SMPTE Meeting Presentation

A Quality Metric For High Dynamic Range

Gary Demos

Image Essence, Perris, California, garyd@alumni.caltech.edu

Written for presentation at the

SMPTE 2014 Annual Technical Conference & Exhibition

Abstract. *The Peak Signal to Noise Ratio (PSNR) metric has long been utilized for codec evaluation and development, and other uses. However, for High Dynamic Range (HDR), the PSNR metric is not suitable. A more appropriate characterization of coding and image quality is to split image brightness into ranges (such as factors of two), and then determine the standard deviation within each such range. Once the standard deviation (sigma) has been determined, the two sigma and above population of pixel differences is shown as percentages of pixels. This is necessary because codec pixel differences do not typically follow a normal Gaussian error distribution. The value of sigma at each brightness range, together with the percentage proportions of two sigma and above outliers, provides an appropriate quality metric system for HDR.*

Keywords. HDR, Sigma, PSNR, Outliers

Introduction

A quality metric is needed for HDR which serves the same function as PSNR, yet extends and clarifies this ubiquitous metric.

Herein we introduce a code tool, called “sigma_compare” to provide a suite of quality metrics over a wide dynamic range.

The “sigma_compare.cpp” Tool

The sigma_compare tool (see the Appendix) compares a pair of frame sequences that are presumed to be the High Dynamic Range (HDR) input and HDR output of an HDR codec. Although codec performance evaluation is the primary intended use of sigma_compare, the sigma_compare tool is also useful for characterizing temporal (thermal and shot) noise from cameras. This is done by capturing a multiple (two or more) frames from a very still scene containing one or more test charts. It is best if the scene is slightly out of focus, and that the air, scene, and camera are extremely still. No more will be said herein about such other uses of sigma_compare. The emphasis here will be on characterizing codecs using sigma_compare.

Since exr half-float files (typically OpenExr) are the dominant HDR file format, the current version of sigma_compare reads two OpenExr frame sequences of files for comparison. The sigma_compare tool also supports dpx32 input (other dpx formats are not currently recommended). If a dpx_file_io reader is not available, declare dpx_read to be an empty function (instead of a prototype), which prints an error message and exits the program (exit(1);). Then sigma_compare can be built for exr file input only.

The second sequence is the “test” sequence, which is presumably the output of a decoder, transformed back into the original half-float (or float) HDR format of the original HDR image. The first file sequence is the “original”. The original image sequence format should be one of the following linear (unity gamma) sigma_compare formats:

One: ACES RGB

Two: CAMERA-NATIVE RGB

Three: XYZ tristimulus from a reference display or projector

Four: P3 RGB from a reference display or projector

Five: A simulated HDR RGB or XYZ scene (or HDR test charts)

The sigma_compare tool is not intended for use with YUV, 4:2:0. If any such gamma chroma-subsampled formats are used in testing, the original RGB (or XYZ) must be restored for use as a test input for sigma_compare. The actual original in 4:4:4, and not a derived original converted from a 4:2:0 version of the original, must serve as the actual original. The 4:2:0 (or 4:2:2) chroma subsampled intermediate format should be considered as part of the operation of the codec. The round-trip from 4:4:4 to 4:2:0 (or 4:2:2) and back to 4:4:4 should be part of the processing of the decoded test image in order to be fair to 4:4:4 codecs which do not include this loss. Thus, the 4:2:0 (or 4:2:2) loss should be included in the codec's test output. It will not be possible to interpret the sigma_compare output in any format not in the above list of supported original (4:4:4) sigma_compare formats vs. the complete processing decoding required to create corresponding test images in the same format.

Note that portions of the codec process, beginning and ending at the exr half-float RGB (or XYZ) format, can be tested one step at a time. This allows separate measurement of the additional loss of each added from/to step from/to exr in codec processing. The final step would typically be the codec itself. Such partitioning is unnecessary for codecs that directly encode and decode linear (gamma 1.0) exr RGB (or XYZ).

Motivation For sigma_compare

The "Peak Signal to Noise Ratio" PSNR quality metric does not apply to HDR images. The use of a "peak" does not correspond to typical HDR scenes, which are typically anchored at a mid-gray (similar to film's Laboratory Aim Density). Further, the use of PSNR applied to gamma-encoded signals does not correspond to a physically meaningful measurement. This becomes even less defined when other non-linear curves are used instead of gamma.

Thus a new quality metric family is needed for use with HDR.

Perceptual Threshold

The DCinema (DCI/DC28) gamma 2.6 X'Y'Z' 12-bit specification was developed using the Barten "flat" and "ramp" perceptual thresholds, as well as additional tests using the Barten methods applied using a DLP projector in a theater.

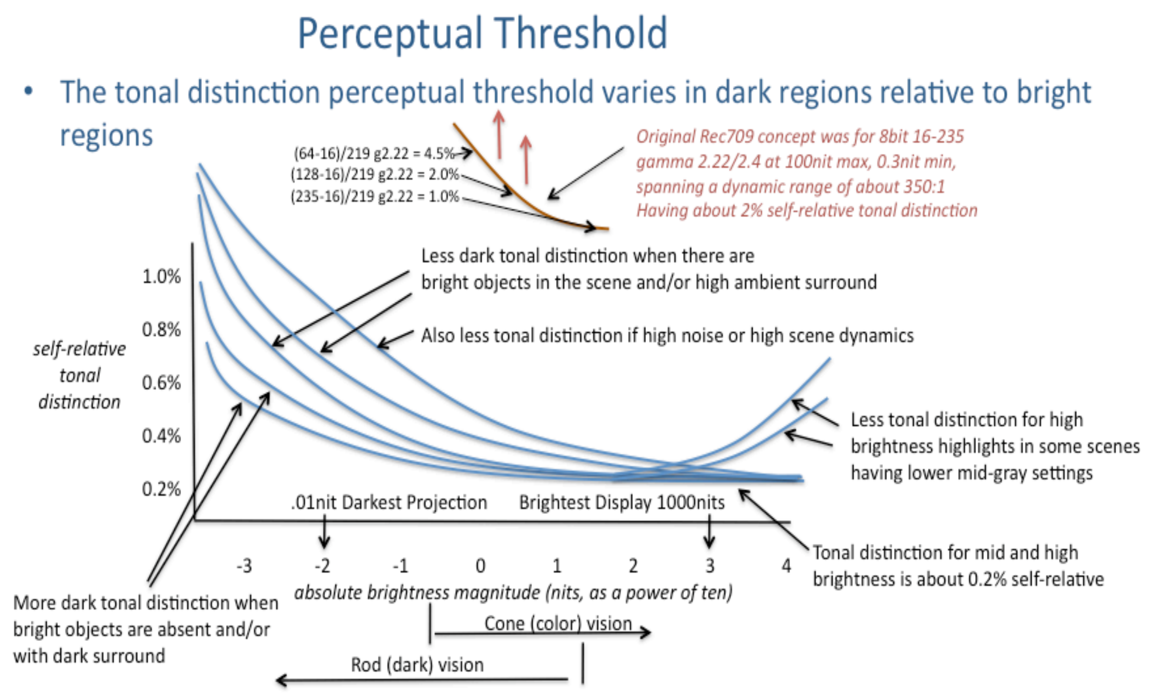
The plots used for such evaluations utilize a self-relative amount, often a percentage, on the vertical axis (either linear or logarithmic). The horizontal axis utilizes a logarithmic brightness level. The plot itself indicates where the threshold falls in units of self-relative steps, against the various brightnesses.

