|  |  |
| --- | --- |
| **Joint Collaborative Team on Video Coding (JCT-VC)**  **of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11**  24th Meeting: Geneva, CH, 26 May – 1 June 2016 | Document: JCTVC-X0036 |

|  |  |  |  |
| --- | --- | --- | --- |
| *Title:* | **Branches: Modified Linearization of Luma Adjustment** | | |
| *Status:* | Input Document to JCT-VC | | |
| *Purpose:* | Proposal | | |
| *Author(s) or Contact(s):* | Jacob Ström, Jonatan Samuelsson, Kenneth Andersson, Per Hermansson Färögatan 6 16480 Stockholm | Tel: Email: | +46107136883 [jacob.strom@ericsson.com](mailto:jacob.strom@ericsson.com) |
| *Source:* | Ericsson | | |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Abstract

This contribution describes a modification of the linearization approximation of the luma adjustment procedure presented in W0107. The contribution highlights a case for which a linear approximation for luma adjustment results in numerical problems. It is described that these problems occur due to the fact that the linearization model does not take into consideration that some color components may clip against their maximum allowed value, resulting in the introduction of errors of significant magnitude. A solution is presented that is claimed to resolve these problems. The result is an approximate method that, does not introduce significant errors due to clipping. The method does not reach the same performance as the original iterative luma adjustment method, but it is an iteration-free technique, which could be good for hardware solutions or other implementations where worst-case complexity is of primary importance.

# Introduction

Francois et al. [1] first described a phenomenon that the anchor processing [3] can result in luminance artifacts. The problem was later quantified to be more than 200 times that what should be just noticeable [2]. A solution to this problem was presented by Ström et al. [4], and we will refer to in this solution as “luma adjustment”. For a more detailed description of the problem and the luma adjustment solution, please see the conference paper by Ström et al. [5]. .

One of the characteristics of the luma adjustment implementation as employed in MPEG and JCTVC is the fact that it is iterative. The iterative nature does not prohibit parallelization; it is possible to employ pipelining and still get one pixel out each clock. However, while the number of average iterations can be made smaller than two by using different bounds, this does not help hardware implementations, where the worst case scenario of ten iterations must be accommodated. Thus the pixel pipeline doing luma adjustment must be ten steps deep, even if only two steps are typically used.

To this end, Norkin proposes to use a linearization [6]. In his contribution, “algorithm 2” is a linearization of the iterative luma adjustment from [4] and [5].

However, as we will see, the linearization only works in the case when all three channels R’, G’ and B’ are free from clipping. If clipping occurs, this can generate substantial errors in luminance, even greater than if the traditional processing chain of [3] without luma adjustment is used.

It will be shown in this document that it is possible to avoid these errors completely. The idea is to first find out which color channels that will clip, and then use a linearization that is targeted for that particular case. As an example, if the blue channel clips, the blue variable will not be part of the equation that is being linearized. We will call this method “full branch linearization”.

In addition to the full branch linearization method, an approximate faster version is also provided. This has the ability to remove many of the clipping artifacts while still being as fast as the previously described linearizations.

We will start going through the linearization, followed by an example where the problem due to clipping appears. A solution to this problem is then presented that keeps the solution iteration-free. Finally the approximate faster version is described.

# Linearization

The linearization presented by Norkin [6] can be derived as follows: The decoder obtains Y’, Cb and Cr, and converts to R’, G’ B’ using

|  |  |  |
| --- | --- | --- |
|  |  | () |

where a13, a22, a23 and a32 are positive constants that depend on which color space is used. (For BT.2020 we have a13 = 1.4746, a22 = 0.1646, a23 = 0.5714, and a32 = 1.8814.) For ease of notation we introduce the helper variables

|  |  |  |
| --- | --- | --- |
|  |  | () |

which means that the conversion can instead be written

|  |  |  |
| --- | --- | --- |
|  | . | (3) |

We can now calculate which luminance the decoded pixel will get as

|  |  |  |
| --- | --- | --- |
|  |  | () |

where , and are constants such as the CIE1931 XYZ constants for Y or the (similar) constants for BT2020, for instance =0.2627, =0.6780, =0.0593.Using Equation (3), we can write this as

|  |  |  |
| --- | --- | --- |
|  |  | () |

In this formula, only is allowed to vary per pixel. As an example, depends on , which is shared between several pixels. We can thus regard , and as constants and the idea with luma adjustment is to find the that will generate the desired luminance . Note that since there are three non-linear functions in the right hand side of Equation (5), it is not possible to invert the right hand side. Therefore we instead introduce a linear approximation of for the red component.

