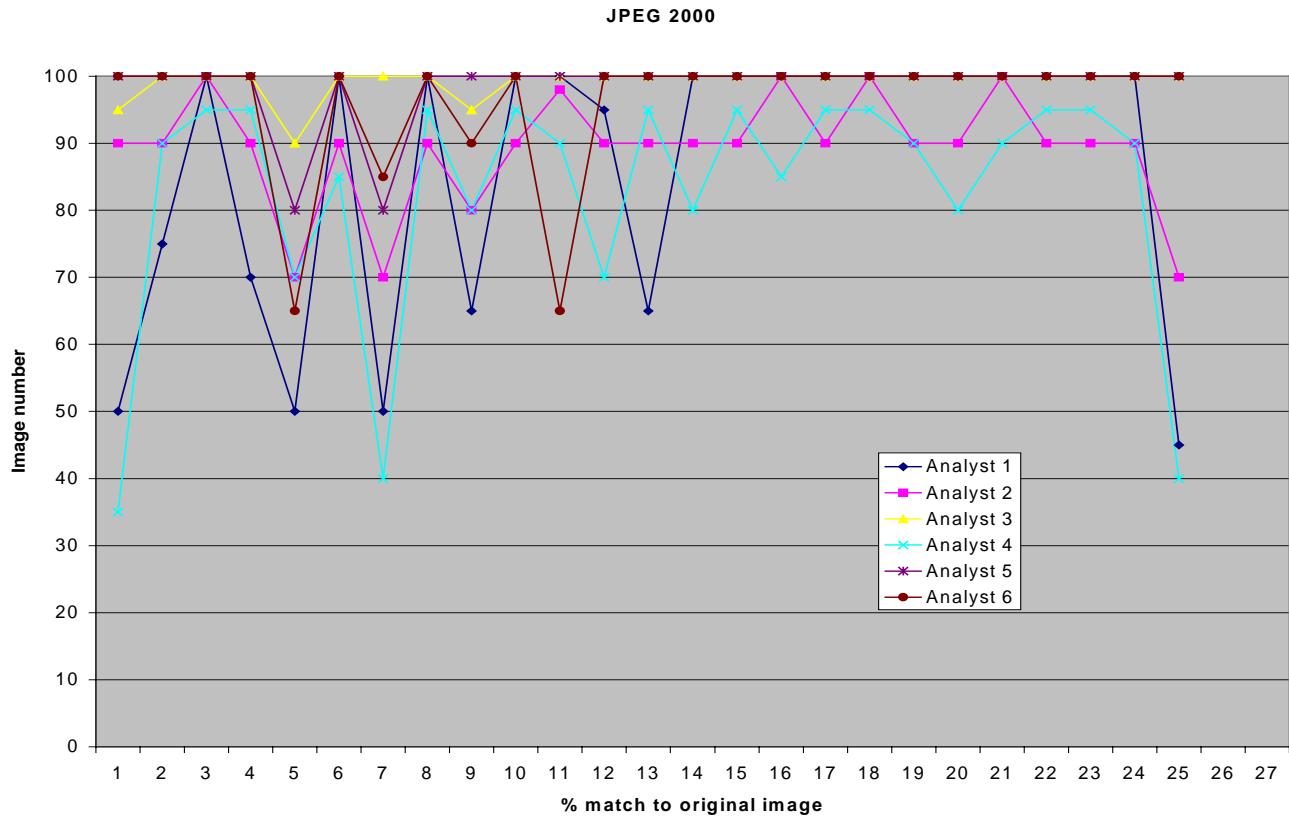


Figure 13 – JPEG 2000 Image Evaluations

And finally, the results of the compact 3-bit compressor would seem to indicate images 1, 5, 7, 12, 21, and 25 we less favorably received when compared to the original images (see Figure 14). These are images with regions of very high contrast, high frequency content. In other words, these images contained regions where large step discontinuities between adjacent pixels existed, and that is where the algorithm was expected to have some difficulties. But there is no correlation between these particular images and their PSNRs; they came in neither lower or higher than average. There is, however, a fair amount of decorrelated rankings among the rest of the images.

