Non-stop and where to find them

Mats Hoem Olsen

2/11-2021

Contents

List of Algorithms

1 Information about the picture

We can tell from the picture we were given that the camera used had the following properties:

1. Canon EOS 5D Mark II
2. firmware version 2.0.8
3. Automatic exposure
4. Canon EF 28-70mm f/2.8L USM lens
5. aperture in the range 3 to 32
6. the camera had an average temperature of 27c
7. focal length 52mm
8. colour space is "sRGB"
9. the picture was taken as 2/11/2011 at 15:22:27.

2 The theory of my approach

This section is divided into several progresses to handle the problem of the invisible Non-stops.

2.1 Grayscale

The formula used to calculate the grayscale of the image is:

$$\text{pixel} = \min(Green, Blue, Red)$$

2.2 Opening

[H] To perform the "opening" operation, we will use the following algorithm[5, page:185-186, 192]

Algorithm 1 Opening procedure

OPENING(A, H) :

1 **return** DILATE(ERODE$(A, H), H)$

These algorithms are written as follows:

Algorithm 2 Dilate procedure

DILATE(A,H) :

```
1  Create map A': M × N → {0, 1}
2  for (p) ∈ M × N
3        A'(p) = 0
4  for (q) ∈ H
5        for (p) ∈ A
6              A'(p + q) = 1
7  return A'
```

The Erode procedure as its own algorithm however it can be expressed as a extension of dilation.

Algorithm 3 Erode procedure

ERODE(A,H) :

```
1  Ī = INVERT(A)
2  H* = REFLECT(H)
3  I' = INVERT(DILATE(Ī, H*))
4  return I'
```

These will help us to remove noise from our picture, given a sutible sized H. This is to make sure that when we proceed we will not get small areas of interest that is not sutible.

2.3 Watershed

Watersheding is not a spesific algorithm but rather a idea of how to segment a blob[4]. The following is a prosegure of my own design[1].

It goes as follows:

[1] If it is an replication of another known algorithm, then it is accidental.

Algorithm 4 Watershed procedure

1 Find the contour of all elements in the picture
2 For every contour in picture, create a distance map from the edge of the contour.
3 All local maxima of the distance map is the centre of each blob.
4 Give each local maxima a unique label.
5 calculate a new distance map, but calculate the distence from the closest local maxima.
6 All pixels closest to a local maxima will inhered its label.

The algorithm will segment the picture according to the "center of mass"[2] of the object.

2.3.1 contour

The contour, like watershed, is not a single algorithm but rather a way proceeding. For this task I have choosen Moore-Neighbor tracing algorithm[2]. What

Algorithm 5 Moore-Neighbor tracing algorithm

CONTOUR(T)

```
1  B = ∅
2  Let M(a) be all pixels around pixel a in a clockwise order from where you entered.
3  Go from bottom to top, left to right until you hit a cell with value 1. This is "s".
4  B ← s
5  move back to the previous pixel.
6  set c to be the next pixel in M(p)
7  while c ≠ s
8       if c = 1
9            B ← c
10           p = c
11           backtrack
12      else
13           c is the next pixel in M(p)
14 return B
```

this is doing is moving along a figure like how we might go about finding out if a object is solid all around. By seeing that there excist a pixel on the boarder we move back and try to hit its neighbouring pixel by moving around the pixel we know is on the boarder.

There are many other alternatives, like:

[2]Every pixel is mass = 1, so the center of the "blob" is the average coordiante. Which in this case is the pixel(s) furterest away from the border.

- square-tracing
- Radical Sweep
- Theo Pavlidis' algorithm
- Full Contour Extraction[3]

But for the purpus of simplicity, I went for "Moore-Neighbor tracing" algorithm.

2.4 Major, and minor axis

To find the appropriate measure for circularity, we will use the proparty of ellipses; The major, and minor axis. For the rest of this paper, $\Re$ refers to the picture in question. We will apply the following theory:

2.4.1 Center moment of an object

Assuming that every pixel has a weight of 1 we can use following natation[5, p. 234]:

$$\bar{x} = \frac{1}{|\Re|} \sum_{(u,v)\in\Re} u$$
$$\bar{y} = \frac{1}{|\Re|} \sum_{(u,v)\in\Re} v$$

With

$$|\Re| = \sum_{(u,v)\in\Re} 1 = \mu_{0,0}$$

For a binary image, as is what we have, This might be written as

$$\mu_{0,0} = \sum_{(u,v)\in\Re} (u-\bar{u})^0 (v-\bar{v})^0$$

With this we can use the more general central moments for a binary image formula[5, page. 234]:

$$\mu_{p,q}(\Re) = \sum_{(u,v)\in\Re} (u-\bar{u})^p (v-\bar{v})^q$$

2.4.2 Major, minor formulas

To find the major and minor axis that can contain the object we must find the eigenvalues of the following matrix[5, page.238]

$$A = \begin{bmatrix} \mu_{2,0} & \mu_{1,1} \\ \mu_{1,1} & \mu_{0,2} \end{bmatrix}$$

