August 2020

Transforming Color Space With Arbitrary Matrix

Pascal Massimino

Maryla Isuka Waclawa Ustarroz-Calonge

TRANSFORMING COLOR SPACE WITH ARBITRARY MATRIX

Abstract: In image compression, and encoder can transform representations of pixels and/or samples or color samples, such as by performing principal component analysis (PCA) on the pixels to generate a transformation matrix, compress the transformed pixels and/or encode the samples, and transmit the compressed pixels to a decoder. The decoder can decompress the pixels and/or decode the samples received from the encoder, and transform the pixels. The decoder can transform the pixels with an inverse matrix of the transformation matrix or any arbitrary matrix. The pixels can represent a color space either before or after transformation.

Image compression can include transforming pixels, which can be considered samples and/or color samples, from a red, green, blue (RGB) representation to another color space. The transformation can be performed by principal component analysis (PCA), which reduces the number of bits needed to represent pixels in images that are "tinted" toward a specific color (such as sepia), and/or in which a single color is dominant within the image. In some examples, the image can include multiple clusters of pixels, with each pixel in the respective cluster being tinted toward a dominant color of the respective cluster. In the examples of multiple clusters, the transformation can be different for pixels in each of the different clusters. An encoder can compress the transformed pixels, and send the transformed pixels to a receiving entity, which can be considered a decoder. Compressing the pixels can also be considered to be coding the samples, such as with a dictionary or alphabet.

The receiving entity and/or decoder can decompress the compressed, transformed pixels. Decompressing the pixels can also be considered to be deciding the samples. The receiving entity and/or decoder can transform the decompressed, transformed pixels using a same algorithm, or using a different algorithm for each cluster. In some examples, the receiving entity and/or decoder can perform arbitrary transformations on pixels in each cluster. The receiving entity and/or decoder can output and/or display the pixels after performing the transformations.

FIG. 1 shows a pipeline of operations performed on an image. The pipeline can include an encoder and/or transmitting entity (referred to herein as an encoder) performing pixel transformation (102) on pixels in the image. The transformation can include performing PCA to generate a transformation matrix.

FIG. 2 shows a transformation matrix 202 and an offset vector 204. The transformation matrix 202 can include a three-by-three matrix that the encoder and/or transmitting entity multiplies by a three-by-one vector representing the RGB values (or other color representation system and/or color space such as YCbCr or YCoCg) to generate a new three-by-one vector representing the pixel. The encoder entity can add the offset vector 204 to the new three-by-one vector to generate a transformed pixel. The addition of the offset vector 204 can reduce the magnitudes of the values in the new three-by-one vectors, allowing the encoder to represent the pixels using fewer bits. In some examples, the transformation matrix 202 and offset vector 204 can be included in a single three-by-four matrix. The transformation matrix 202 and offset vector 204 can transform the bits from their internal representations to displayable samples. The displayable samples can be red green and blue (RGB) values, or YCbCR values.

In some examples, the encoder can perform different transformations on different pixels. The encoder can determine multiple color spaces based on multiple colors toward which portions of an image are tinted.

FIG. 3A shows an image 300 with three bottles 302, 304, 306 tinted toward different colors. In this example, a blue bottle 302 included in the image 300 is tinted toward the color blue, a red bottle 304 included in the image 300 is tinted toward the color red, and a yellow bottle 306 included in the image 300 is tinted toward the color yellow. The encoder can

transform pixels representing each of the blue bottle 302, the red bottle 304, and the yellow bottle 306 using different matrices.

FIG. 3B shows an RGB cube 350 with nodes representing pixels included in the image 300 of FIG. 3B. The RGB cube 350 can represent a color space in which pixels are represented by red (R), green (G), and blue (B) values. The pixels representing the blue bottle 302 in the image 300 form a blue cluster 352 in the RGB cube 350, the pixels representing the red bottle 304 in the image 300 form a red cluster 354 in the RGB cube 350, and the pixels representing the yellow bottle 306 in the image 300 form a yellow cluster 356 in the RGB cube 350. The encoder can recognize the clusters 352, 354, 356 by performing a nearest-neighbor clustering algorithm, for example, or any other clustering algorithm. The offset vector 204 can point to a center of mass for a respective cluster 352, 354, 356.

FIG. 4A shows an image 400 that is tinted toward orange. In an example, the encoder can generate a transformation matrix 202 and/or offset vector 204 based on the orange tint of the image 400. The encoder can perform the transformation on the pixels into a custom color space based on the generated transformation matrix 202 and/or offset vector 204, and compress the transformed pixels.