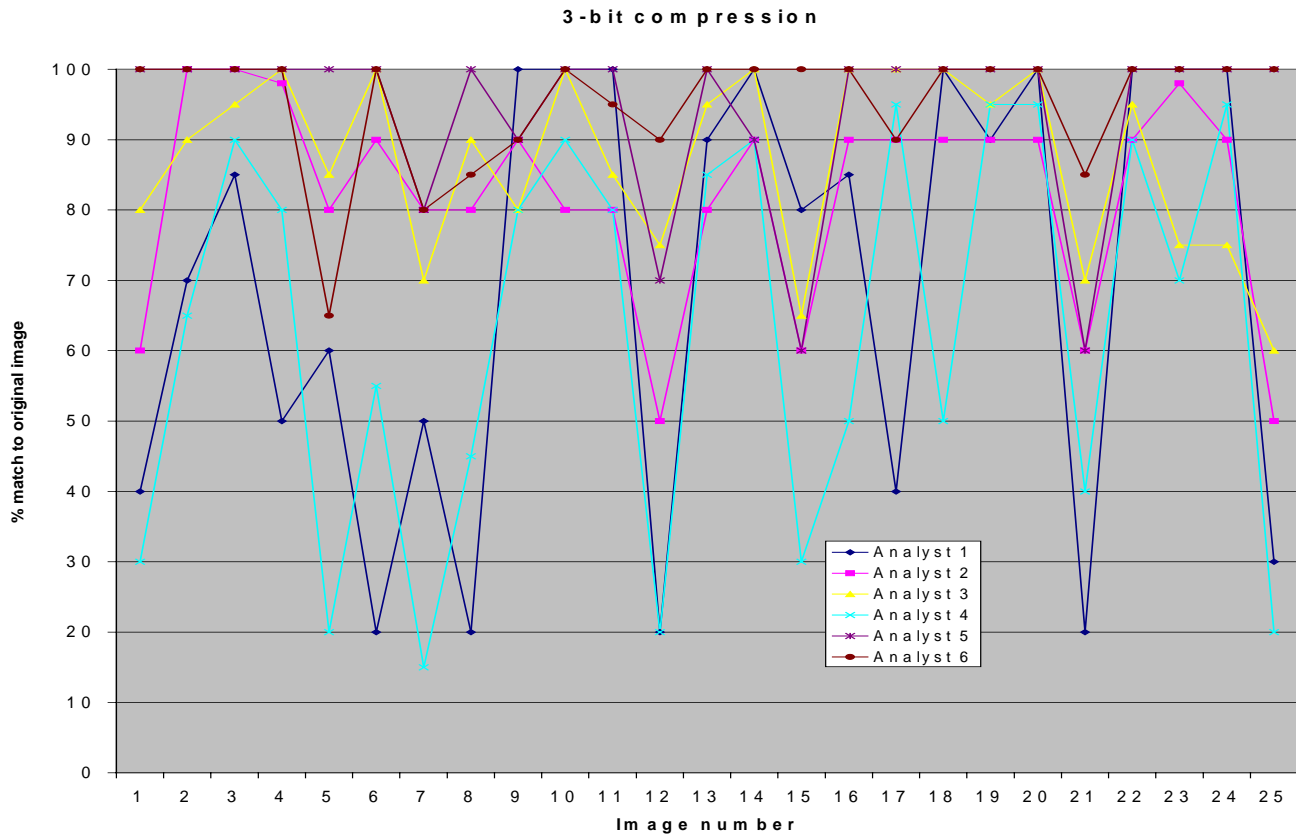


Figure 14 – 3-bit encoded Image Evaluations

This was a qualitative ranking, and the sample size was small. Future studies would benefit from an increased sample size, formalizing the instructional process for ranking, and including a number of cognizant experts from groups outside of the reconnaissance evaluation community, notably astronomers, engineers, physicists, planetologists and geologists.

FUTURE WORK – THE EXPANDED SERIES FOR ANY m-bit IMAGE

This algorithm can be expanded to work with any number of bits per pixel image. The following table shows the bits assignments for such pixel configurations, as well as compression ratios. Across the bottom of Table 8, the compression ratios of utilizing n-bit codewords for an m-bit image is listed, as well as the compression ratios of utilizing 3-bit codewords for a m-bit image. Utilizing 3-bit codewords is the equivalent of keeping only the 8 most significant bits in an m-bit image, and encoding it as described in the 8-bit examples discussed thus far.