Often such plots show gamma curves, quasi-log curves, and various candidate perceptual quantizers. The plot typically indicates the stepsize of the least significant bit for a given integer bit depth in some tests, or the discrimination limit (horizontal low contrast bars vs. vertical, in one key test). This goes by various names including contrast sensitivity (which is often displayed as a function of spatial frequency), tonal distinction, and the perceptual threshold. The perceptual threshold is probably the most appropriate name for use herein, but these various descriptive names will be sometimes used interchangeably.

Plots and Variability of the Perceptual Threshold

For a codec, the perceptual threshold is an over-simplification. Usually codecs operate well above the perceptual threshold. In 8-bit video coding, for example, the least significant of the 8 bits is an order of magnitude more crude than any perceptual threshold metric. Only the highest quality and highest bitrate uses of 8 bit codecs even come close to 8 bits (perhaps typically achieving 7bits at very high bitrates).

Further, there is no single perceptual threshold. Such a threshold depends upon the type of test, the scene contents, and ambient surround conditions. The Barten flat and Barten ramp tests differ by a factor of approximately two in showing thresholds at ¼% and ½% self-relative. Tests may also be accurate only for specific brightness levels, although the resulting threshold curves are sometimes inaccurately applied at other brightness levels. The main issue, however, with all perceptual thresholds, is that they are inherently variable depending on scene contents, absolute brightness, and ambient surround conditions.



Self-Relative Statistics

It can also be seen from any perceptual threshold plot that the spread function is missing. The threshold is shown as a very specific delineation, whereas statistical variation due to a codec is some sort of error spread distribution. Given that PSNR is not suitable, the sigma_compare tool takes the approach of creating a standard deviation (sigma) of the original vs. decoded pixel values. The sigma_compare tool does so for each octave over a very large range. The current code setting (a compile flag called "`NUMBER_OF_STOPS`") is 32 stops of range. The 5-bit half-float exponent represents 32 stops of range (two stops used for flag codes), but does not account for the denormalized dark range (at the lowest exponent). The 32-stop range of sigma_compare is split between below 1.0 and above 1.0, with 16 stops of range in each. Since sigma is being created in linear light, it also corresponds to Root Mean Square (RMS) error energy in relation to the average brightness energy within each stop. The average brightness of each pixel primary is used for the self-relative ratio of sigma to the primary's value. This is more appropriate than just using the center of each stop range, since pixel values may not be (and usually are not) evenly distributed within each stop of range.

Gaussian vs. Non-Gaussian Statistics

The error function of many common random statistical distributions is a Gaussian function, with sigma having a specific location within the Gaussian (such that sigma completely specifies the Gaussian). However, there is usually nothing about the tools used within codecs that will yield a "normal" Gaussian distribution. Thus, the statistical distribution for self-relative brightness sigma will not, in general, be a Gaussian. Thus, singular use of only sigma does not define the error distribution when characterizing codecs. Because of this, the sigma_compare tool determines sigma as a first pass. A subsequent pass is then applied, once sigma has been determined, in order to compute the percentage of pixel primary values which lie outside of two sigma, three sigma, and higher sigma multiples. With a standard Gaussian, about 32% of pixel deviations lie outside one sigma, 5% lie outside two sigma, and 0.27% lie outside of three sigma. Here is the outlier distribution for the Gaussian Normal Distribution for one color primary:

Sigma Multiple	Outside Sigma	4K Outliers/Frame	4K Outlier Pixels/Min
		(4k x 2160)	(4Kx2160x24fpsx60sec)
1 sigma	31.73%	2.807MPixels/Frm	4.04 GPixels/Min
2 sigma	4.55%	402.6 KPixels/Frm	580 MPixels/Min
3 sigma	.270%	23.9 KPixels/Frm	34.4 MPixels/Min
4 sigma	.00633%	560 Pixels/Frm	806 KPixels/Min
6 sigma	.000000197%	.017 Pixels/Frm	25.1 Pixels/Min

With codecs, these numbers will typically be very different (not being a normal Gaussian error distribution). Given one to ten million pixels in an image, small proportions of outliers will often be visible somewhere in each image. The use of sigma multiples remedies a longstanding deficiency in PSNR by identifying the energy concentration of outlier errors at a given codec setting within a given codec.

It should further be noted that codec transforms have particular difficulty on the dark side of a sharp bright edge. Some transform ringing into the dark region can result in very large self-relative errors. This may occur at only a small percentage of the pixels within the image. Whether such large errors are visible or not requires visual scrutiny of the specific error-prone regions. It should also be noted that noise in extreme dark regions may not be reproduced by a codec (effectively reducing the resulting dark noise). Such an error may be a large self-relative proportion, and yet may actually be visually beneficial. Other dark errors may quantize-away dark detail, or worse yet, replace such detail with meaningless artifacts. Again, visual scrutiny often represents the only effective means of evaluating these types of codec errors.

Interpreting The (One) Sigma Results

The sigma values are printed first, over every stop range where there is at least one pixel value for that primary within that stop of range. Recall from above that sigma must be determined in a first pass before sigma multiples can be determined and characterized.

For each primary, sigma is given in its pixel units (whatever is in the image) within that stop of range. The ratio of that sigma to the average value in that stop range is shown next, along with the average value. Lastly, the number of pixels at that value is shown.

Most codecs give less bit precision to blue (or Z in XYZ). Further, most blue signals from cameras (and film) are much noisier, and sometimes have less resolution (although sometimes blue has full resolution). Thus, one will typically see higher self-relative sigma proportions in the blue channel compared to red and green.

Interpreting Sigma Multiples

In a second pass, since sigma has already been determined by the first pass, the number of pixels for each primary outside various sigma multiples are shown whenever there is at least one such outlier pixel. Also, the percentage of all pixels for that primary outside the sigma multiple within that one stop range is shown. The values of sigma multiples are shown for convenience, but are simply the given multiple times sigma (and corresponding self-relative values) from the first pass. It is the percentage of the pixels (and total number of pixels) within each stop range for each primary outside of the sigma multiple that is of interest. That percentage provides a reasonable indication of the statistical distribution of the codec's coding errors by measuring the percentage of multiple sigma outliers. The specific outlier count is also relevant, since the count can sometimes be only a few pixels over a sequence of many frames (usually associated with high sigma multiples).

Cases have been seen where a codec produces very high quality at sigma, but has outliers many times beyond sigma. This can be due to simple careful processing for most pixels, but where there is complex divergent processing for a small portion of the pixels. The sigma_compare tool currently is implemented with two, three, four, six, eight, twelve, and sixteen-sigma outlier statistics.

Approximate Relationship Between Self-Relative and Mantissa Precision

Floating point can be thought of as being piecewise linear logarithmic. The exponent is logarithmic, and the mantissa is linear over each factor of two that is scaled by the exponent. A self-relative ratio measurement is conceptually logarithmic, since logarithmic values have a constant ratio from one value to the next. The half-float has one bit of sign, five bits of exponent (32 stops of range, less two codes for out-of-range), and a ten bit mantissa. With 32-bit float, the high order bits of the mantissa will be the same as with the 16-bit half-float, but there will be many more low-order mantissa bits beyond those high order bits.

