

## EVALUATION OF ROBUST COLOR RECOGNITION ALGORITHMS

Samuel Kosolapov

Braude Academic College of Engineering, Karmiel, Israel

### Abstract

*A number of color recognition algorithms are known - from algorithms using classic HSL/HSV space to modern AI algorithms. In some practically important situations, the unavoidable electronic noise of the digital camera makes color recognition non-reliable, for example, in dark regions and in over-bleached regions. A new robust approach is proposed to mark problematic regions and, depending on parameters selected by the user, to exclude those problematic regions from the later analysis. In order to enable a human observer and to third-party applications to recognize problematic regions (in which color recognition is not reliable), it was decided to present results as a true color map, in which problematic regions have a strong color component (for example: red, green, or blue). The paper provides some results of applying the proposed algorithm for synthetic and real-life images.*

**Keywords:** image processing, color recognition, Hue, robust processing

### 1. INTRODUCTION

These days digital cameras of different sizes and resolutions are practically everywhere. This fact results in the developing of many applications using different image processing algorithms. Some of those applications use color recognition algorithms. Image processing algorithms can be roughly divided into algorithms designed for a Human Observer and for algorithms designed for Machine Vision applications. The goal of algorithms designed for a Human Observer is to present a processed image in the most impressive way – considering the preferences of a typical human observer. The goal of algorithms designed for Machine Vision applications is to analyze the image in order to execute some machine operation – for example, cutting materials about some color edge. For example, in the photo of “lemons and mandarines” (see Fig.1), a human observer can easily point to “yellow lemons”, to “red mandarines”, and easily distinguish them from green leaves.



**Fig. 1.** Photo of “lemons and mandarines”

This photo was taken by a forward camera of a smartphone “Samsung Galaxy Note 20 Ultra 5G”. In order to provide an adequate presentation of the images, by taking into account publication requirements, high-resolution JPG photo was rescaled to the 640x359 true color BMP image.

Considering that modern digital cameras, in most cases, create images of high color fidelity, one can conclude that in this case, no additional image processing is needed. Even if it is known, that specific camera, used to get this photo, creates some color distortion (in order to “impress” the human observer by presenting to him “more vivid” image), those distortions will not prevent the human observer from easily distinguishing yellow lemons from a green leaves and other colored objects (red mandarines and white wall in the background) that can be seen on this photo.

However, if, for example, the goal is to point some robotic arm to collect yellow lemons only, the software of this robotic arm must contain some image processing algorithms. In this specific case, those algorithms may recognize (and mark) “yellow lemons” as “interesting objects” having specific “yellow” color, whereas leaves and other objects having other colors may be marked as “non-interesting regions”. In that case, the resulting image must be a binary image having two colors only, for example – “black pixels” marking pixels “recognized as yellow” and “white pixels” marking pixels “non recognized as yellow”.

A number of classical color recognition algorithms are known. Some algorithms are based on using HSL/HSV color spaces (Russ & Neal 2017, Jain et al. 1995, Foley et al. 1997). Other algorithms use additional techniques, like scalar product in the RGB space, normalized correlation in RGB space and color jumps (Russ & Neal 2017, Kosolapov 2007, Kosolapov et al. 2013).

The simplest color recognition algorithm is based on converting pixels in the RGB space to HSL/HSV space, and, by using the hue value as a color descriptor. Mathematically, hue has a value in the range  $\{0..360\}$  degrees, but in order to present the “Hue map” to a human observer, the hue value is typically scaled to range  $\{0..255\}$ . Then, hue values can be presented to the human observed as a “Hue Map”. This “Hue map” can be used as by human observers, as by machine vision algorithms. The human observer can recognize objects of the same type as having nearly the same gray level. Machine vision applications can simply recognize an “interesting objects” as having hue values in a specific region. For example, the hue value of green color is 120 degrees. Then, a machine vision application may treat all pixels having hue from, say, 115 to 125 degrees, as pixels of the “interesting object”.