Pixel Bits		Bit assignment								
		8 bit pixels	9 bit pixels	10 bit pixels	11 bit pixels	12 bit pixels	13 bit pixels	14 bit pixels	15 bit pixels	16 bit pixels
Bit	Value									
0	1		bit 1 - zero		bit 1 - zero		bit 1 - zero		bit 1 - zero	
1	2	bit 1 - zero		bit 1 - zero		bit 1 - zero		bit 1 - zero		bit 1 - zero
2	4		bit 1 - one		bit 1 - one		bit 1 - one		bit 1 - one	
3	8	bit 1 - one		bit 1 - one		bit 1 - one		bit 1 - one		bit 1 - one
4	16		bit 2 - zero		bit 2 - zero		bit 2 - zero		bit 2 - zero	
5	32	bit 2 - zero		bit 2 - zero		bit 2 - zero		bit 2 - zero		bit 2 - zero
6	64		bit 2 - one		bit 2 - one		bit 2 - one		bit 2 - one	
7	128	bit 2 - one		bit 2 - one		bit 2 - one		bit 2 - one		bit 2 - one
8	256		bit 3 - zero		bit 3 - zero		bit 3 - zero		bit 3 - zero	
9	512			bit 3 - zero		bit 3 - zero		bit 3 - zero		bit 3 - zero
10	1024				bit 3 - one		bit 3 - one		bit 3 - one	
11	2048					bit 3 - one		bit 3 - one		bit 3 - one
12	4096						bit 4 - zero		bit 4 - zero	
13	8192							bit 4 - zero		bit 4 - zero
14	16384								bit 4 - one	
15	32768									bit 4 - one
sign bit		bit 3	bit 4	bit 4	bit 4	bit 4	bit 5	bit 6	bit 7	bit 8
unassigned 1/2 bits		none	bit 3 - one	bit 3 - one	none	none	bit 4 - one	bit 4 - one	none	none
# bits/code word		3	4	4	4	4	5	5	5	5
resolution (bits)		2	1	2	1	2	1	2	1	2
Compression ratio		8:3	9:4	10:4	11:4	12:4	13:5	14:5	15:5	16:5
		2.67	2.25	2.50	2.75	3.00	2.60	2.80	3.00	3.20
Compression ratio if only 3 bits used		8:3	9:3	10:3	11:3	12:3	13:3	14:3	15:3	16:3
		2.67	3.00	3.33	3.67	4.00	4.33	4.67	5.00	5.33

Table 8 – Extrapolated table of codebook values for images of various bits gray-level

Additionally, the encoded values (powers of 2) in Table 8 are based on utilizing the individual bits of an m-bit image. There is no reason that these are the best values to encode. A survey of the histograms of past images acquired might suggest that there are other value that would be better suited to encode. For the 8-bit to 3-bit example, we have chosen 128, 32, 8, and 2 to encode as the values for differences. A survey of image histograms might indicate that the differences of 245, 74, 25, and 2 are the most likely to occur in most space-acquired images, and therefore these might be better DC deltas to encode.

UNIQUENESS:

The codebook and the encoded/decoded values, or deltas, are what makes this compression algorithm unique and inventive.

For the 3-bit compression, if the $n-2^{\text{th}}$ and $n-1^{\text{th}}$ have a DC value of X, and the n and $n+1^{\text{th}}$ have a value of Y (that is, 2 distinct DC values over at least 2 bits each), then this algorithm covers at least 75% of the difference over two iterative applications iterations (two pixels).

For the 4-bit compression, if the $n-2^{\text{th}}$ and $n-1^{\text{th}}$ have a DC value of X, and the n and $n+1^{\text{th}}$ have a value of Y (that is, 2 distinct DC values over at least 2 bits each), then this algorithm covers at least 92% of the difference over two iterative applications iterations (two pixels).

Since most images exhibit a great deal of interpixel redundancy, this accounts for even the 3-bit algorithm's ability to achieve PSNRs of approximately 30 (the figure generally accepted as a minimum value required to have one image indistinguishable from another).