Although it is very approximate to do so, it is possible to think of the self-relative deviation as being the number of accurate bits within the mantissa. For a 16-bit half float, a value of 0 in the mantissa can be considered as representing 1024/2048 having an implied one bit in the eleventh most significant bit. A mantissa value of 1023 can be considered as being 2047/2048. Spanning this range, a middle value of 1536 (512 in 10bits + 1024 implicit) can be used to approximate the central value for these self-relative approximations. This yields a self relative sigma of 0.5% as representing about 7 mantissa bits. A self relative three sigma of 4.0% would indicate that about 4 mantissa bits of accuracy lie within three sigma.

For the 16-bit half-float:

Self-Relative %	**Approx. Number of Accurate Mantissa Bits**	**Inaccurate Mantissa Bits**
0.065%	10 bits	0 (half float limit)
0.13%	9 bits	1 bit
0.25%	8 bits	2 (percept thresh)
0.5%	7 bits	3 (alt. percept thr)
1.0%	6 bits	4 bits
2.0%	5 bits	5 bits
4.0%	4 bits	6 bits
8.0%	3 bits	7 bits
16%	2 bits	8 bits

Note: this table is the same for 32-bit floats, except that 13 bits should be added to the number of inaccurate bits (23bits of mantissa instead of half-float's 10bit mantissa).

Negative Numbers

Since negative numbers are common in HDR data, but are difficult to understand and utilize, they are handled separately within sigma_compare. A basic characterization is provided for average negative values. Self-relative sigma proportions of the differences in proportion to the average of those negative values are shown. The intent is to give some indication of the average and sigma of the population of negative primary values (when present). For codecs that clip negative values up to zero, the negative average and sigma will be similar in value, since sigma will usually be relatively near the negative average, and is thus near the actual negative clip error.

Test Patterns With Zero, and Cases of Low Sigma Having Many-Sigma Outliers

Some test patterns will have values at 0.0. Since this is not negative, it will fall into the first stop of positive range. If there are no other values (than 0.0) in the lowest stop, then the average within the first stop will be 0.0. The printout divides any sigma value by the average value to obtain the self-relative proportion. This will yield infinity, even for extremely small standard deviations (e.g. 1.0e-7/0.0). This will similarly occur in the printout of sigma multiples for the lowest stop. This is not a problem, and will typically only occur with test images.

Note also that very accurate coding on most pixels, with a few disparate pixels, can show up as a very low sigma, but with a few outliers at many times sigma. Such rare outliers may still have very low absolute and self-relative differences, but may still show up as many sigma outliers due to these special circumstances.

A perusal of all of the printout values from sigma_compare can provide the appropriate perspective.

Derived Printouts and Plots

The sigma_compare source code can easily be modified to print out subsets. However, any modification to the printout makes it difficult to share common results for comparison. Making additional printouts, or modified printouts to one or more custom files, can provide customization while leaving the existing printout in tact. It may also sometimes be preferable to extract fields from the existing output using a parsing editor (e.g. SED).

Plotting tools (e.g. gnuplot or excel) can provide plots from simplified or parsed printouts. Recall, however, that no single plot, such as one-sigma, can provide a characterization of outlier statistics. A suite of plots can provide such a perspective.

Customization of the sigma_compare results is encouraged. Such customization can aid in understanding, or as an analysis tool for specific issues. However, effective communication of sigma_compare results requires a common form, since the data often quickly becomes overwhelming. Improved printouts or plots using the sigma_compare engine can potentially provide substantial benefit. Such benefit is strongest if it takes a common format for use in communicating. Anyone who experiments with any modifications of the sigma_compare printout, and shares such modified printout with others, will quickly learn the difficulty of communicating such results. It is therefore recommended that improved printouts or plots be made in addition to the existing sigma_compare printout, or as a parallel version which is run in conjunction with the existing version.

The sigma_compare printout is fairly simple when the images have limited dynamic range. If the images have high dynamic range, then the printout will become lengthy (perhaps a dozen pages of long print lines). However, with use, the sigma_compare printout will become familiar such that its meaning is quickly seen.

If new plots or printouts are found to be broadly useful, then they could potentially be added to sigma_compare. However, sigma_compare is currently generic with respect to scale factor (i.e. the mid-gray) and other attributes. Custom plots or printouts may be specific to some such attributes, or may represent a subset of information, and thus would not be appropriate for adding to sigma_compare for broad use. For broadly useful additions, however, future sigma_compare upgrades could be highly beneficial.

Scale Factor

There is a command-line factor for the amplitude of the horizontal axis (thus the pixel values themselves). The default for amplitude_factor is 1.0. However, if the half-float pixel values represent a different scale for 1.0 (e.g. 1.0 = 100nits), then a different amplitude can be utilized to scale the pixel values (e.g. amplitude_factor set to 100.0). This amplitude factor has no affect on the percentages of sigma multiple outliers, and has no affect on the self-relative proportions. The absolute sigma and average primary pixel values are scaled, however.

Since the essential information is the statistics of multiple sigma outliers, and the self-relative proportion of sigma, the amplitude_factor does not affect the fundamental operation of sigma_compare.

Testing Integer Codecs vs. Other Integer Codecs and vs. Floating Point Codecs

Typical integer codecs have used a gamma or modified gamma function. For example, DCinema has used gamma 2.6 X'Y'Z' at 12-bits with JPEG-2000. The 12-bit specification is a maximum limit, and actual decoded values will have less than 12-bits of accuracy (typically with several inaccurate low order bits). Similarly, H.265 Main-10 has a maximum of 10-bits in 4:2:0, typically with a video-style gamma function (2.22 or 2.4 gamma with linear toe at black). Again, round-trip coding and decoding will not yield 10-bits of accuracy from H.265 Main-10.

Codecs have also used quasi-log functions with cameras, such as S-Log, PanaLog, and Log-C.

It is difficult to compare candidate HDR codecs in non-linear spaces, since it is difficult to understand the meanings of integer differences in these spaces, as well as the statistics of these differences. It is therefore herein recommended that all testing be done in 4:4:4 linear (gamma 1.0) spaces using the sigma_compare tool. Such testing can be used to refine and adjust codec efficiency and capability. Such testing can also be used to compare various candidate HDR configurations of non-HDR codecs vs. HDR codecs. There is interest in attempting to use perceptually-optimized non-linear functions similar to quasi-log and gamma, but with a high dynamic range compaction into the reduced dynamic range corresponding to the capabilities of common integer codecs. There is further interest in understanding how this compares to HDR-native codecs, such as floating-point codecs.

To test a non-linear function used with JPEG-2000, begin and end with a floating point 4:4:4 HDR image in XYZ tristimulus (gamma 1.0), P3 RGB (gamma 1.0), or ACES RGB (gamma 1.0).

Compress and decompress with appropriate test compression bitrates. Compare the input and output using sigma_compare.