The problem with this classical HSV/HSL approach is its low reliability of color recognition in some practically important situations. For example, when the pixel is gray, the hue value cannot be calculated. In that case, most practical algorithms set hue to zero. But this is a problematic practice because RED color has a hue value of 0 degrees. In that case, the “Hue map” can be misleading for the human observer.

Another problem is the noise in the camera. Mathematically, it is clear that pixel  $\{255,0,0\}$  is RED. Then, mathematically, pixel  $\{1,0,0\}$  is also a RED. Unfortunately, with real cameras, this is not the case. Electronics cameras have an inherent electronic random noise. In the range  $\{0..255\}$  this noise can reach levels of 25 units. In most cases, this noise is not seen because the human visual system can “smooth” it. These days most images are in JPG format. Those JPG images effectively filtered out the noise of the cameras, which is an excellent feature for the human observer. But for the machine vision application, this “internal smoothing” can be considered as “non-requested serious distortion” of the original image. It may lead to severe problems in real applications. This is why most machine vision application uses “raw data”, so that camera noise is not filtered out before processing.

In the dark regions and regions with low color saturation (gray images), the HSV/HSL approach became non-reliable if camera noise was present. Additionally, in some situations, colors are distorted by the effect of “over bleaching”.

To cope with this problem, classical HSL/HSV algorithms were modified in an attempt to make them more robust. This paper is a continuation of the previous work (Kosolapov 2021). It will present additional details of the implementation and some new results.

## 2. MODIFIED ALGORITHMS DESCRIPTION

### 2.1. *Input and output arrays and structures*

In order to provide analysis of the selected algorithms, a special test application was created by using Microsoft Visual Studio 2022. Plain classic C language was selected because this language enables (at the end) to create fast and memory-effective image processing algorithms that can be easily ported to a wide range of microprocessors and microcontrollers – from simplest to dedicated DSP systems.

A special synthetic image, “Green Digital Ramp,” was created for the algorithm test. The input of the HSL/HSV algorithms, in this case, was a true-color image declared as a static three-dimensional array of bytes (“unsigned char”) having fixed CGA sizes: `NUMBER_OF_ROWS = 240`, `NUMBER_OF_COLUMNS = 320` and `NUMBER_OF_COLORS = 3` (known as “R”, “G” “B”). The resulting “Hue map” was declared as a three-dimensional array of bytes having fixed sizes: `NUMBER_OF_ROWS = 240`, `NUMBER_OF_COLUMNS = 320`. In the case of “real life” images, sizes were set in accordance with the actual image sizes. Dynamic allocation was not used because it is not always available in small systems. Additionally, to make the code of the algorithms human readable, pointers were not used in this article. It is clear that real-life applications will use pointers to make execution fast.

Additionally, in order to lessen the number of parameters in the functions, some structures were declared. Fig.2 presents some important structures that will be used later. Original pixel values are packed in the `sRGB` structure. Its fields are declared as “unsigned char” values. Unsigned char type is problematic for the mathematical calculations (for example, because of integer over float) hence structures `sHSLVf` and `sHueValidityParameter` use mostly double values. (In Visual Studio, double is a recommended format, but in the case of porting to the microprocessors, float values will probably be more adequate.

```
40
41 struct sHSLVf
42 {
43     double H;    // H is the same for HSL and HSV
44     double SL;   // Saturation for HSL
45     double SV;   // Saturation for HSV
46     double L;    // Lightness
47     double V;    // Value
48     int Hflag;   // Hue Validity flag
49 };
50
51 struct sRGB
52 {
53     unsigned char r;
54     unsigned char g;
55     unsigned char b;
56 };
57
58 struct sHueValidityParameter
59 {
60     unsigned char rgbMin;
61     unsigned char rgbMax;
62
63     double Vmin;
64     double Vmax;
65
66     double Lmin;
67     double Lmax;
68
69     double lSaturationMin;
70     double vSaturationMin;
71 };
72
```

**Fig. 2.** Important structures

It must be noted that in this paper, values of “Saturation” will not be used, and values of “Lightness” and “Value” will be used only as intermediate variables needed to calculate the value of the “Hue”.