CONCLUSION:

This algorithm can be utilized as an on-the-fly, non-probabilistic stand-alone compression, or as a probabilistic compression pre-processor. If used as the latter, a Huffman, Shannon-Fano or Arithmetic code could be applied to further reduce the bitstream.

Either way, this algorithm maintains PSNRs for most images at approximately 30dB for all cases. Its strength is in its simplicity, being non-computational and non-memory intensive, and its compactness.

- [1] Wong, Al. Galileo Frequently Asked Questions. 31 Dec. 1997. NASA/JPL. 8 Feb. 2008. <<http://www2.jpl.nasa.gov/galileo/faqimage.html>>.

APPENDIX
IMAGES and IDL CODE



Image 1



Image 2

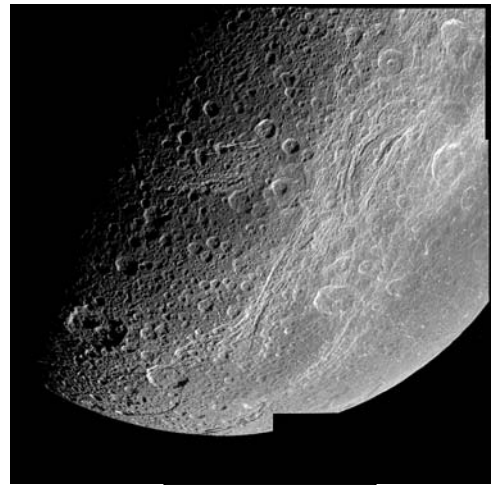


Image 3

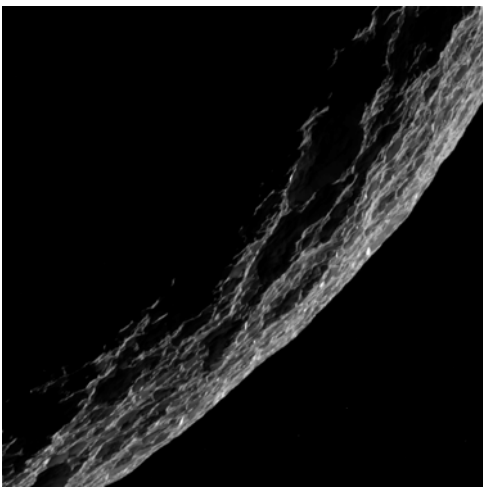


Image 4

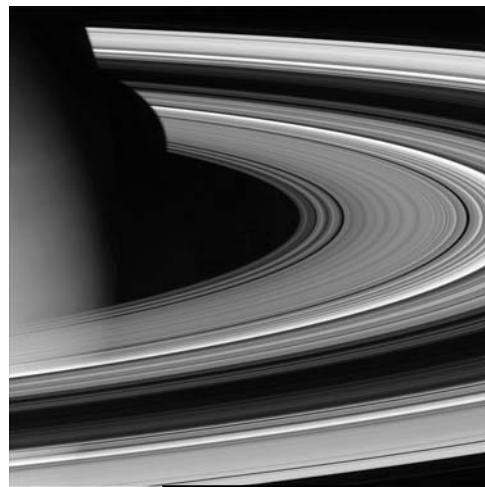


Image 5

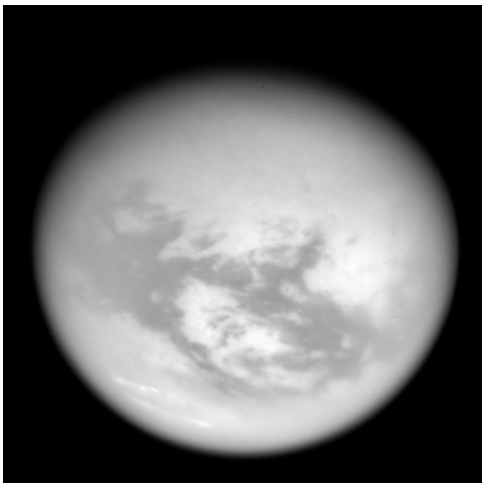


Image 6

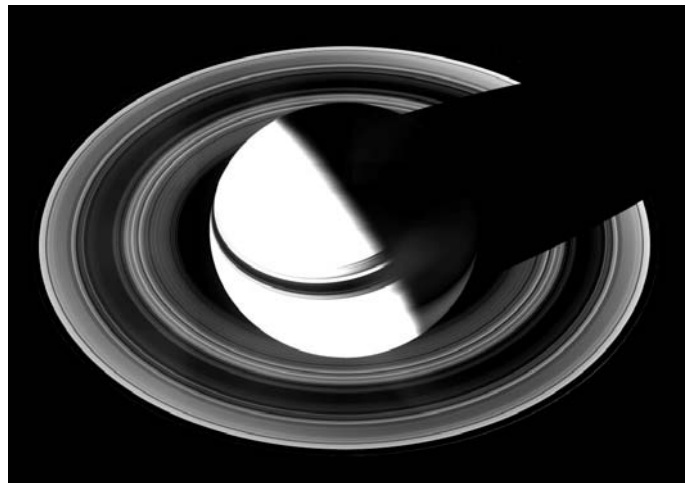


Image 7



Image 8

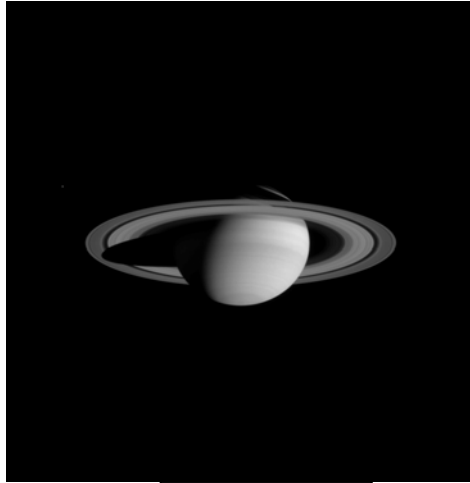


Image 9



Image 10

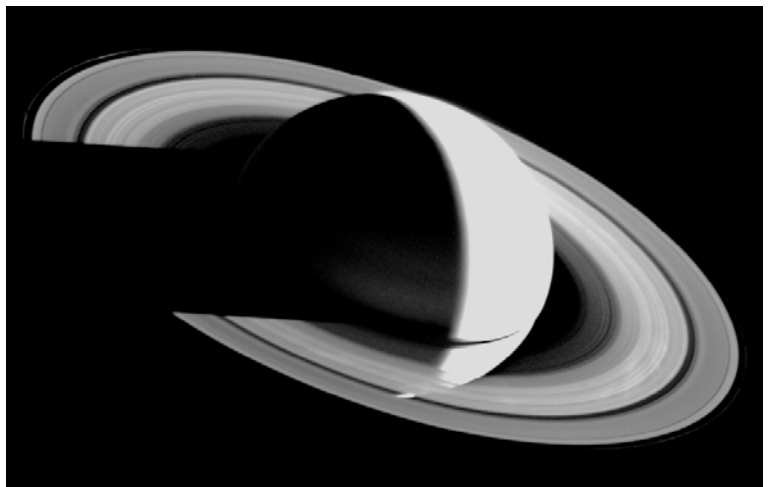


Image 11

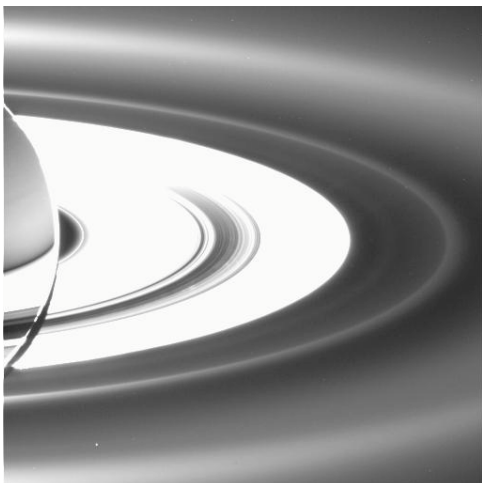


Image 12

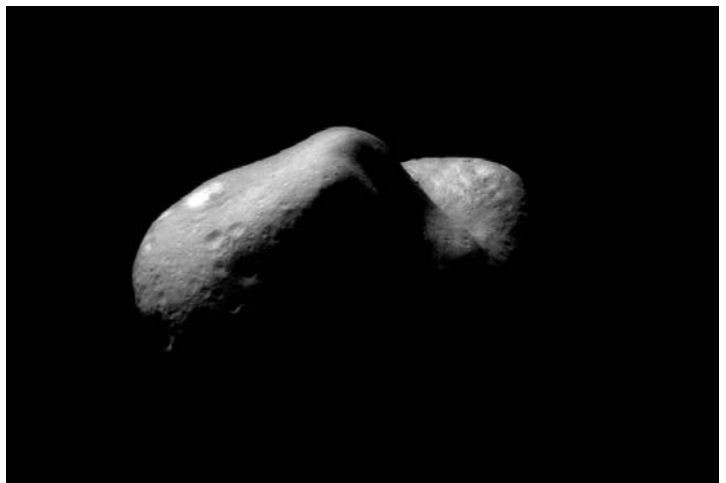


Image 13

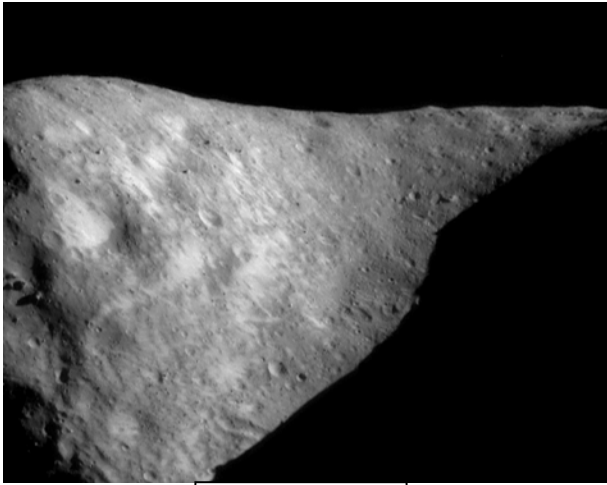


Image 14

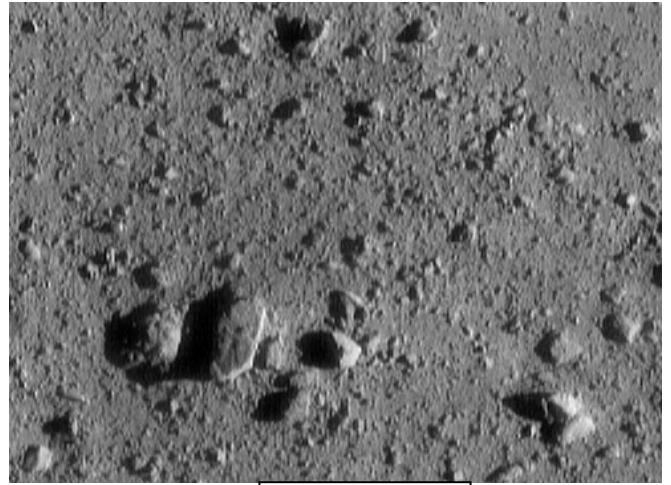


Image 15

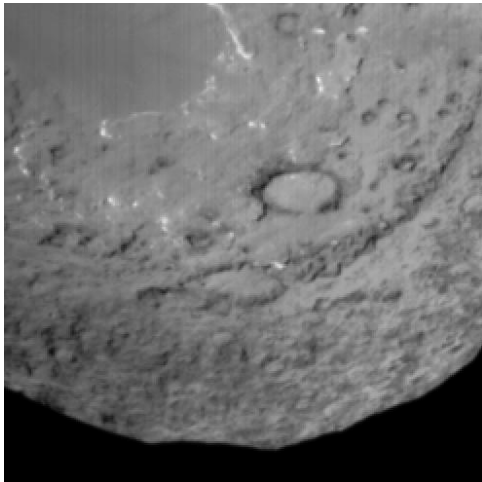


Image 16

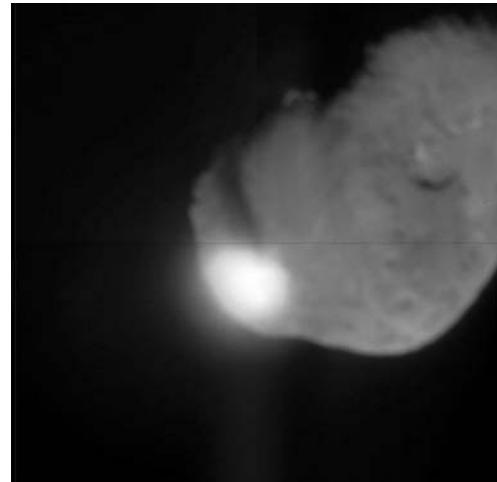