To test a non-linear function used with H.265, begin and end with a floating point 4:4:4 HDR image in a space corresponding to the space used for comparison. For example, if comparing with JPEG-2000 using XYZ tristimulus (gamma 1.0) floating point, use this same space as input and output for testing a non-linear function applied to H.265. Then that space must be converted to 4:2:0 (for Main-10, for example), and may require additional color space alternations appropriate for use with H.265. Compress and decompress with appropriate test compression bitrates. With H.265, there may also be both intra and block motion compensated tests. Invert the conversion from 4:2:0 back to XYZ tristimulus (gamma 1.0) 4:4:4 floating point. Then compare the input and output using sigma_compare. The output of sigma_compare can then be used for comparison of H.265 and JPEG-2000 at various test bitrates. This approach also applies to Rec709, P3, BT2020, and other color spaces. It is also common to utilize multiple intermediate color transforms, although in theory all such transforms can be concatenated into a single matrix when applied.

Current HDR floating point codecs accept and reproduce these linear floating point color spaces natively, so the only fundamental choices are bitrate and intra vs. motion-compensation. Secondary codec compression configuration choices will also typically be selected for testing (such as weighting of bitrate to dark vs. light regions, which serves the same role as non-linear functions as used with integer codecs). Sigma_compare can be applied directly to the input and output (no conversions are needed). The output of sigma_compare can then be used for comparison of HDR floating point codecs with H.265 and JPEG-2000, when the H.265 and JPEG-2000 testing is processed as described.

Limitations

As with PSNR, no numeric characterization of pixel differences can reveal all quality attributes of lossy image compression codecs. A visual evaluation of codec image quality should also be utilized. For example, the distribution of pixel coding differences is not defined with respect to patterns, such as macroblock edges, which may be highly visible, but may not yield much sigma difference. Similarly, the sharpness of image details, both low amplitude and high amplitude, may not be apparent in sigma values, and yet may be quite apparent in visual comparison and evaluation. Codec transform errors on the dark side of high dynamic range sharp edges may or may not be visible, even when there is locally high self-relative error.

The treatment of noise is also difficult to characterize, both visually and with sigma. A removal of some amount of noise, or a reduction in noise, may not hinder image quality, and yet will increase the sigma difference. It is common for codecs to reduce noise to some degree, which will yield a higher sigma if the source images are noisy. Another simple example of sigma (and PSNR) limitation is an overall or regional slight offset in pixel value. Such an offset may be invisible to the eye, yet may significantly increase sigma. It is best to remove any regional pixel value offsets whenever possible (such as by the use of careful rounding). The sigma value should thus indicate localized medium and small spatial frequency differences that occur when coding various levels of detail within each image frame. A spatial frequency low-pass filter can be used with both the original and test image to determine the amount of regional low spatial frequency disparity. Typically such low spatial frequency disparity should be relatively small, and thus yield a small sigma.

Future Explorations

The Structural Similarity Index (SSIM Wang et. al., Hassan et. al.) provides a displacement-tolerant model of quality according to a model of the Human Visual System (HVS). This differs from a perceptual threshold-based model like sigma_compare. SSIM is using a weighted blend of measures for local luminance difference, local contrast difference (using sigma), and local structure (using correlation). However, constants used in these terms prevent extension to dark values, and SSIM is currently implemented for a video gamma 0 to 255 range (255 being the white peak signa). As mentioned above, metrics applied in video gamma yield uncertainty in physical meaning.

Extension of sigma_compare into a self-relative linear metric with some provision for the tolerances (such as local displacement or brightness offset) inherent in SSIM HVS local modelling, may be potentially useful for some applications. Many codecs have relied upon PSNR/MSE (Mean Square Error) metrics, which metrics have no displacement tolerance, and have no provision for global offsets in value (which may be invisible in practice). High quality codecs (and other processing tools) are likely to have no offsets nor displacements, such that sigma_compare is appropriate. For other codecs (perhaps not aiming for the highest quality), a more flexible quality metric might be more appropriate.

Conclusion

It is hoped that the more complete characterization that is provided by sigma_compare will gradually come to take the place of previous tools such as PSNR as codecs are applied to HDR scenes. The use of the self-relative measure on linear half-float data provides a way to understand and evaluate codec behavior quantitatively.

Acknowledgements

Thanks to Bill Mandel and Chad Fogg for initial testing of sigma_compare. Thanks to Don Eklund for pointing out the confusing landscape of OETF/EOTF non-linear transforms being tested in HDR with integer codecs.

Journal Articles

Gary Demos, “Layered Motion Compensation for Moving Image Compression” SMPTE Journal, Jan/Feb 2009.

Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, Eero P. Simoncelli, “Image Quality Assesment: From Error Visibility to Structural Similarity” IEEE Transactions on Image Processing, Vol 13, No 4, April 2004.

Mohammed Hassan, Chakravarthy Bhagvati, “Structural Similarity Measure for Color Images” International Journal of Computer Applications, Vol 43, No 14, April 2012