Which is expresed by

$$\lambda_1 = \frac{\mu_{2,0} + \mu_{0,2} + \sqrt{(\mu_{2,0} - \mu_{0,2})^2 + 4\mu_{1,1}^2}}{2}$$
$$\lambda_2 = \frac{\mu_{2,0} + \mu_{0,2} - \sqrt{(\mu_{2,0} - \mu_{0,2})^2 + 4\mu_{1,1}^2}}{2}$$

The solution given will be used to give the major, and minor axis expressed as

$$r_{major} = 2 * \sqrt{\frac{\lambda_1}{|\Re|}}$$
$$r_{minor} = 2 * \sqrt{\frac{\lambda_2}{|\Re|}}$$

2.5 Measure of a ellipse

The circilarity of an object can be boiled down to two measures:

$$\text{circularity}(\Re) = 4\pi \frac{A(\Re)}{P^2(\Re)}$$
$$\text{round}(\Re) = \sqrt{\frac{\lambda_{2,\Re}}{\lambda_{1,\Re}}} = \frac{r_{minor,\Re}}{r_{major,\Re}}$$

Both is a valid way of measure if a object is round. However the last measure gives us a way of checking if a piece is broken by measuring if $|\Re| \ll \pi * r_{minor,\Re} * r_{major,\Re}$.

3 Implementation

I will start this section with the following warning:

The code is incomplete and only here for demonstration purposes.

3.1 Opening

The following code was used for the Closing procedure (figure 1)

Filter H get converted to a dictionary so we can work with relative coordinates rather than keeping track of where centre is of the filter H.

```
1 def Opening(I,H,max_int=255):
2     print("Opening")
3     return Dilate(Erodion(I,H,max_int),H,max_int)

4 def Erodion(I,H,max_int=255):
5     print("Erodion")
6     I_inv = Invert(I,max_int)
7     H_ref = Reflect(H)
8     new_I = Invert(Dilate(I_inv,H_ref,max_int))
9     return new_I

10 def Dilate(I,H,max_int=255):
11     """
12         H er oddetall i lengde og bredde med (0,0) i
               middle.
13         I er binært bilde.
14     """
15     print("Dilate")
16     length = len(I)
17     width = len(I[0])
18     new_I = [[0 for _ in range(width)] for _ in range(
           length)]
19
20     H_dict = conv2dict(H)
21
22     for H_value in H_dict:
23         if not(H_dict[H_value]):
24             continue
25         for row in range(length):
26             for col in range(width):
27                 if I[row][col] != max_int:
28                     continue
29                 if ((row + H_value[0]) >= length) or ((
                       col + H_value[1]) >= width):
30                     continue
31                 if ((row + H_value[0]) < 0) or ((col +
                       H_value[1]) < 0):
32                     continue
33                 new_I[row + H_value[0]][col + H_value[1]]
                       = max_int
34
35     return new_I
```

Figure 1: The closing procedure

```
1 def grayscale_img(img):
2     print("grayscale_img")
3     length = len(img)
4     width = len(img[0])
5
6     new_img = [[0 for x in range(width)] for y in range(
          length) ]
7
8     for row in range(length):
9         for v in range(width):
10            new_img[row][v] = colour2grayscale(img[row,v
                  ,0],img[row,v,1],img[row,v,2])
11
12    return new_img

13 def colour2grayscale(red,green,blue):
14    return int(min(red,green,blue))
```

Figure 2: The greyscale procedure

3.2 Grayscale

The following code was used for the greyscale procedure (figure 2) where we convert our colour image to grayscale by picking the lowest value from all the colour channels in that location.

This procedure is what produced image 10a. A valid alternative is to find the "length" of the colour

$$|\text{pixel}| = \sqrt{\frac{\text{Red}^2 + \text{Green}^2 + \text{Blue}^2}{3}}$$

However I found better results with my **colour2grayscale** code in this particular instance.

3.3 contour

There are many ways to apply a contour to a figure, however for my implementation I went for "Moore-Neighbor tracing" which was implements as shown in figure 3.

```
def moore_neighborhood(P, pre_cell):
    """
    p = (x,y), pre_cell = (x,y)
    """
    print("moore_neighborhood")
    MN = [(P[0]+x,P[1]+y) for x,y in [(-1, -1), (-1, 0),
        (-1, 1),(0, 1), (1, 1), (1, 0), (1, -1), (0, -1)]]
    MN_length = len(MN)
    start_clock = MN.index((pre_cell[0], pre_cell[1]))
    cell_array = MN[start_clock:] + MN[:start_clock]
    return cell_array



def contour_blob(I,X,label):
    """
        X er start koordinatet, I er (x,y):(V,label)
        return {(x,y):(V,label)}
    """
    print("contour_blob")
    B = {tuple(X):(I[X],label)}
    p = (X[0],X[1])
    s = X
    pre_cell = (p[0]-1,p[1])

    M = moore_neighborhood
    current_M = M(p, pre_cell)

    current_M.pop(0) # removes the pre_cell
    c = current_M.pop(0)
    while c != s:
        if I[c][0]:
            B.update({tuple(c):(I[c][0],label)})
            p = tuple(c)
            current_M = M(p, pre_cell)
            current_M.pop(0)

        pre_cell = tuple(c)
        if len(current_M) != 0:
            c = current_M.pop(0)
        else:
            break

    return B
```