Image 17



Image 18

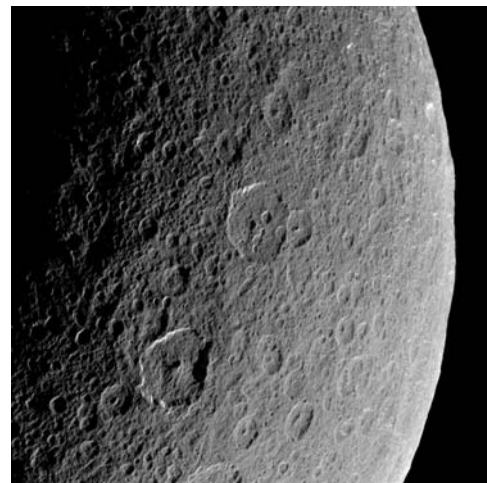


Image 19

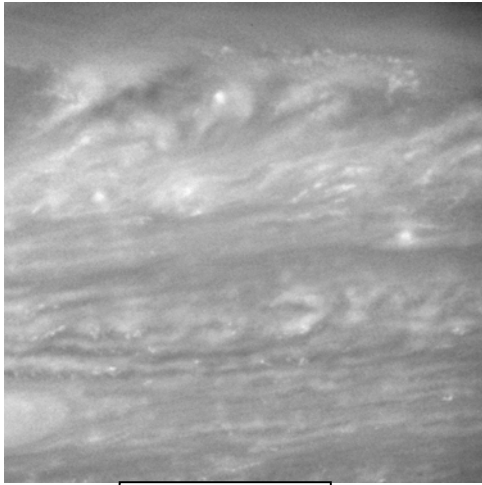


Image 20



Image 21

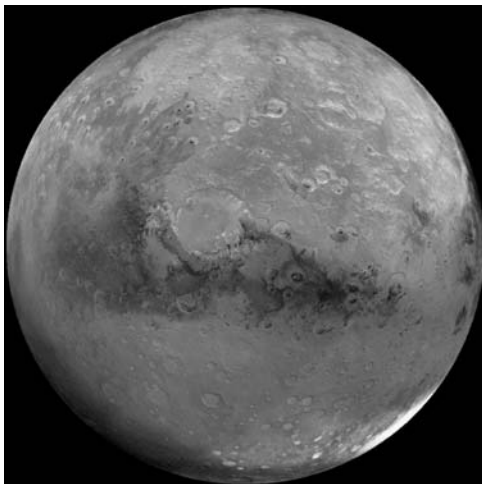


Image 22

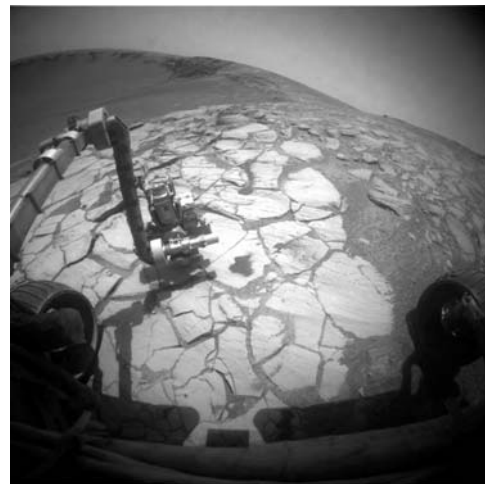


Image 23

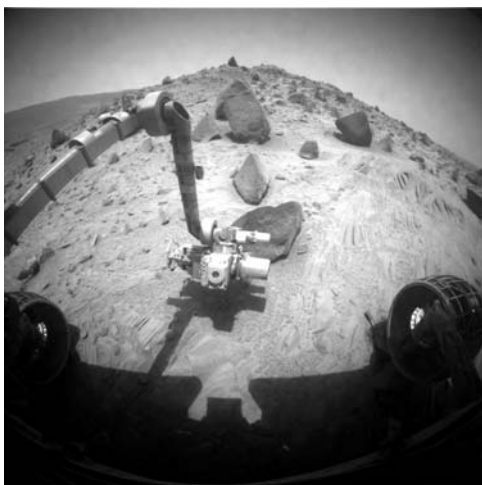


Image 24

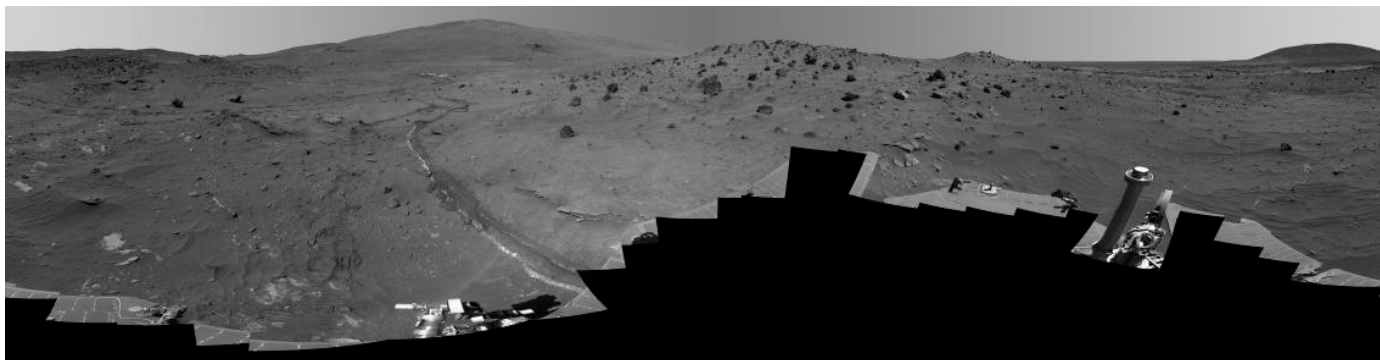


Image 25

IDL code for 3-bit compression
Invented and written by Gregory Gosian, April 2007

```
Pro FLIGHT
  A=INTARR(8)
  D=INTARR(8)
  E=INTARR(8)
  A=[-2,-8,-32,-128,2,8,32,128]
  FOR Y=0,1027 DO BEGIN
    P=127
    FOR X=1,1027 DO BEGIN
      FOR I=0,7 DO BEGIN
        D(I)=P+A(I)
        READ M
        E(I)=ABS(M-D(I))