FIG. 4B shows a compression graph 450 showing compression gains for the image 400. As shown in the compression graph 450, the compression gains for the custom color space based on the orange tint of the image 400 are greater than compression gains for a classic YCbCr color space.

Returning to FIGs. 1 and 2, the encoder can generate a transformation matrix 202 and offset vector 204 for each cluster 352, 354, 356 corresponding to a color toward which a portion of the image 300 is tinted. The encoder can generate the transformation matrix 202 and offset

vector 204 for each cluster 352, 354, 356 based, for example, on performing principal component analysis (PCA). The encoder can transform each pixel based on the transformation matrix for the cluster of which the respective pixel is a member. The encoder can determine whether to apply the transformation matrix 202 based on a determination of whether the transformation matrix 202 is a good fit for the image 300, 400, such as based on an autocorrelation and/or covariance of the pixels included in the image 300, 400 before and after the compression.

The encoder can compress (104) the transformed pixels. The encoder can compress the transformed pixels by any of various compression methods, such as transform coding, run-length encoding, or entropy encoding, as non-limiting examples. The encoder can also encode the transformation matrix(es) and/or offset vector(s).

The encoder can transmit (106) the compressed, transformed pixels, transformation matrix(es) 202, and/or offset vector(s) 204. The transformation matrix(es) 202, and/or offset vector(s) 204 can be signaled in a bitstream with fixed precision, with constraints such as the first entry of each column in the transformation matrix 202 being positive. The bitstream can include labels indicating which cluster each pixel belongs to, and/or which transformation should be performed on each pixel.

In some examples, the encoder can transmit only five out of the nine values in the transformation matrix 202, reducing the number of bits to be transmitted. The encoder can perform principal component analysis to find a custom color space (such one luma-plane and two chroma-planes) from RGB pixel samples. This can result in an orthogonal three-by-three transformation matrix in which all column-vectors and row-vectors have the same magnitude, the first row and first column contain only positive values, column-vectors are mutually orthogonal, and row-vectors are mutually orthogonal. This transformation matrix 202 can be

used to transform samples in the custom color space to RGB and is represented by the A to I letters shown in FIG. 2A.

These elements can be considered fixed-point integers (twelve-bit precision and sixteen-bit maximum values). In the matrices below, a bracketed value indicates that the value is not yet available, italics indicate that only the sign is missing, and standard typeface indicates that the value is completely signaled/deduced.

The first step is to signal the maximum number of bits N used by each of A, B, C, D, and E. A, B and C, which are positive numbers, are signaled using N bits each. The ABC vector of length L, that will be used in some of the constraints below, is obtained by calculating the square root of (A*A+B*B+C*C). Another option would be to signal A, B, L, and deduce C from A, B, and L. At this point A, B, and C are known, but D, E, F, G, H, and I are not yet available:

A B C
[D] [E] [F]
[G] [H] [I]

Since the magnitude of ADG is known to be L, and A is available, the maximum value of D can be estimated (by assuming that G is zero). Either this maximum or N can be used to signal the bounded D:

A B C
D [E] [F]
[G] [H] [I]

The same process can be performed for the absolute value of E, but both BEH and DEF are used to estimate the maximum value (still assuming that H and F are zero):

A B C
D *E* [F]
[G] [H] [I]

Using the magnitude L, the absolute values of F, G and H are deduced from DEF, ADG, and BEH, respectively. Then using the same technique, I is computed from CFI and GHI (taking the average of the two results):

A	B	C
D	*E*	*F*
G	*H*	*I*

Due to the orthogonality of row-vectors, the element generating the highest sub-dot-product has an opposite sign compared to the two others. For example, by considering the vector BEH and its orthogonal ADG, the signs of E and H can be deduced with the following possibilities: |A*B| > |D*E| and |A*B| > |G*H|, therefore E and H are negatives; |A*B| > |D*E| and |A*B| < |G*H|, therefore E is positive and H is negative; or |A*B| < |D*E| and |A*B| > |G*H|, therefore E is negative and H is positive. The matrix is then as follows:

A	B	C
D	E	*F*
G	H	*I*

The same method is applied to CFI compared to ADG to obtain the signs of F and I:

A	B	C
D	E	*F*
G	H	*I*

Rounding errors can occur, especially with fixed point computation, so the decoded result may not be exactly what was encoded. To compensate for this difference, the encoder can either adapt the original matrix so that the matrix encodes without loss, or use a few more bits to correct the deduced values with delta-coding (correct F,G,H,I and loosen the maximum values of D,E). Pseudocode for these operations is included at the end of this publication.