In the frame of this paper, modifications of classic algorithms deal with too low values of “rgb”, too low “Lightness/Value”, “over bleaching,” and low values of saturation. The idea of modifying a classic code is that the color of “dark” pixels cannot be calculated in a reliable way; hence, those pixels must be in some way marked as “dark” and excluded from the color analysis. Of course, in this approach, some pixels are excluded from color recognition, but this effectively prevents “false recognitions” in the dark regions. The additional idea is to exclude pixels, the color of which is close to gray. To do this, the following definitions were provided (See Fig. 3).

```
27
28  #define HUE_IS_VALID      (0)
29  #define IS_DARK           (-1)
30  #define IS_OVERBLEACHED   (-2)
31  #define IS_GRAY           (-3)
32
33  // If Hue is Valid          : pixel is GRAY according to its HUE value
34  //                          : scaled to 255
35  // If Too Dark              : Pixel is RED
36  // If Too Light (overbleached) : Pixel is GREEN
37  // If Is Gray               : pixel is BLUE
38  unsigned char ResultingHueImage[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS][NUMBER_OF_COLORS];
39
```

**Fig. 3.** Definitions for the Resulting Image

An important change of the traditional gray “Hue Map” is that now the H-map is a color image. Current algorithm supports four situations defined at lines 28-31 in the Fig. 3. When the value of Hflag in the line 48 on the Fig 2(after proper calculations described later) is set to HUE\_IS\_VALID (see definition at the line 28 on the Fig. 3, then all r, g, b values of the ResultingImage (see line 38 on the Fig. 3 are set to calculated H value in the range {0..255}. In that case the pixel is seen as gray, and the gray value is connected with the color of this pixel according to the hue calculation. In case pixel found too dark the pixel is seen as RED. In case of overbleaching, the pixel is set to GREEN, and in case of low saturation, the pixel is set to BLUE. Then, by observing this “Color Hue map”, the human observer immediately recognised “problematic” pixels as having strong color. Machine vision application can easily do the same, but, additionally, application may change illumination and/or camera parameters, so that less pixels will be marked as problematic.

## 2.2. Important Functions

Most of the modifications were provided in the function ConvertRGBtoHSLVf. For the tests, this function has a flag “useLimits”. If this parameter is set to TRUE, then parameters specified in the sHueValidityParameter structure are used. In case “useLimits” is set to FALSE, then this function provides calculations in a classical way. In order to make code simpler for the tests, HSL and HSV algorithms are implemented in the same function. This is possible because the results of this function are values of H, values of L, V, and S are used to evaluate a hue value's validity.