        G=MIN(E, H)
      ENDFOR
      F=P+A(H)
      IF (F GT 255) OR (F LT 0) THEN BEGIN
        H=H-1
        IF H EQ 3 THEN H=0
        IF H EQ -1 THEN H=4
      ENDIF
      TRANSMIT H
      P=P+A(H)
    ENDFOR
  ENDFOR
END
```

VARIABLE LIST

A =8 element integer array (code book values)
D =8 element integer array (sums of code values and compressed image previous pixel)
E =8 element integer array (differences between *D* and real image current pixel)
F =integer variable (*P*+*A*(*H*), DC of compressed image current pixel)
G =integer variable (dummy)
H =integer variable (index of minimum value of *E*)
I =integer variable (code book index)
M =integer variable (real image current pixel)
P =integer variable (compressed image previous pixel)
X =integer variable (*X* position index)
Y =integer variable (*Y* position index)

READ M and *TRANSMIT H* statements are not IDL code – they represent where the actual image pixel DC value (*M*) is read into the code, and where the code word *H* is transmitted to the receiver.

MEMORY REQUIREMENTS:

Each integer variable is 16 bits (8 variables – total of 128 bits 8 16 bit words)
Each 8 element array is 128 bits (3 variables – total of 384 bits 24 16 bit words)
Code takes 2488 bits (311 8 bit words)

Total= 3000 bits

Entire compression takes <1 kilobyte (equivalent of 375 bytes {375 8-bit words})