|  |  |  |
| --- | --- | --- |
|  | , | () |

where is the transformed version of the linear component of the original pixel (the O stands for orignal). Doing the same for the green and blue components and inserting in Equation (5) gives

|  |  |  |
| --- | --- | --- |
|  |  | () |

Moving the first term in each expression to the left hand side, we get

|  |  |  |
| --- | --- | --- |
|  |  | () |

But since , the negative terms of the first line sum to the original luminance and the first line hence becomes zero. Collecting terms for Y’ now gives

|  |  |  |
| --- | --- | --- |
|  |  | () |

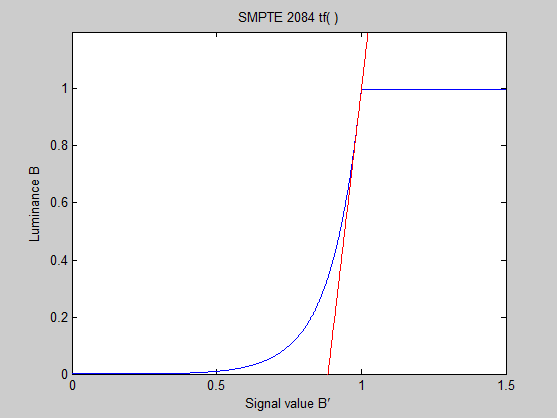
and we can thus calculate Y’ as

|  |  |  |
| --- | --- | --- |
|  |  | () |

This is mathematically equivalent to Equation (15) in W0107, which is the “Algorithm 2” method [6].

# Significant errors when clipping

As is well-known, a linearization works by replacing a function with a line tangential to the curve in the linearization point. As an example, the blue curve in Figure 1 plots the transfer function SMTPE 2084 for the blue component . The red line shows a linearization in the point 0.99. Note how this is a good approximation of the curve to the left of , at least close to the linearization point. However, at the transfer function saturates, which means that for values above 1.0, the red line will be a poor approximation of the blue curve, even for values close to 1.0.



*Figure 1. The blue curve equals the transfer function . The red line shows a linear approximation taken in the point . Note how this gives a bad approximation for values of even when is close to due to the saturation of the curve.*

As an example, consider two neighboring pixels and . After application of the inverse of the transfer function , we get

|  |  |  |
| --- | --- | --- |
|  |  | () |

and after conversion to we get

|  |  |  |
| --- | --- | --- |
|  | . | () |

The Cb and Cr values for the downsampled and upsampled image will be something similar to average of these two colors,

|  |  |  |
| --- | --- | --- |
|  |  | () |

When inserting these values in Equation (2) we get , , . We can now calculate the approximated according to the formula in Equation (10). According to Equation (11) , and . The derivative of the transfer function in these points are and . Inserting these values into Equation (10) yields a value of of . Equation (5) can now give us the resulting luminance for this pixel as .

The correct luminance can be obtained by inserting the linear color RGB1 into

|  |  |  |
| --- | --- | --- |
|  | , | () |

which gives , about *twice as bright*. The anchor processing would just take from the top line of equation (12), , which would give a luminance of . So in this case the anchor processing would be *better* than the linear approximation of luma adjustment. Doing full iterative luma adjustment according to [4,5] would instead result in , giving a luminance of , very close to the correct value.