A receiving entity and/or decoder (referred to as a decoder herein) can receive the compressed, transformed pixels, transformation matrix(es) 202, and/or offset vector(s) 204. In

the example in which the encoder transmits only five out of the nine values in the transformation matrix 202, the decoder can deduce the remaining four values of the transformation matrix 202.

The decoder can decompress (108) the compressed, transformed pixels, transformation matrix(es), and/or offset vector(s). The decoder can decompress the compressed, transformed pixels, transformation matrix(es), and/or offset vector(s) using a decompression method corresponding to the compression method that the encoder used to compress the transformed pixels, transformation matrix(es), and/or offset vector(s).

The decoder can perform a reverse transformation (110) on the decompressed, transformed pixels. In some examples, the decoder can subtract the offset vector 204 from the decompressed, transformed pixels. After subtracting the offset vector 204 from the decompressed, transformed pixels, the decoder can multiply the decompressed, transformed pixels by a matrix. In some examples, the matrix can be an inverse of the transformation matrix 202. However, the decoder can multiply the decompressed, transformed pixels by any arbitrary matrix. The multiplication of the decompressed, transformed pixels by the matrix can generate pixels in either the original RGB color space or a new color space. The decoder can present the pixels via a display, or pass the pixels on to another entity for display.

PSEUDOCODE

```
signal_int(N)
signal_int(A, max_num_bits=N)
signal_int(B, max_num_bits=N)
signal_int(C, max_num_bits=N)
L = sqrt(A*A + B*B + C*C)
max_D = sqrt(L*L - A*A)
if (max_D < 2^N)
     signal_int(D, max_value=max_D)
else
     signal_int(D, max_num_bits=N)
max_E = (sqrt(L*L - B*B) + sqrt(L*L - D*D)) / 2
if (max_E < 2^N)
```

```
      signal_int(E, max_value=max_E)
else
      signal_int(E, max_num_bits=N)
F = sqrt(L*L - D*D - E*E)
G = sqrt(L*L - A*A - D*D)
H = sqrt(L*L - B*B - E*E)
I = (sqrt(L*L - C*C - F*F) + sqrt(L*L - G*G - H*H)) / 2
if (D*E < max(A*B, G*H))
      E = -E
if (G*H < max(A*B, abs(D*E)))
      H = -H
if (D*F < max(A*C, G*I))
      F = -F
if (G*I < max(A*C, abs(D*F)))
      I = -I
```

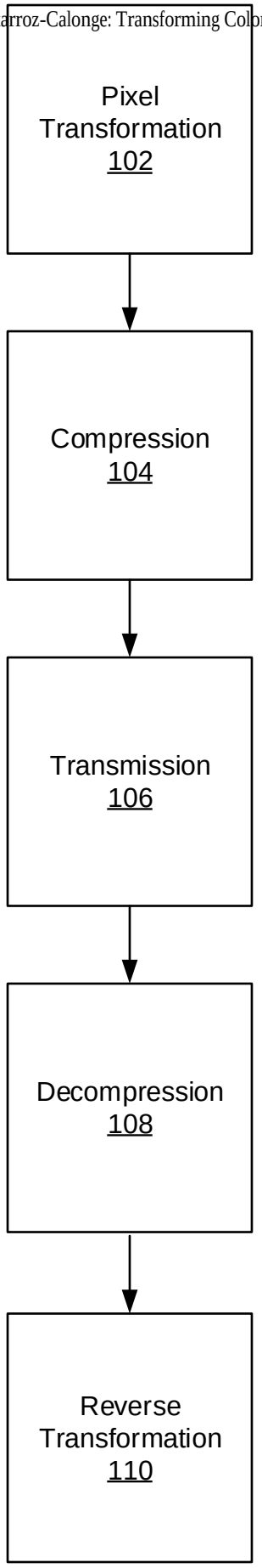
Pixel Transformation 102
Compression 104
Transmission 106
Decompression 108
Reverse Transformation 110