All modifications are written inside multiple IF statements (if (useLimits == TRUE)). Then the parameters are compared with original pixel values and calculated values. For example, if (pixel.r < param.rgbMin), then Hflag is marked as “IS\_DARK”. But if (pixel.r > param.rgbMax), then Hflag is marked as “IS\_OVERBLEACHED”. There are six “IF” of that tests – for r, g, and b values of the specific pixel. After individual values of the current pixel are checked, L and V are calculated and checked similarly: if they are within the limits specified for this specific image. Checks for rgb, L, and V are presented in Fig. 4 (at the left – checks for “rgb”, at the right – checks for L and V). Calculations of L and V were not changed.

```

85 struct sHSLVf ConvertRGBtoHSLVf(
86     struct sRGB pixel,
87     sHueValidityParameter param,
88     int useLimits)
89 {
90     sHSLVf hslvf;
91     hslvf.H = 0;
92     hslvf.SL = 0;
93     hslvf.SV = 0;
94     hslvf.L = 0; // Can be always calculated
95     hslvf.V = 0; // Can be always calculated
96     hslvf.Hflag = HUE_IS_VALID;
97
98     if (useLimits == TRUE)
99     {
100         if (pixel.r < param.rgbMin)
101         {
102             hslvf.Hflag = IS_DARK;
103             return hslvf;
104         }
105
106         if (pixel.g < param.rgbMin)
107         {
108             hslvf.Hflag = IS_DARK;
109             return hslvf;
110         }
111
112         if (pixel.b < param.rgbMin)
113         {
114             hslvf.Hflag = IS_DARK;
115             return hslvf;
116         }
117
118         if (pixel.r > param.rgbMax)
119         {
120             hslvf.Hflag = IS_OVERBLEACHED;
121             return hslvf;
122         }
123
124         if (pixel.g > param.rgbMax)
125         {
126             hslvf.Hflag = IS_OVERBLEACHED;
127             return hslvf;
128         }
129
130         if (pixel.b > param.rgbMax)
131         {
132             hslvf.Hflag = IS_OVERBLEACHED;
133             return hslvf;
134         }
135     }
136
137     // normalized values in the range 0..1
138     double r = (pixel.r / 255.0);
139     double g = (pixel.g / 255.0);
140     double b = (pixel.b / 255.0);
141
142     double min = Min(a:Min(a:r, b:g), b);
143     double max = Max(Max(a:r, b:g), b);
144     double delta = max - min;
145
146     // Calculations of V (Value)
147     hslvf.V = max;
148
149     if (useLimits == TRUE)
150     {
151         if (hslvf.V < param.Vmin)
152         {
153             hslvf.Hflag = IS_DARK;
154             return hslvf;
155         }
156
157         if (hslvf.V > param.Vmax)
158         {
159             hslvf.Hflag = IS_OVERBLEACHED;
160             return hslvf;
161         }
162     }
163
164     // Calculations of L (Lightness)
165     hslvf.L = (max + min) / 2;
166
167     if (useLimits == TRUE)
168     {
169         if (hslvf.L < param.Lmin)
170         {
171             hslvf.Hflag = IS_DARK;
172             return hslvf;
173         }
174
175         if (hslvf.L > param.Lmax)
176         {
177             hslvf.Hflag = IS_OVERBLEACHED;
178             return hslvf;
179         }
180     }

```

**Fig. 3.** Calculation of “Hue” by RGB and limit parameters

Left: providing “rgb” checks. Right: providing V and L checks

Only two flags can be set during tests presented in Fig. 3: “IS\_DARK” and “IS\_OVERBLEACHED”. After checks of “rgb”, L, and V, tests in that style were added to the saturation calculations. Again, this function calculates saturation as for the HSL algorithm as for the HSV (SL and SV) without changes. Important to understand that values of saturation and values of L and V are used only to calculate the hue value, so that calculations provided by two algorithms are used here to set flags only.

Then, finally, hue is calculated. Fortunately, hue calculations are the same as for the HSL as for the HSV algorithm. Hue calculations are based on the value of “delta” – which is calculated as a difference “max(r,g,b) – min (r,g,b)”. It is clear that if this difference is zero, then the pixel is gray. It was decided not to add minimal value of the “delta” to the set of parameters “sHueValidityParameter” because nearly the same calculations and checks were provided in lines 181 - 191 and 201 – 216. It is clear that when saturation is close to zero, the flag “IS\_GRAY” must be set. Code for the calculations and checks of saturation and hue is provided in Fig. 4. On the left, one can see code relevant for the calculations and validity checks for the saturation (by using two algorithms – HSL and HSV). On the right – one can see calculations and validity checks for the hue. In this code final value of the hue is set, but this value can be considered as valid only if the flag Hflag was set to “HUE\_IS\_VALID”..