The results of the different techniques for calculating luma and the resulting luminance are summarized in Table 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Calculation of luma for pixel RGB=(1000, 1000, 10000) with Cb= and Cr = .** | | | | |
| **method** | **Y’** | **Y** | **Y error** | **relative error** |
| **Original 4:4:4** | - | 1533,40 cd/m2 | - | - |
| **Anchor [3]** | 0.7665 | 1359.59 cd/m2 | -173.82 cd/m2 | -11.33% |
| **Luma Adjustment [4,5]** | 0.7889 | 1533.52 cd/m2 | 0.1147 cd/m2 | 0.007% |
| **W0107 Alg 2 [6]** | 0.6399 | 828.89 cd/m2 | -704.52 cd/m2 | -45.94% |
| **W0107 Alg 1 [6]** | 0.4609 | 632.53 cd/m2 | -900.87 cd/m2 | -58.75% |
| **Proposed method** | 0.7889 | 1533.52 cd/m2 | 0.1147 cd/m2 | 0.007% |

*Table 1: Error in luminance Y when calculating Y’ using different techniques.*

As can be seen in the table the linear approximations give relative errors of 45% and 58%, whereas the anchor method gives an error of 11%. Note also that the error for the regular luma adjustment method is very small. The last line contains the output of the proposed method which in this case is the same as for the regular luma adjustment method, and hence produces a good result.

The example above was deliberately extreme in order to highlight the issue, with values near 10000. However, even for the range of values used in the MPEG test sequences, [0, 4000]cd/m2, this issue appears. As an example running algorithm 1 on, and will generate a luminance Y of 1364 cd/m2, whereas the correct luminance is 1593 cd/m2, a relative error of14%. The proposed method will in this case give a much closer 1643 cd/m2, a relative error of 3%.

The hybrid log gamma (HLG) transfer function uses a scene referred scale, and therefore the full range up to 1.0 is typically used. In such a system the type of clipping errors such as the one described above would be even more common.

# Solution

The problem can be summarized as follows: We want to find the solution to the equation

|  |  |  |
| --- | --- | --- |
|  |  | () |

where clips the value to the interval . The problem is that we don’t know whether the optimal , let’s call it , will cause one or more of the channels to clip. For instance, if will cause the blue component to clip against 1 but not clip the red and green components (as in the example above), we should not linearize Equation (5) but instead linearize and solve

|  |  |  |
| --- | --- | --- |
|  |  | () |

The problem though is that to know which channels will clip, we need , but to get that we need to apply a formula such as Equation (16), but to know which formula to use we need to know which channels will clip.

To break this catch 22, we can take advantage of the fact that we don’t need to know the exact value of to select the right equation to linearize. As an example, if we disregard the red and green component for a moment, to select between linearizing Equation (5) and Equation (16), we only need to know whether the optimal is larger than the needed for the blue component to clip against 1. This value is known, since it will clip exactly when . We can therefore calculate the linear luminance in this point and see if that is larger or smaller than the desired . If it is larger, then we know that the optimal will not clip the blue component and we can linearize Equation (5). Otherwise we know that the optimal will indeed clip the blue component so it is better to use a linearization of Equation (16). In the example above, we know that the blue component will clip at . Inserting this into Equation (5) or (15) gives that the luminance at this point is 616.36 cd/m2. But since this is less than the desired luminance of 1533,40 cd/m2 we know that the optimal must be larger than 0.4129, and that we therefore should linearize Equation (16) rather than Equation (5). Doing that results in , which gives a luminance of 1533.52 cd/m2. In this particular case the rounding of Y’ to the nearest 10-bit integer meant that the linearization got the same result as the iterative method [4,5], but that will not always be the case since this is an approximation.

We can regard the two equations (5) and (16) as branches of a tree (hence the title), and we need to find which branch to use. Since there are three color channels and each can clip against 0 or 1 or not clip, there are 27 branches to choose from. We will now present an easy and efficient way to select the correct branch. First however we will present a linearization equation that works for all branches.

## Derivation of general linearization

In this section we will derive a linearized version of Equation (15) that can be used both when the variables clip and when they do not clip. First, define the variables and as

|  |  |  |
| --- | --- | --- |
|  |  | () |
|  |  | () |

The variables , , and are defined analogously. This means that we can write Equation (15) as

|  |  |  |
| --- | --- | --- |
|  |  | () |

where is shorthand for . Now we introduce the linearization approximation from Equation (6) and ditto for green and blue, which gives

|  |  |  |
| --- | --- | --- |
|  |  | () |

Gathering terms on the right hand gives

|  |  |  |
| --- | --- | --- |
|  |  | () |

Which gives the following expression for

|  |  |  |
| --- | --- | --- |
|  |  | () |

## Calculation of clipping variables

In order to use the generalized formula in Equation (22) we need the variables and for the color channels. They can be calculated using the following steps:

1. First calculate , , and using Equation (2).
2. Calculate the value where the red channel starts to clip. It is t if it clips against , and if it clips against 1. Note that it cannot clip both 0 and 1 for legal values of . Store this value as , and do the same for green and blue.
3. Sort the values , call the sorted values .
4. Calculate the luminances associated with using Equation (15).
5. Figure out which interval contains the desired luminance . For instance, if then we know that belongs to the interval .
6. Figure out the clipping variables and for this interval. This can for instance be done by calculating the mid-point of the interval and testing against and for the red channel and likewise for green and blue.