Articles Published in Proceedings

G. Demos, “Filtering in a High Dynamic Range (HDR) Context,” *Proc. SMPTE*. October 2013.

Appendix

```
/* sigma compare, author Gary Demos June 2014 */

/* no warranties expressed nor implied */

/* no representation is herein made as to usefulness nor suitability for any purpose */

/* caution, code may contain bugs and design flaws */

/* use at your own risk */


// to build:

/*


g++ sigma_compare.cpp dpx_file_io.cpp -o sigma_compare_linux -lpthread \

 -I /usr/local/include/OpenEXR \

 /usr/local/lib/libHalf.a /usr/local/lib/libIlmImf.a /usr/local/lib/libIex.a /usr/local/lib/libIlmThread.a \

 /usr/local/lib/libz.a -m64 -O1


*/


#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/stat.h>

#include <math.h>


#include <OpenEXR/ImfRgbaFile.h> /* OpenExr */

#include <OpenEXR/ImfArray.h> /* OpenExr */


using namespace Imf; /* OpenExr */

using namespace Imath; /* OpenExr */


#define MAX(a,b) (((a) > (b)) ? (a) : (b))

#define MIN(a,b) (((a) < (b)) ? (a) : (b))


#define NUMBER_OF_STOPS 32 // note: the 5 bit half-float exponent is limited to a range of 31 stops (value 31 = +-inf nan out-of-range code).  Note: value 0-exp denorm extends lower),  Use 32-bit floating point I/O (e.g. dpx32) for ranges greater than 31 (+denorm) stops


// Prototypes:

void dpx_read (char *inname, float **pixels_read, short *width, short *height, short cineon, short half_flag);
```

```
/*************************************************************************************************************/
int
main(int argc, char **argv)
{

short x, y, c, j, k;
float *pixels = NULL, *pixels2 = NULL, *pixels3 = NULL;
short *bin_value = NULL;
char infile1[300], infile2[300];

int iframe=0, first_frame=0, last_frame=0;

short h_reso, v_reso;
float col[3];
float flt_log2;
double square_sum_col[3][NUMBER_OF_STOPS];
double average_col[3][NUMBER_OF_STOPS];
float sigma_col[3][NUMBER_OF_STOPS];
double furthest_outlier_avg[3][NUMBER_OF_STOPS];

double count_col[3][NUMBER_OF_STOPS]; /* must be double precision 64-bit floating point to continue to accumuate counts
beyond the 23-bit lsb (could also use long-long 64-bit int, but 32-bit int is insufficient) */
#define SIGMA_MULTIPLES 7 /*2,3,4,6,8,12,16*/
double count_col_multiple_sigma[SIGMA_MULTIPLES][3][NUMBER_OF_STOPS];
double count_neg_col[3];
float col_dif[3];
double average_neg_col[3];
double square_sum_neg_col[3];
float sigma_neg_col[3];
short dpx_in_file1=0, dpx_in_file2=0;
float amplitude_factor;

    Array2D<Rgba> half_float_pixels; /* unused if both input files are dpx */

if (argc <= 2) {
  printf(" usage: %s original_files, test_comparison_files, first_frame, last_frame, amplitude_factor\n", argv[0]);
  printf(" note: use %% format for frame numbers, such as %%07d, within the file names\n");
  exit(1);
}
```

```
  if (argc > 4) {
    first_frame = atoi(argv[3]);
    last_frame = atoi(argv[4]);
    printf(" first_frame = %d last_frame = %d\n", first_frame, last_frame);

  } else { /* argc <= 4 */
    printf(" no frame range, one frame will be processed\n");
  } /* argc > 4 or not */

  if (argc > 5) {
    amplitude_factor = atof(argv[5]);
    printf(" setting amplitude_factor to %f\n", amplitude_factor);
 /* can scale nits to unity relationship here
 e.g. if 1.0=100nits, use 100.0 here for the amplitude_factor in order to see the printout relative to nits (sigma and average value),
does not affect self_relative
 e.g. also if looking at nits, but wish to see in terms of 100nits = 1.0, then use .01 for amplitude_factor to range 100nits down to 1.0
(e.g. a nominal white at 1.0) */
  } else { /* argc <= 5 */
    amplitude_factor = 1.0;
    printf(" using default amplitude_factor = %f\n", amplitude_factor);
  } /* argc > 5 or not */



  for (c=0; c<3; c++) {
    for (j=0; j<NUMBER_OF_STOPS; j++) {
      square_sum_col[c][j] = 0.0;
      average_col[c][j] = 0.0;
      count_col[c][j] = 0;
      furthest_outlier_avg[c][j] = 0;
      for(k=0; k < SIGMA_MULTIPLES; k++) {
        count_col_multiple_sigma[k][c][j] = 0;
      } /* k */
    } /* j */
    square_sum_neg_col[c] = 0.0;
    count_neg_col[c] = 0;
    average_neg_col[c] = 0;
  } /* c */



 for (iframe = first_frame; iframe <= last_frame; iframe++) {
  sprintf(infile1, argv[1], iframe);
  sprintf(infile2, argv[2], iframe);

  short num_chars = strlen(infile1); /* length of in_file_name string */

  if ((!strcmp(&infile1[num_chars-1],  "x"))  ||(!strcmp(&infile1[num_chars-1],    "X")) || /* DPX file ending in ".dpx" (not recommended,
10bit linear has insufficient precision near black) */
      (!strcmp(&infile1[num_chars-3],  "x32"))||(!strcmp(&infile1[num_chars-3],  "X32")) || /* dpx32 or DPX32 */
      (!strcmp(&infile1[num_chars-3],  "x16"))||(!strcmp(&infile1[num_chars-3],  "X16")) || /* dpx16 or DPX16 (not recommended) */
      (!strcmp(&infile1[num_chars-4], "xhlf"))||(!strcmp(&infile1[num_chars-4], "XHLF"))) { /* dpxhlf or DPXHLF, 16-bit half-float (non-
standard, improperly defined in dpx 2.0 standard) */
    dpx_in_file1 = 1;
  }
  num_chars = strlen(infile2); /* length of in_file_name string */

  if ((!strcmp(&infile2[num_chars-1],  "x"))  ||(!strcmp(&infile2[num_chars-1],    "X")) || /* DPX file ending in ".dpx" (not recommended,
10bit linear has insufficient precision near black) */
      (!strcmp(&infile2[num_chars-3],  "x32"))||(!strcmp(&infile2[num_chars-3],  "X32")) || /* dpx32 or DPX32 */
      (!strcmp(&infile2[num_chars-3],  "x16"))||(!strcmp(&infile2[num_chars-3],  "X16")) || /* dpx16 or DPX16 (not recommended) */
      (!strcmp(&infile2[num_chars-4], "xhlf"))||(!strcmp(&infile2[num_chars-4], "XHLF"))) { /* dpxhlf or DPXHLF, 16-bit half-float (non-
standard, improperly defined in dpx 2.0 standard) */
    dpx_in_file2 = 1;
  }


  if (dpx_in_file2 == 1) {

    dpx_read (infile2, &pixels, &h_reso, &v_reso, 0/*not cineon*/, 0/*no half_flag*/);

    if (amplitude_factor != 1.0) {

      for (c=0; c<3; c++) {
        for(y=0; y< v_reso; y++) {
          for(x=0; x< h_reso; x++) {
            pixels[(c*v_reso + y) * h_reso + x] = amplitude_factor * pixels[(c*v_reso + y) * h_reso + x];
          } /* x */
        } /* y */
      } /* c */
    } /* amplitude_factor != 1.0 */

  } else { /* exr */
```

```
{
  RgbaInputFile file (infile2, 1);

  Box2i dw = file.dataWindow();

  h_reso = dw.max.x - dw.min.x + 1;
  v_reso = dw.max.y - dw.min.y + 1;