FIG. 1

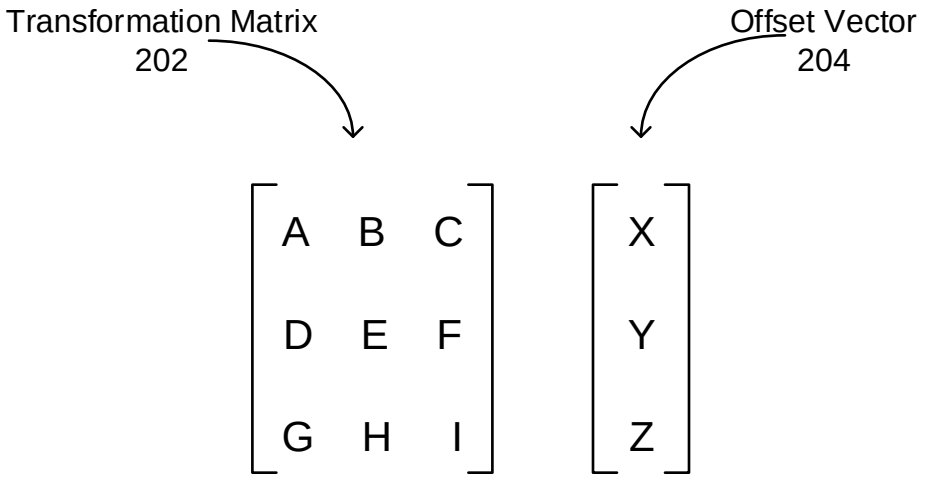
Transformation Matrix
202
Offset Vector
204
A B C
D E F
G H I
X
Y
Z

FIG. 2


Image
300
Red
Bottle
304
Blue
Bottle
302
Yellow
Bottle
306

FIG. 3A

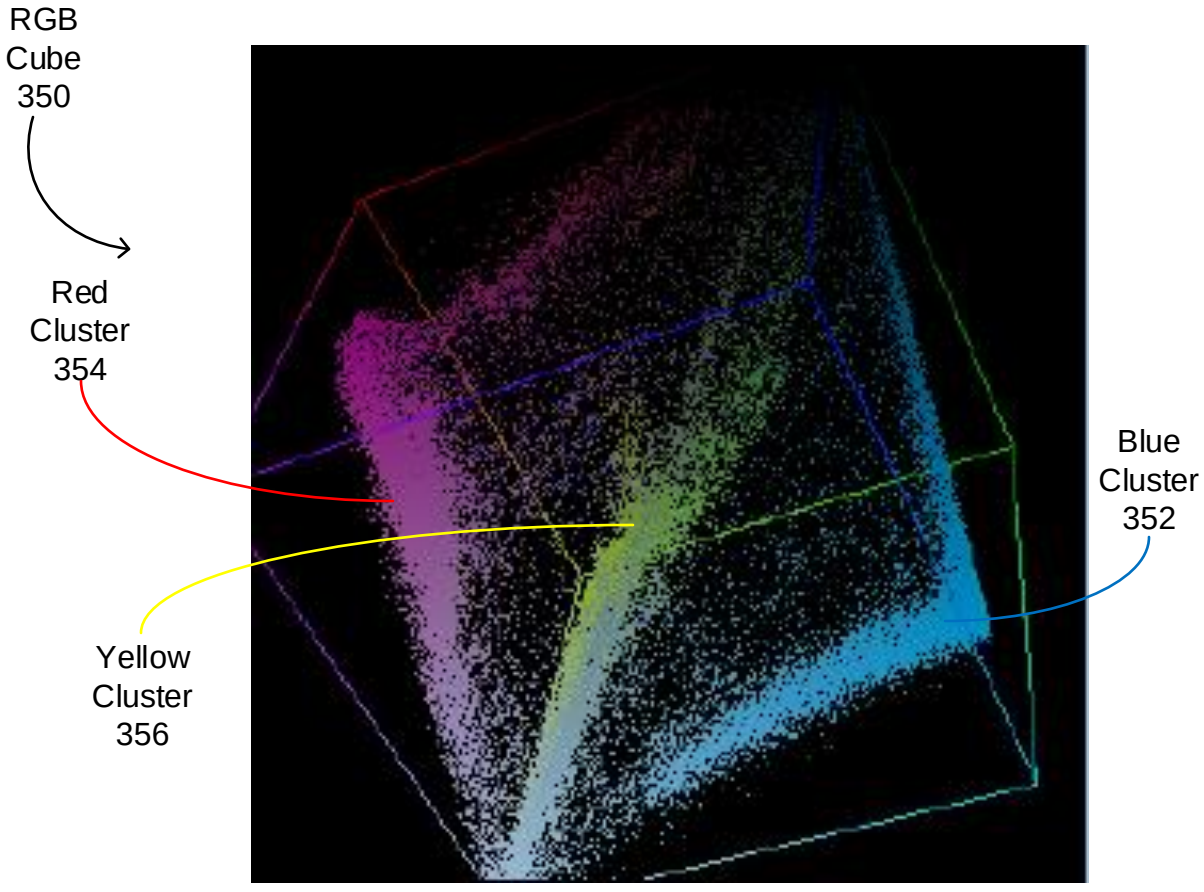
RGB
Cube
350
Red
Cluster
354
Blue
Cluster
352
Yellow
Cluster
356