For code that calculates the clipping variables, see Appendix A.

## Application to Alg 1 of W0107

The method presented above works for linearization of luma adjustment when we try to match the luminance of the decoded pixel with the luminance of the original pixel as is done in [4] and [5]. This means that it is applicable to “algorithm 2” in by W0107 Norkin [6] in the manner described above.

Norkin also proposes a method where instead the sum of the squared errors of the linear components is minimized,

|  |  |  |
| --- | --- | --- |
|  |  | (23) |

where is the original color and is the corrected color for a pixel. This is called “algorithm 1” in W0107. Unfortunately, it is not straightforward to apply the method presented above to algorithm 1. The reason is that the error function in Equation (23) is not guaranteed to have only one local minimum, but can have several.

As an example, assume we have the following two linear RGB-values:

RGB1 = (113, 3900, 33);  
RGB2 = (3.11, 0.17, 3950);

Applying the inverse EOTF gives

RGBp1 = (0.5204, 0.8998, 0.4015)  
RGBp2 = (0.2157, 0.0776, 0.9012)

Converting to YCbCr gives

YCbCr1 = (0.7706, -0.1962, -0.1697)  
YCbCr2 = (0.1627, 0.3925, 0.0359)

and averaging these two values gives

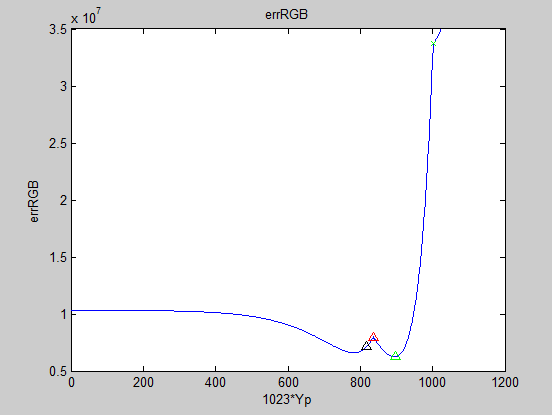
YCbCr = (0.4667 0.0982 -0.0669)

Now, if we want to select a Y’ for the first pixel (RGB1) we set Ro’ = 0.5204, Go’ = 0.8998 and Bo’ = 0.4015. We have Cb = 0.0982 and Cr = -0.0669.

Now we can plot the RGB-error:

errRGB = wR\*(tf(Y’ + a13\*Cr)-Ro)^2 + wG\*(tf(Y’-a22\*Cb-a23\*Cr)-Go)^2 + wB\*(tf(Y’+a32\*Cr)-Bo)^2

This is plotted in Figure 2.



*Figure 2: the errRGB plotted as a function of Yp. Yp is multiplied by 1023 in this figure. Note how the curve has two local minima, one around 900 and another around 780. There is thus no simple way to know which approximation to use.As an example, even if the function is known in the point marked with the red triangle, and it is also known that the a point to the left of this (marked with a black triangle) is at a lower level, it is still not possible to conclude that the minimum is left of the red point. In fact it is right of this point.*

As can be seen in this figure, it has two local minima, only one of which is the global minimum. This means that it is not possible to simply know which side of a known value the minimum lies. As an example, assume that we have calculated the value of the error function in the point marked with a red triangle in the figure, and that we also knew that a value closely to the left of this (marked with a black triangle) would generate a lower value. If the error function was guaranteed to only have one global minimum, we would then be able to conclude that the optimal value must be to the left of the red triangle. But in fact it is to the right of the red triangle, at the green triangle. Therefore it is not straightforward to select which branch to pick.