The next important function is “ConvertRGBImageToHueImage”. It consists of two loops (for row and for column) so that for every pixel of the trueColorImage, hue and Hflag are calculated. Then, according to their values, pixel of the resulting image is set to gray value in accordance to the hue value (in case Hflag is set to HUE\_IS\_VALID,) or to one of the three bright warning colors (RED, GREEN,, BLUE), in case value of the hue cannot be considered as reliable. The code can be seen in the Fig.5.

```

180
181 // Calculate Saturation for HSL
182 if ( (hslvf.L == 0) || (hslvf.L == 1))
183 {
184     hslvf.SL = 0;
185 }
186 else
187 {
188     hslvf.SL = (hslvf.V - hslvf.L)
189         / Min(a:hslvf.L, b:1- hslvf.L);
190 }
191
192 if (useLimits == TRUE)
193 {
194     if (hslvf.SL < param.lSaturationMin)
195     {
196         hslvf.Hflag = IS_GRAY;
197         return hslvf;
198     }
199 }
200
201 // Calculate Saturation for HSV
202 if (hslvf.V == 0)
203 {
204     hslvf.SV = 0;
205 }
206 else
207 {
208     hslvf.SV = delta / hslvf.V;
209 }
210
211 if (useLimits == TRUE)
212 {
213     if (hslvf.SV < param.vSaturationMin)
214     {
215         hslvf.Hflag = IS_GRAY;
216         return hslvf;
217     }
218 }
219
220 // Calculate Hue in the range 1..360 degrees
221 if (delta == 0) // Pixel is Gray
222 {
223     if (useLimits == TRUE)
224     {
225         hslvf.Hflag = IS_GRAY;
226         return hslvf;
227     }
228     else
229     {
230         hslvf.H = 0;
231     }
232 }
233
234 if (hslvf.V == r) // RED is the highest color
235 {
236     hslvf.H = 60 * (0 + (g - b) / delta);
237     return hslvf;
238 }
239
240 if (hslvf.V == g) // GREEN is the highest color
241 {
242     hslvf.H = 60 * (2 + (b - r) / delta);
243     return hslvf;
244 }
245
246 if (hslvf.V == b) // BLUE is the highest color
247 {
248     hslvf.H = 60 * (2 + (r - g) / delta);
249     return hslvf;
250 }
251
252 return hslvf;
253

```

Fig. 4. Calculation and validity checks of “Saturation” and “Hue”

Left: Calculations and validity checks for “Saturation”. Right: the same for “Hue”

Code in Fig. 5 uses three-dimensional indexing. Of course, this is slow, and pointers must have been used here. But, considering the goal of this paper– to describe modifications in the well-known classical algorithm, it was decided to make it more reader-friendly and not to hide implementation details by using tricks with pointers and pointer arithmetic. It must be reminded that the “mathematical” hue value is in the range  $\{0..360\}$  degrees. The resulting “Hue map”, for obvious reasons, is rescaled to the values in the range  $\{0..255\}$ , whereas value 0 is valid value – meaning pure RED color.

### 3. RESULTS

In order to test the modified algorithm, well known synthetic test image: "Green Digital Ramp" was created by software with and without pseudo-random noise added to each pixel. Upper-left image in Fig 6. presents the synthetic image "GreenDigitalRamp". It consists of a repeated sequence of 16 color strips. Top-right image in Fig 6 presents a profile of a typical line of this image – it is simply a graph of the