  half_float_pixels.resizeErase (v_reso, h_reso);

  file.setFrameBuffer (&half_float_pixels[0][0] - dw.min.x - dw.min.y * h_reso, 1, h_reso);
  file.readPixels (dw.min.y, dw.max.y);

  printf(" reading target comparison file %s having h_reso = %d v_reso = %d\n", infile2, h_reso, v_reso);

  if (pixels == NULL) { pixels = (float *) malloc(h_reso * v_reso * 12); /* 4-bytes/float * 3-colors */ }

  for(y=0; y< v_reso; y++) {
    for(x=0; x< h_reso; x++) {
      pixels[(0*v_reso + y) * h_reso + x] = amplitude_factor * half_float_pixels[y][x].r;
      pixels[(1*v_reso + y) * h_reso + x] = amplitude_factor * half_float_pixels[y][x].g;
      pixels[(2*v_reso + y) * h_reso + x] = amplitude_factor * half_float_pixels[y][x].b;
    } /* x */
  } /* y */
}
  } /* dpx vs exr */
/**************************************************************************************************/

  if (dpx_in_file1 == 1) {
short h_reso_b, v_reso_b;

    dpx_read (infile1, &pixels3, &h_reso_b, &v_reso_b, 0/*not cineon*/, 0/*no half_flag*/);

    if ((h_reso_b != h_reso) || (v_reso_b != v_reso)) {
      printf(" error, file resolutions do not match, infile1 h_reso = %d v_reso = %d, infile2 h_reso = %d v_reso = %d, aborting\n",
        h_reso, v_reso, h_reso_b, v_reso_b);
      exit(1);
    }

    if (amplitude_factor != 1.0) {
      for (c=0; c<3; c++) {
        for(y=0; y< v_reso; y++) {
         for(x=0; x< h_reso; x++) {
           pixels3[(c*v_reso + y) * h_reso + x] = amplitude_factor * pixels3[(c*v_reso + y) * h_reso + x];
         } /* x */
        } /* y */
      } /* c */
    } /* amplitude_factor != 1.0 */

  } else { /* exr */

{
  RgbaInputFile file (infile1, 1);

  Box2i dw = file.dataWindow();

short h_reso_b, v_reso_b;
  h_reso_b = dw.max.x - dw.min.x + 1;
  v_reso_b = dw.max.y - dw.min.y + 1;

  if ((h_reso_b != h_reso) || (v_reso_b != v_reso)) {
    printf(" error, file resolutions do not match, infile1 h_reso = %d v_reso = %d, infile2 h_reso = %d v_reso = %d, aborting\n",
      h_reso, v_reso, h_reso_b, v_reso_b);
    exit(1);
  }

  half_float_pixels.resizeErase (v_reso, h_reso);

  file.setFrameBuffer (&half_float_pixels[0][0] - dw.min.x - dw.min.y * h_reso, 1, h_reso);
  file.readPixels (dw.min.y, dw.max.y);

  printf(" reading original file %s having h_reso = %d v_reso = %d\n", infile1, h_reso, v_reso);
}
  } /* dpx vs exr file1 */

  if (pixels2 == NULL) { pixels2 = (float *) malloc(h_reso * v_reso * 12); /* 4-bytes/float * 3-colors */ }
  if (bin_value == NULL) { bin_value = (short *) malloc(h_reso * v_reso * 6); /* 2-bytes/short * 3-colors */ }


  for (c=0; c<3; c++) {
    for(y=0; y< v_reso; y++) {
      for(x=0; x< h_reso; x++) {
        if (dpx_in_file1 == 1) {
          col[c] = pixels3[(c*v_reso + y) * h_reso + x];
        } else { /* exr */
          if (c==0) { col[c] = amplitude_factor * half_float_pixels[y][x].r; }
          if (c==1) { col[c] = amplitude_factor * half_float_pixels[y][x].g; }
          if (c==2) { col[c] = amplitude_factor * half_float_pixels[y][x].b; }
        } /* dpx vs exr */
/* store difference from original in pixels2 */
        col_dif[c] = col[c] - pixels[(c*v_reso + y) * h_reso + x];
        pixels2[(c*v_reso + y) * h_reso + x] = col_dif[c];

        if (col[c] < 0.0) {
          square_sum_neg_col[c] = square_sum_neg_col[c] + col_dif[c] * col_dif[c];
          average_neg_col[c] = average_neg_col[c] + col[c];
          count_neg_col[c] = count_neg_col[c] + 1;
          bin_value[(c*v_reso + y) * h_reso + x] = -1;
        } else { /* col[c] >= 0 */
          flt_log2 = log2f(MAX(1e-16, col[c]));
          j = flt_log2; /* float to int */
          if (flt_log2 > 0.0) j = j + 1; /* must round one side of zero for continuity of integer j */
          j = MIN(NUMBER_OF_STOPS - 1, MAX(0, j + NUMBER_OF_STOPS / 2)); /* split NUMBER_OF_STOPS half above 1.0
and half below 1.0 */
          square_sum_col[c][j] = square_sum_col[c][j] + col_dif[c] * col_dif[c];
          average_col[c][j] = average_col[c][j] + col[c];
          count_col[c][j] = count_col[c][j] + 1;
          bin_value[(c*v_reso + y) * h_reso + x] = j;
        } /* col[c] < 0 or not */
      } /* x */
    } /* y */
  } /* c */

  if (dpx_in_file1 == 1) {
    free(pixels3);
  } /* dpx */
  if (dpx_in_file2 == 1) {
    free(pixels);
  } /* dpx */

  } /* iframe */
```

```
 printf("\n amplitude_factor = %f\n", amplitude_factor);


  for (c=0; c<3; c++) {
printf("\n");
    if (count_neg_col[c] > 0) {
      square_sum_neg_col[c] = square_sum_neg_col[c] / count_neg_col[c];
      sigma_neg_col[c] = sqrtf(square_sum_neg_col[c]);
      average_neg_col[c] = average_neg_col[c] / count_neg_col[c];
      if (c==0) { printf(" sigma_neg_red = %e self_relatve = %f (%f%%) at average value = %e for %.0f pixels\n",
        sigma_neg_col[c], sigma_neg_col[c] / average_neg_col[c], 100.0 *sigma_neg_col[c] / average_neg_col[c],
average_neg_col[c], count_neg_col[c]); }
      if (c==1) { printf(" sigma_neg_grn = %e self_relatve = %f (%f%%) at average value = %e for %.0f pixels\n",
        sigma_neg_col[c], sigma_neg_col[c] / average_neg_col[c], 100.0 *sigma_neg_col[c] / average_neg_col[c],
average_neg_col[c], count_neg_col[c]); }
      if (c==2) { printf(" sigma_neg_blu = %e self_relatve = %f (%f%%) at average value = %e for %.0f pixels\n",
        sigma_neg_col[c], sigma_neg_col[c] / average_neg_col[c], 100.0 *sigma_neg_col[c] / average_neg_col[c],
average_neg_col[c], count_neg_col[c]); }
    } /* count_neg_col[c] > 0 */


    for (j=0; j<NUMBER_OF_STOPS; j++) {
      if (count_col[c][j] > 0) {
        square_sum_col[c][j] = square_sum_col[c][j] / count_col[c][j];
        sigma_col[c][j] = sqrtf(square_sum_col[c][j]);
        average_col[c][j] = average_col[c][j]  / count_col[c][j];
//          if (j == 0) { average_col[c][j] = MAX(1e-16, average_col[c][j]); /* prevent divide by zero */ }
        if (c==0) { printf(" sigma_red[%d] = %e self_relative = %f (%f%%) at average value = %e for %.0f pixels\n",
          j, sigma_col[c][j], sigma_col[c][j] / average_col[c][j], 100.0 * sigma_col[c][j] / average_col[c][j], average_col[c][j],
count_col[c][j]); }
        if (c==1) { printf(" sigma_grn[%d] = %e self_relative = %f (%f%%) at average value = %e for %.0f pixels\n",
          j, sigma_col[c][j], sigma_col[c][j] / average_col[c][j], 100.0 * sigma_col[c][j] / average_col[c][j], average_col[c][j],
count_col[c][j]); }