In contrast, if we try to match luminance as algorithm 2 in W0107 does, it is straightforward to select the correct interval (branch) if the luminance is known at all clipping points. It will simply be the branch that contains the desired luminance .

## Parallelizability and worst case complexity

In order to identify the correct clipping variables and , Section 4.2.2 describes that the luminance is calculated in three clipping points . Since each such calculation involves a calculation of the transfer function up to two times (one of the variables clip and does therefore not need transformation) this can be quite expensive, even compared to the iterative technique: Combining the mathematical bounds presented by Ström et al. [6] with the bounds presented by Tourapis et al. JCTVC-W0052 [7], it is possible to get the average number of iterations down to less than two iterations per pixel. This is comparable to the method proposed here, since it will also need the equivalent of two luminance conversions. However, an important aspect when it comes to hardware implementation is that the worst case performance for the proposed method is equivalent of two luminance conversions. For the iterative bisection search method the worst case is ten luminance conversions, since there is no way to guarantee that the bounds will give anything better than ten iterations per pixel.

## Faster versions for software implementations

For software implementations however, it may very well be the case that the bisection approach is faster than the method presented above. To speed up a software implementation, it is possible to employ an approximate method. First, Equation (6) is used to obtain a value. Next, this value is tested for clipping. If it turns out that it clips any of the channels, the proposed method is employed, calculating the clipping parameters using the code in Appendix A and then employing Equation (22). However, if it turns out that the pixel doesn’t clip any of the channels, the value calculated using Equation (6) is used and the method then proceeds to the next pixel. Since most pixels will not generate clipping, this method will be almost as fast as the method in [6], but will avoid the significant errors described in the example above. We call this method “Condition 01”.

An even faster approach will be to employ the proposed method only for values clipping against 1 and not for values clipping against 0, since the errors are much bigger for the first case. We call this method “Condition 1”.

A faster method still would be the following: In case a pixel is clipping against 1, instead of executing the code in Appendix A to calculate the clipping parameters, the obtained in Equation (6) is used to decide the clipping parameters that are then used in Equation (22). In detail, the obtained in Equation (6) is used instead of in step 6 in the code in Appendix A, and the resulting and variables are then used in Equation (22). We call this method “Conditional 1 approx”.

The table below summarizes the above-mentioned methods:

|  |  |  |  |
| --- | --- | --- | --- |
| Method | First step | Second step | Comment |
| “Full Branch Linearization” | Calculate and using code in Appendix A | Use Eqn (22) |  |
| “Conditional 01” | Use Eqn (6) | If resulting clips against 0 or 1, do “full” | Worse worst-case performance than “full” but better average case performance |
| “Conditional 1” | Use Eqn (6) | If resulting clips against 1, do “full” | Same worst-case performance as “Conditional 01” but better average performance |
| “Conditional 1 approx” | Use Eqn(6) | If resulting clips against 1, use and then step 6 in Appendix A to calculate and , then use Eqn (22) | Fastest average case performance and worst case performance, but may give approximate results. |

# Conclusion

This document has described an enhancement of the linearization techniques described in W0170. A problem has been identified that can cause artifacts under certain conditions. The problem is due to the fact that the luma value needed to reach the desired luminance may clip one or more of the color channels. By calculating the luminance at the luma values where the each channel clips it is possible to figure out which formula to linearize (which branch to take). This avoids the significant errors that are otherwise possible during clipping. The methods have a worst case complexity that is considerably lower than iterative luma adjustment. For software where average performance is more of interest, several faster versions have been suggested.

# Appendix A

Code for calculating the clipping variables and .

/ Figure out which values of Y' are on the limit of clipping.  
// Step 1: Calculate Rterm, Gterm and Bterm from Cr and Cb and  
// set up some variables. The desired luminance is stored  
// in yLinear.  
  
int clipListIndex = 0;  
double YpThatClip[5] = {0.0, 0.0, 1.0, 1.0, 1.0};   
// The first and the last are 0.0 and 1.0 respectively.   
// Only the 3 middle ones are calculated.   
  
double YLinAtClipBoundary[5] = {-0.1, 0.0, 1.0, 1.0, 1.0};   
// The first and the last are -0.1 and 1.0 respectively, to stop the search   
// in the correct position if we have Ytarget = 0.0 or 1.0.  
  