"light intensity" of the green component versus the column of the specific pixel. Considering the upper left image: the human observer may see a different number of "strips" depending on the quality of the monitor in use. By using a "technical tool" – profiler, it can be seen that the value of "GREEN" on the left is zero, and the maximal value of "GREEN" is 255. Those values are "legal" for a synthetic image – an image created by user-defined software. However, in the real-life cameras, values 0 and 255 in most cases are signaling that camera parameters are not optimal, and, in those situations, colors are distorted.

```
303 void ConvertRGBImageToHueImage(  
304     unsigned char trueColorImage[][NUMBER_OF_COLUMNS][NUMBER_OF_COLORS],  
305     unsigned char resultingHueImage[][NUMBER_OF_COLUMNS][NUMBER_OF_COLORS],  
306     sHueValidityParameter param,  
307     int useLimits)  
308 {  
309     sHSLVf hslvf;  
310     sRGB pixel;  
311  
312     for (int row = 0; row < NUMBER_OF_ROWS; row++)  
313     {  
314         for (int column = 0; column < NUMBER_OF_COLUMNS; column++)  
315         {  
316             pixel.b = trueColorImage[row][column][B];  
317             pixel.g = trueColorImage[row][column][G];  
318             pixel.r = trueColorImage[row][column][R];  
319  
320             hslvf = ConvertRGBtoHSLVf(pixel, param, useLimits);  
321  
322             // scale range {0..360} to {0..255}  
323             if ( hslvf.Hflag == HUE_IS_VALID )  
324             {  
325                 resultingHueImage[row][column][B] = (unsigned char) (255.0*hslvf.H/360.0);  
326                 resultingHueImage[row][column][G] = (unsigned char) (255.0*hslvf.H/360.0);  
327                 resultingHueImage[row][column][R] = (unsigned char) (255.0*hslvf.H/360.0);  
328             }  
329  
330             if (hslvf.Hflag == IS_DARK) // Color pixel as RED  
331             {  
332                 resultingHueImage[row][column][B] = 0;  
333                 resultingHueImage[row][column][G] = 0;  
334                 resultingHueImage[row][column][R] = 255;  
335             }  
336  
337             if (hslvf.Hflag == IS_OVERBLEACHED) // Color pixel as GREEN  
338             {  
339                 resultingHueImage[row][column][B] = 0;  
340                 resultingHueImage[row][column][G] = 255;  
341                 resultingHueImage[row][column][R] = 0;  
342             }  
343  
344             if (hslvf.Hflag == IS_GRAY) // Color pixel as BLUE  
345             {  
346                 resultingHueImage[row][column][B] = 255;  
347                 resultingHueImage[row][column][G] = 0;  
348                 resultingHueImage[row][column][R] = 0;  
349             }  
350         } // of column  
351     } // of row  
352 }
```

**Fig. 5.** Function converting original image to “Hue Map”.  
Depending on the flags, pixel is set as gray or as color

Two images on the bottom line in Fig. 6 present resulting “Hue Map” for the case when flag “UseLimits” was set to “FALSE” (bottom left image in Fig. 6) and when “UseLimits flag was set to “TRUE” (bottom right image in Fig. 6). Thus, bottom-left image is a “Hue Map” created by “standard HSL/HSV algorithm”. It can be seen that at the left part of this image, there is a “BLACK” strip. For the human observer, this black color is misleading – because the hue value of “RED” is also zero. All other pixels of this image has a constant value, equivalent to the Hue value of “GREEN”. So, by exploring this image, a human observer may arrive at the wrong conclusion that this image contains two colors – “RED” and “GREEN”. Bottom-right image, thus, presents a “Hue Map” created by a modified algorithm. It can be seen that at the left part of the image there is a “BLUE” strip – signaling that those pixels are “GRAY” (originally, their “rgb” values are {0,0,0}). Additionally, one can see a “GREEN” strip – signaling about “over bleaching” – in the pixels, for which “rgb” value is {0,255,0}. All other pixels of this map have a constant gray level, equivalent to the hue value of a “GREEN” color.