        if (c==2) { printf(" sigma_blu[%d] = %e self_relative = %f (%f%%) at average value = %e for %.0f pixels\n",
          j, sigma_col[c][j], sigma_col[c][j] / average_col[c][j], 100.0 * sigma_col[c][j] / average_col[c][j], average_col[c][j],
count_col[c][j]); }
      } /* count_col[c][j] > 0 */
    } /* j */
  } /* c */


printf("\n beginning second pass over frames (multiples of sigma, now that sigma for all frames has been determined)\n\n");


  for (iframe = first_frame; iframe <= last_frame; iframe++) {
    sprintf(infile1, argv[1], iframe);
    sprintf(infile2, argv[2], iframe);
  if (dpx_in_file2 == 1) {

    dpx_read (infile2, &pixels, &h_reso, &v_reso, 0/*not cineon*/, 0/*no half_flag*/);

    if (amplitude_factor != 1.0) {

      for (c=0; c<3; c++) {
        for(y=0; y< v_reso; y++) {
          for(x=0; x< h_reso; x++) {
            pixels[(c*v_reso + y) * h_reso + x] = amplitude_factor * pixels[(c*v_reso + y) * h_reso + x];
          } /* x */
        } /* y */
      } /* c */
    } /* amplitude_factor != 1.0 */

  } else { /* exr */

{
  RgbaInputFile file (infile2, 1);

  Box2i dw = file.dataWindow();

  half_float_pixels.resizeErase (v_reso, h_reso);

  file.setFrameBuffer (&half_float_pixels[0][0] - dw.min.x - dw.min.y * h_reso, 1, h_reso);
  file.readPixels (dw.min.y, dw.max.y);

  printf(" reading target comparison file %s having h_reso = %d v_reso = %d\n", infile2, h_reso, v_reso);

  for(y=0; y< v_reso; y++) {
    for(x=0; x< h_reso; x++) {
      pixels[(0*v_reso + y) * h_reso + x] = amplitude_factor * half_float_pixels[y][x].r;
      pixels[(1*v_reso + y) * h_reso + x] = amplitude_factor * half_float_pixels[y][x].g;
      pixels[(2*v_reso + y) * h_reso + x] = amplitude_factor * half_float_pixels[y][x].b;
    } /* x */
  } /* y */
}
  } /* dpx vs exr */
```

```
/*************************************************************************************************/

    if (dpx_in_file1 == 1) {
short h_reso_b, v_reso_b;

      dpx_read (infile1, &pixels3, &h_reso_b, &v_reso_b, 0/*not cineon*/, 0/*no half_flag*/);

      if (amplitude_factor != 1.0) {
        for (c=0; c<3; c++) {
          for(y=0; y< v_reso; y++) {
            for(x=0; x< h_reso; x++) {
              pixels3[(c*v_reso + y) * h_reso + x] = amplitude_factor * pixels3[(c*v_reso + y) * h_reso + x];
            } /* x */
          } /* y */
        } /* c */
      } /* amplitude_factor != 1.0 */

    } else { /* exr */

{
    RgbaInputFile file (infile1, 1);

    Box2i dw = file.dataWindow();

    half_float_pixels.resizeErase (v_reso, h_reso);

    file.setFrameBuffer (&half_float_pixels[0][0] - dw.min.x - dw.min.y * h_reso, 1, h_reso);
    file.readPixels (dw.min.y, dw.max.y);

    printf(" reading original file %s having h_reso = %d v_reso = %d\n", infile1, h_reso, v_reso);
}
  } /* dpx vs exr file1 */



/* recompute pixels2 and bin_value */
    for (c=0; c<3; c++) {
      for(y=0; y< v_reso; y++) {
        for(x=0; x< h_reso; x++) {
          if (dpx_in_file1 == 1) {
            col[c] = pixels3[(c*v_reso + y) * h_reso + x];
          } else { /* exr */
            if (c==0) { col[c] = amplitude_factor * half_float_pixels[y][x].r; }
            if (c==1) { col[c] = amplitude_factor * half_float_pixels[y][x].g; }
            if (c==2) { col[c] = amplitude_factor * half_float_pixels[y][x].b; }
          } /* dpx vs exr */
/* store difference from original in pixels2 */
          col_dif[c] = col[c] - pixels[(c*v_reso + y) * h_reso + x];
          pixels2[(c*v_reso + y) * h_reso + x] = col_dif[c];

          if (col[c] < 0.0) {
            bin_value[(c*v_reso + y) * h_reso + x] = -1;
          } else { /* col[c] >= 0 */
            flt_log2 = log2f(MAX(1e-16, col[c]));
            j = flt_log2; /* float to int */
            if (flt_log2 > 0.0) j = j + 1; /* must round one side of zero for continuity of integer j */
            j = MIN(NUMBER_OF_STOPS - 1, MAX(0, j + NUMBER_OF_STOPS / 2)); /* split NUMBER_OF_STOPS half above 1.0
and half below 1.0 */
            bin_value[(c*v_reso + y) * h_reso + x] = j;
          } /* col[c] < 0 or not */
        } /* x */
      } /* y */
    } /* c */



    for (c=0; c<3; c++) {
      for(y=0; y< v_reso; y++) {
        for(x=0; x< h_reso; x++) {

          if (bin_value[(c*v_reso + y) * h_reso + x] >= 0) { /* dont count negs */
            j = bin_value[(c*v_reso + y) * h_reso + x];
            if (count_col[c][j] > 0) {
              col_dif[c] = pixels2[(c*v_reso + y) * h_reso + x];

              for(k=0; k < SIGMA_MULTIPLES; k++) {
short multiple;
                multiple = (1 << ((k>>1) + 1));
                multiple = multiple + (k & 1) * (multiple >> 1); /* turns k=0,1,2,3,4,5,6,7 into multiple=2,3,4,6,8,12,16 */
                if (fabsf(col_dif[c]) > (multiple * sigma_col[c][j])) {
                  count_col_multiple_sigma[k][c][j] = count_col_multiple_sigma[k][c][j] + 1;

                  if(k== (SIGMA_MULTIPLES-1)) {
```

```
//  printf(" c=%d j=%d x=%d y=%d col_dif[%d]=%e\n", c, j, x, y, c, col_dif[c]);
                    furthest_outlier_avg[c][j] = furthest_outlier_avg[c][j] + fabsf(col_dif[c]);
                  } /* k== (SIGMA_MULTIPLES-1) */
                } /* fabsf(col_dif[c]) > (multiple * sigma_col[c][j]) */
              } /* k */
            } /* count_col[c][j] > 0 */
          } /* bin_value[(c*v_reso + y) * h_reso + x] >= 0 */

        } /* x */
      } /* y */
    } /* c */

    if (dpx_in_file1 == 1) {
      free(pixels3);
    } /* dpx */
    if (dpx_in_file2 == 1) {
      free(pixels);
    } /* dpx */

   } /* iframe */



   printf("\n amplitude_factor = %f\n", amplitude_factor);


    for (c=0; c<3; c++) {
printf("\n");
      if (count_neg_col[c] > 0) {
        if (c==0) { printf(" sigma_neg_red = %e self_relatve = %f (%f%%) at average value = %e for %.0f pixels\n",
          sigma_neg_col[c], sigma_neg_col[c] / average_neg_col[c], 100.0 *sigma_neg_col[c] / average_neg_col[c],
average_neg_col[c], count_neg_col[c]); }
        if (c==1) { printf(" sigma_neg_grn = %e self_relatve = %f (%f%%) at average value = %e for %.0f pixels\n",
          sigma_neg_col[c], sigma_neg_col[c] / average_neg_col[c], 100.0 *sigma_neg_col[c] / average_neg_col[c],
average_neg_col[c], count_neg_col[c]); }
        if (c==2) { printf(" sigma_neg_blu = %e self_relatve = %f (%f%%) at average value = %e for %.0f pixels\n",