double Rterm = a13\*Cr;  
double Gterm = -a22\*Cb – a23\*Cr;  
double Bterm = a22\*Cb;

// Unless Rterm = 0 we cannot have a situation where Y' in range [0, 1] clips against  
// 0 and another Y' in range [0, 1] clips against 1.  
// For instance, if Rterm > 0 then R' = Y' + Rterm will clip against 0 only for Y' < 0,   
// which never happens. We can thus discard that one.

if( Rterm < 0 )  
 YpThatClip[1 + clipListIndex++] = -Rterm;  
else  
 YpThatClip[1 + clipListIndex++] = 1-Rterm;  
if( Gterm < 0 )  
 YpThatClip[1 + clipListIndex++] = -Gterm;  
else  
 YpThatClip[1 + clipListIndex++] = 1-Gterm;  
if( Bterm < 0 )  
 YpThatClip[1 + clipListIndex++] = -Bterm;  
else  
 YpThatClip[1 + clipListIndex++] = 1-Bterm;  
  
// Step 2: Sort the three middle values so that YpThatClip are in increasing order.  
// Use bubble sort.  
bubble3(&YpThatClip[1]);  
  
// Step 3: Evaluate Ylinear for every Y' in the list  
for(int q = 0; q<3; q++)  
 YLinAtClipBoundary[1+q] = convertToYLinear(YpThatClip[1+q], Rterm, Gterm, Bterm);  
  
// Step 4: Find out which interval of Y' we belong to, i.e., which branch we are in.   
int qq = 1;  
while(YLinAtClipBoundary[qq] < yLinear)  
 qq++;

// Step 5: Find a representative for that branch to find out which variables we clip.   
double YpMid = (YpThatClip[qq-1] + YpThatClip[qq])/2.0;  
  
// Step 6: Set Clipping variables for this branch  
int l[3] = {0, 0, 0};  
int n[3] = {0, 0, 0};

if( (YpMid + Rterm > 0)&&(YpMid + Rterm < 1) )  
 n[0] = 1;  
else if( YpMid + Rterm > 1)  
 l[0] = 1;  
if( (YpMid + Gterm > 0)&&(YpMid + Gterm < 1) )  
 n[1] = 1;  
else if( YpMid + Gterm > 1)  
 l[1] = 1;  
if( (YpMid + Bterm > 0)&&(YpMid + Bterm < 1) )  
 n[2] = 1;  
else if( YpMid + Bterm > 1)  
 l[2] = 1;

# References

1. E. François, P. Lopez, Y. Olivier, (Technicolor) “About using a BT.2020 container for BT.709 content”, MPEG2013/m35255, Sapporo 2014.
2. J. Strom, (Ericsson) “Investigation of HDR Color Subsampling”, m35841, 110th MPEG meeting, Geneva.
3. Ajay Luthra, Edouard François, Walt Husak, “Call for Evidence (CfE) for HDR and WCG Video Coding”, n15083, 110th MPEG meeting, Geneva
4. J. Ström, J. Samuelsson, M. Pettersson, K. Andersson, P. Wennersten, R. Sjöberg, “Ericsson's response to CfE for HDR and WCG”, ISO/IEC (MPEG) document m36184, Warsaw, Poland, June 2015.
5. Jacob Ström, Jonatan Samuelsson and Kristofer Dovstam, "Luma Adjustment for High Dynamic Range Video", Proceedings of the IEEE Data Compression Conference (DCC), Snowbird, March 2016.
6. A. Norkin “Closed form HDR 4:2:0 chroma subsampling (HDR CE1 and AHG5 related)”, JCTVC-W0107, San Diego, Feb 2016.
7. A. Tourapis, Y. Su, D. Singer “Enhanced Luma Adjustment Methods”, JCTVC-W0052, San Diego, Feb 2016.

# Patent rights declaration(s)

**Ericsson may have current or pending patent rights relating to the technology described in this contribution and, conditioned on reciprocity, is prepared to grant licenses under reasonable and non-discriminatory terms as necessary for implementation of the resulting ITU-T Recommendation | ISO/IEC International Standard (per box 2 of the ITU-T/ITU-R/ISO/IEC patent statement and licensing declaration form).**