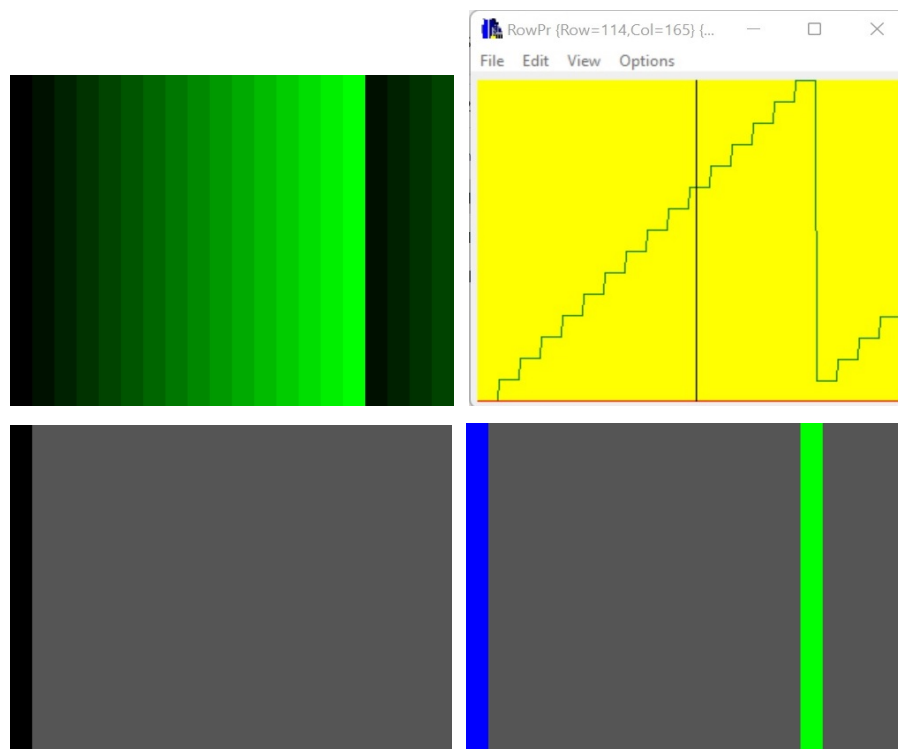


**Fig. 6.** Processing Synthetic Test Image: Green Digital Ramp

Upper: Left: Image as it seen by BMP Viewer. Left: Horizontal profile.

Bottom: Left: “Hue Map” created “without limits” – by using “standard HSL/HSV algorithm”. Right: “Hue Map created by modified algorithm.

Fig. 7 presents the same procedures, but in this case, pseudo-random noise is added to every pixel of the “Green Digital Ramp” synthetic image. The level of noise is set to “3”. Typical human observers can hardly see this noise in the “top-left” image in Fig. 7. This noise is clearly seen in the bottom-left image – profile of the typical line of the upper image. At the bottom part of this image, one can see the noise of “RED” and “BLUE” components of the synthetic image. Thus, on the top-right image in Fig 7 left strip is now “RED,” – signaling that pixel is “DARK”, and thus the hue value cannot be reliably calculated. Bottom-right image in Fig. 7 presents the profile of the typical line of the upper-right image. It can be seen that in the middle of the profile, the hue value is nearly noiseless (that is reliable). In the still “dark” regions (for example, at the left) hue value is correct but noisy. By selecting limit parameters, the user can fine-tune the application in order to provide better color recognition reliability..

Fig. 8 presents the real-life image of “Lemons and Mandarines” (upper left image). The profile of the exemplary line of this image shows that this real-life image has a visible noise, too dark pixels, and over-bleaching pixels. The upper-right image represents the “Hue Map”. The human observer can easily recognize yellow lemons, nearly red mandarines, green leaves, and a white wall in that background. An experienced human observer can easily recognize mandarins as having a nearly black gray level (the equivalent of a nearly “RED” color”). Lemons are seen on that image as more “GRAY”. Leaves have a typical level of a hue value of “GREEN”. This map, additionally, contains blue and green pixels. By comparing the “Hue Map” with the original image, it can be seen that a white wall is marked as “BLUE,” – signaling that those pixels are “GRAY”, and thus, a reliable hue value cannot be calculated for those pixels. Additionally, some pixels of the wall are marked as “GREEN” – that is, “over bleached”. For those pixels, the hue value cannot be reliably calculated too. It is obvious that machine vision applications can easily process modified “Hue Map” – by using “VALID” hue values to recognize an object by those values and ignore pixels for which Hflag differs from “0”.