FIG. 3B

Image 400

FIG. 4A

Compression Graph 450

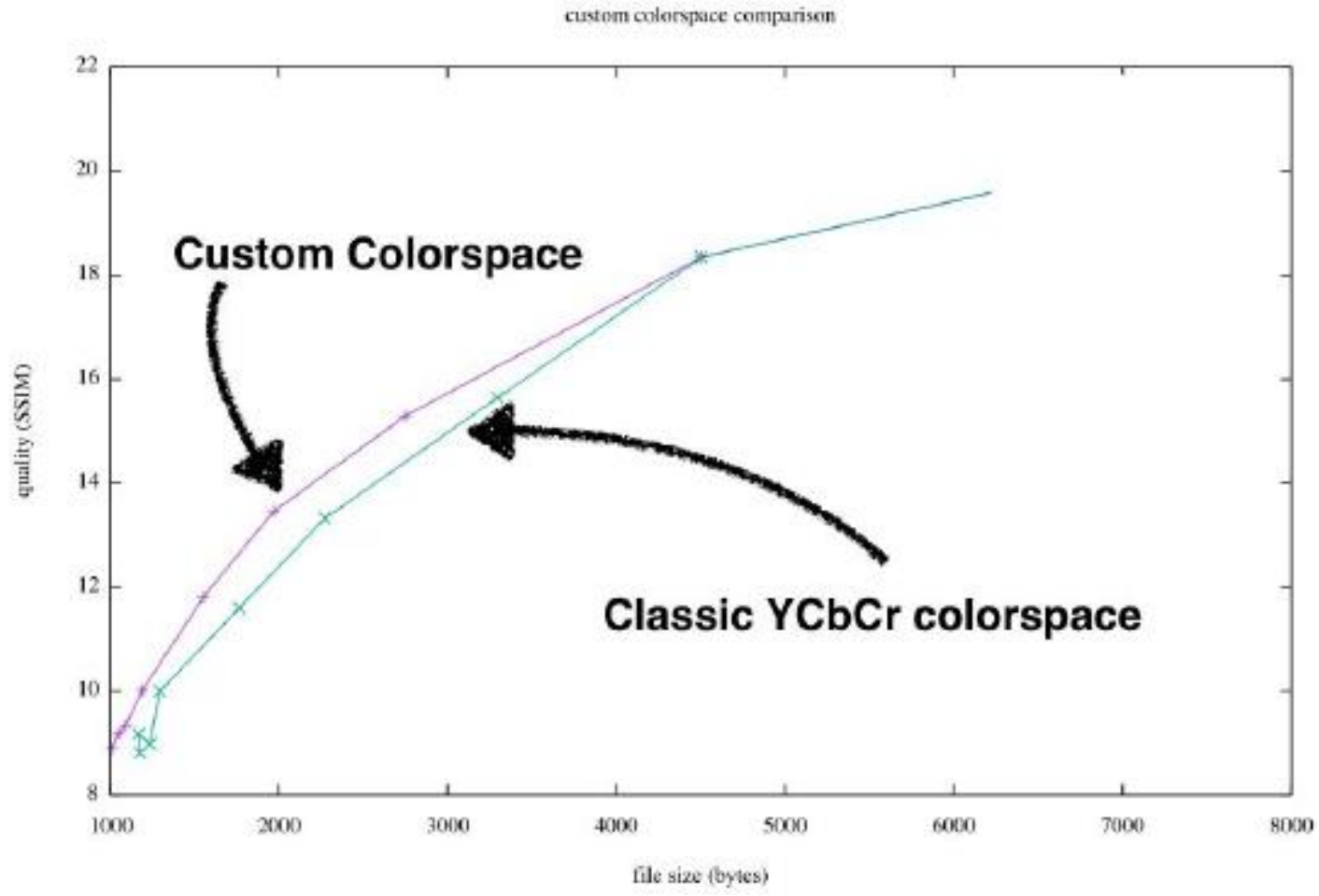

FIG. 4B