          sigma_neg_col[c], sigma_neg_col[c] / average_neg_col[c], 100.0 *sigma_neg_col[c] / average_neg_col[c],
average_neg_col[c], count_neg_col[c]); }
      } /* count_neg_col[c] > 0 */



      for (j=0; j<NUMBER_OF_STOPS; j++) {
        if (count_col[c][j] > 0) {
          if (c==0) { printf(" sigma_red[%d] = %e self_relative = %f (%f%%) at average value = %e for %.0f pixels\n",
            j, sigma_col[c][j], sigma_col[c][j] / average_col[c][j], 100.0 * sigma_col[c][j] / average_col[c][j], average_col[c][j],
count_col[c][j]); }
          if (c==1) { printf(" sigma_grn[%d] = %e self_relative = %f (%f%%) at average value = %e for %.0f pixels\n",
            j, sigma_col[c][j], sigma_col[c][j] / average_col[c][j], 100.0 * sigma_col[c][j] / average_col[c][j], average_col[c][j],
count_col[c][j]); }
          if (c==2) { printf(" sigma_blu[%d] = %e self_relative = %f (%f%%) at average value = %e for %.0f pixels\n",
            j, sigma_col[c][j], sigma_col[c][j] / average_col[c][j], 100.0 * sigma_col[c][j] / average_col[c][j], average_col[c][j],
count_col[c][j]); }
        } /* count_col[c][j] > 0 */
      } /* j */
    } /* c */



short multiple;
    for (c=0; c<3; c++) {
printf("\n");
      for (j=0; j<NUMBER_OF_STOPS; j++) {
        if ((count_col[c][j] > 0) && (count_col_multiple_sigma[0][c][j] > 0)) {
          if (c==0) { printf("  sigma_red[%d] = %e self_relative = %f (%f%%) at average value = %e for %.0f pixels\n",
            j, sigma_col[c][j], sigma_col[c][j] / average_col[c][j], 100.0 * sigma_col[c][j] / average_col[c][j], average_col[c][j],
count_col[c][j]); }
          if (c==1) { printf("  sigma_grn[%d] = %e self_relative = %f (%f%%) at average value = %e for %.0f pixels\n",
            j, sigma_col[c][j], sigma_col[c][j] / average_col[c][j], 100.0 * sigma_col[c][j] / average_col[c][j], average_col[c][j],
count_col[c][j]); }
          if (c==2) { printf("  sigma_blu[%d] = %e self_relative = %f (%f%%) at average value = %e for %.0f pixels\n",
            j, sigma_col[c][j], sigma_col[c][j] / average_col[c][j], 100.0 * sigma_col[c][j] / average_col[c][j], average_col[c][j],
count_col[c][j]); }
        } /* count_col[c][j] > 0 */
        for(k=0; k < (SIGMA_MULTIPLES-1); k++) {
          if ((count_col_multiple_sigma[k][c][j] > 0) && (count_col_multiple_sigma[k][c][j] > count_col_multiple_sigma[k+1][c][j])) {
            multiple = (1 << ((k>>1) + 1));
            multiple = multiple + (k & 1) * (multiple >> 1); /* turns k=0,1,2,3,4,5,6,7 into multiple=2,3,4,6,8,12,16 */
            if (c==0) { printf(" %dsigma_red[%d] = %e self_relative = %f (%f%%) at average value = %e for %.0f pixels which is %f %%
of the %.0f pixels within a stop of this value\n",
              multiple, j, multiple*sigma_col[c][j], multiple*sigma_col[c][j] / average_col[c][j], 100.0 * multiple*sigma_col[c][j] /
average_col[c][j], average_col[c][j], count_col_multiple_sigma[k][c][j], (count_col_multiple_sigma[k][c][j] * 100.0)/(1.0 *
count_col[c][j]), count_col[c][j]); }
            if (c==1) { printf(" %dsigma_grn[%d] = %e self_relative = %f (%f%%) at average value = %e for %.0f pixels which is %f %%
of the %.0f pixels within a stop of this value\n",
              multiple, j, multiple*sigma_col[c][j], multiple*sigma_col[c][j] / average_col[c][j], 100.0 * multiple*sigma_col[c][j] /
average_col[c][j], average_col[c][j], count_col_multiple_sigma[k][c][j], (count_col_multiple_sigma[k][c][j] * 100.0)/(1.0 *
count_col[c][j]), count_col[c][j]); }
            if (c==2) { printf(" %dsigma_blu[%d] = %e self_relative = %f (%f%%) at average value = %e for %.0f pixels which is %f %%
of the %.0f pixels within a stop of this value\n",
              multiple, j, multiple*sigma_col[c][j], multiple*sigma_col[c][j] / average_col[c][j], 100.0 * multiple*sigma_col[c][j] /
average_col[c][j], average_col[c][j], count_col_multiple_sigma[k][c][j], (count_col_multiple_sigma[k][c][j] * 100.0)/(1.0 *
count_col[c][j]), count_col[c][j]); }

          } /* count_col_multiple_sigma[k][c][j] > 0 */

        } /* k */

        k = SIGMA_MULTIPLES-1;

        if (count_col_multiple_sigma[k][c][j] > 0) {

            furthest_outlier_avg[c][j] = furthest_outlier_avg[c][j] / count_col_multiple_sigma[k][c][j]; /* compute average from sum */

            multiple = (1 << ((k>>1) + 1));

            multiple = multiple + (k & 1) * (multiple >> 1); /* turns k=0,1,2,3,4,5,6,7 into multiple=2,3,4,6,8,12,16 */

            if (c==0) { printf(" beyond %dsigma_red[%d] furthest_outlier_avg dif = %e which is %f sigma (sigma being %e), self_relative
= %f (%f%%) at average value = %e for %.0f pixels which is %f %% of the %.0f pixels within a stop of this value\n",

              multiple, j, furthest_outlier_avg[c][j], furthest_outlier_avg[c][j] / sigma_col[c][j], sigma_col[c][j], furthest_outlier_avg[c][j] /
average_col[c][j], 100.0 * furthest_outlier_avg[c][j] / average_col[c][j], average_col[c][j], count_col_multiple_sigma[k][c][j],
(count_col_multiple_sigma[k][c][j] * 100.0)/(1.0 * count_col[c][j]), count_col[c][j]); }

            if (c==1) { printf(" beyond %dsigma_grn[%d] furthest_outlier_avg dif = %e which is %f sigma (sigma being %e), self_relative
= %f (%f%%) at average value = %e for %.0f pixels which is %f %% of the %.0f pixels within a stop of this value\n",

              multiple, j, furthest_outlier_avg[c][j], furthest_outlier_avg[c][j] / sigma_col[c][j], sigma_col[c][j], furthest_outlier_avg[c][j] /
average_col[c][j], 100.0 * furthest_outlier_avg[c][j] / average_col[c][j], average_col[c][j], count_col_multiple_sigma[k][c][j],
(count_col_multiple_sigma[k][c][j] * 100.0)/(1.0 * count_col[c][j]), count_col[c][j]); }

            if (c==2) { printf(" beyond %dsigma_blu[%d] furthest_outlier_avg dif = %e which is %f sigma (sigma being %e), self_relative
= %f (%f%%) at average value = %e for %.0f pixels which is %f %% of the %.0f pixels within a stop of this value\n",

              multiple, j, furthest_outlier_avg[c][j], furthest_outlier_avg[c][j] / sigma_col[c][j], sigma_col[c][j], furthest_outlier_avg[c][j] /
average_col[c][j], 100.0 * furthest_outlier_avg[c][j] / average_col[c][j], average_col[c][j], count_col_multiple_sigma[k][c][j],
(count_col_multiple_sigma[k][c][j] * 100.0)/(1.0 * count_col[c][j]), count_col[c][j]); }

        } /* (count_col_multiple_sigma[k][c][j] > 0) */

      } /* j */

    } /* c */



} /* main */
```