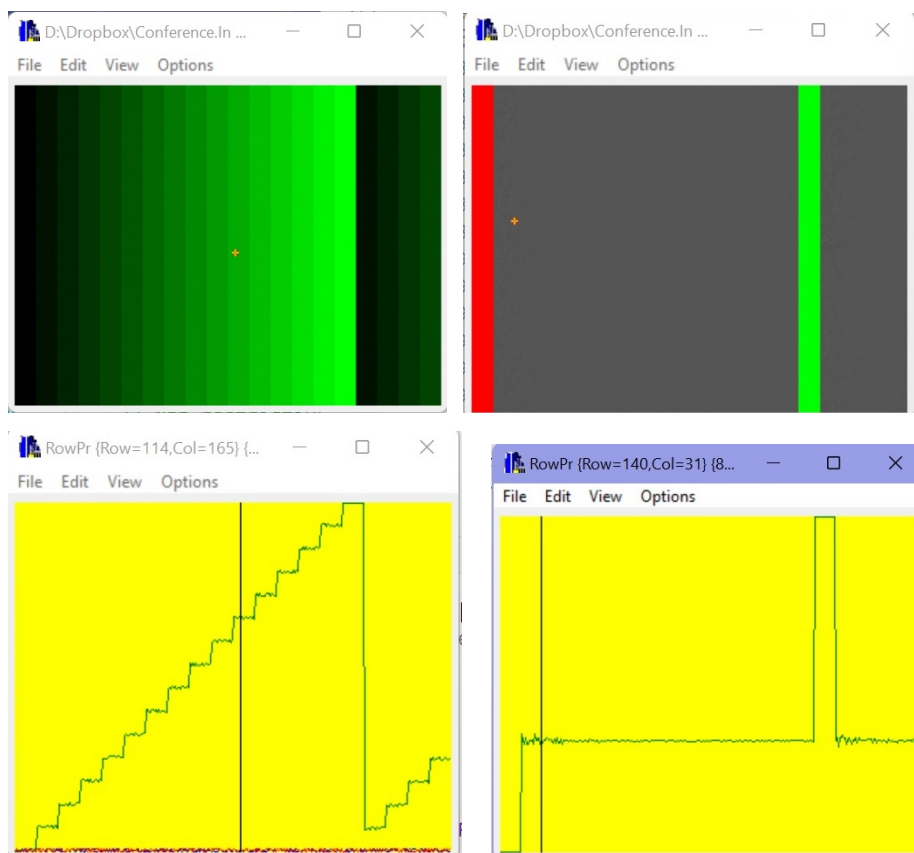


**Fig. 7.** Processing Synthetic Test Image: Green Digital Ramp with added pseudorandom noise



**Fig. 8.** Processing image of “Lemons and Mandarines”  
Cursors is set on the same position to validate that modified algorithms provides adequate results

#### 4. DISCUSSION AND CONCLUSIONS

The results described in this paper can be considered only as preliminary. However, it is clear that modifications of the classic color recognition algorithms described in this paper resulting in a more reliable color recognition as for a human observer as for machine vision applications. It is clear that to make additional conclusions about the quality of the modified algorithms, more images must be analyzed. Additionally, more tests with different values of the parameters of the algorithms used in this paper must be provided.

#### REFERENCES

1. Russ, JC, Neal FB, 2017, *The Image Processing Handbook*, CRC Press.
2. Jain, R, Kasturi, R, Schunck, DG 1995, *Machine Vision*, MIT Press and McGraw Hill, Inc.
3. Foley JD, van Dam, A, Feiner SK, & Hughes JF 1997, *Computer Graphics Principles and Practice. Second Edition in C*, ADDISON-WESLEY.
4. Kosolapov, S, 2007, Color edge based system and method for determination of 3d surface topology. US Patent 20070085849A1.
5. Kosolapov, S, Gershikov, E, Sabag, N, 2013, *Feasibility of a Camera-Based Instant Feedback System*, CONTENT 2013, The Fifth International Conference on Creative Content Technologies, ThinkMind, Valencis, Spain, pp 23-29.
6. Kosolapov, S, 2021, Comparison of Robust Color Recognition Algorithms, Journal of International Scientific Publications, Materials, Methods and Technologies, Volume 15, 2012, pp 274 – 283