Figure 3: Moore-Neighbor tracing algorithm, part 1

```
44 def contour(I):
45     """
46         I is a binary image with 0 and 1, where 0 is the
               background
47         returns {(x,y),(V,label)}
48     """
49     print("contour")
50     label = 2
51
52     contours = dict() # (x,y):(V,label)
53
54     current_pix = [0,0]
55
56     max_coord = list(max(I, key= lambda P: P[0]**2+P
           [1]**2))
57     max_coord[0] += 1 # to set it inside the margent
58
59     while current_pix != max_coord:
60             row = current_pix[0]
61             col = current_pix[1]
62
63             if I[tuple(current_pix)][0] and ((col,row)
                   not in contours):
64                 contours.update(contour_blob(I,(col,row),
                       label))
65                 label += 1
66                 current_pix = max([[x,y] for x,y in
                       contours if (contours[(x,y)][1] ==
                       contours[(col,row)][1]) and (y == row)
                       ],key= lambda P: P[0])
67                 current_pix[0] += 1
68
69             elif (col,row) in contours:
70                 current_pix = max([[x,y] for x,y in
                       contours if (contours[(x,y)][1] ==
                       contours[(col,row)][1]) and (y==row)],
                       key= lambda P: P[0])
71                 current_pix[0] += 1
72
73             current_pix[0] += 1
74             if current_pix[0] >= max_coord[0]:
75                 current_pix[0] = 0
76                 current_pix[1] += 1
77     return contours
```

Figure 4: Moore-Neighbor tracing algorithm, part 2

83 To save time does this code check if we have meet a contour before, then proceeds to find the other side of the contour. Essentially skipping over the object.

3.4 watershed

The watershed algorithm[3] described under is comprised of several algorithms as shown in figure 5.

```
1  def watershed(I):
2      """
3          I = {(x,y):(V,label)}
4      """
5      print("watershed")
6      segments = dict(I)
7
8      segments_contours = contour(segments)
9
10     segments_maximums = find_maximas(segments_contours)
11
12     segments = euclid_dist_map(segments,segments_maximum)
13
14     return segments
```

Figure 5: watershed code

3.4.1 Find all local maximum

To find all the local maximums in the 2D picture I used the property that

$$\text{local\_maximum}(\Re) \geq \text{Moore\_neighboorhood}((x,y) \in \Re)$$

This get reflected in the code shown in the figure 6.

[3]This code does not work in its completeness, however its essence follows the algorithm described in section 2.3

```
1 def maxima_check(I,X):
2     """
3     I = {(x,y):(V,label)}, X = (x,y)
4     """
5     print("maxima_check")
6     pix_next = [(X[0] + x,X[1] + y,I[(X[0] + x,X[1] + y)]
          if (X[0] + x,X[1] + y) in I else 0) for x in
          range(-1,1) for y in range(-1,1) if not((x == 0)
          and (y==0))]
7     return I[X] >= max(pix_next,key = lambda P: P[2])
8
9 def find_maximas(M):
10    """
11        M = {(x,y):(V,label)}
12    """
13    print("find_maximas")
14    maximas = dict()
15    for P in M:
16        if maxima_check(M,P):
17            maximas.update(P)
18
19    return maximas
```

Figure 6: local maximum code

3.4.2 euclidean distance mapping of contour

To find the euclidean distance I went for finding the shortest square distance from the contour. This is to save computing power as the square of a number retains its relative size to other numbers. The algorithm is shown in figure 7.

```
1  def euclid_dist_map(M, I):
2      """
3          I = {(x,y):(V, label)}, M = {(x,y):(V, label)}
4      """
5      print("euclid_dist_map")
6      labels_taken = []
7
8      e_dist_map = dict()
9
10     current_pix = [0,0]
11
12     max_width = max(M, key = lambda P: M[P][0])
13
14     max_coord = list(max(M, key = lambda P: P[0]**2+P
           [1]**2))
15     max_coord[0] += 1
16
17     while current_pix != max_coord:
18         if tuple(current_pix) not in M:
19             continue
20         if (I[tuple(current_pix)][1] not in labels_taken)
               :
21             current_label = I[tuple(current_pix)][1]
22             labels_taken.push(current_label)
23             border_for_label = {p:I[p] for p in I if I[p
                   ][1] == current_label}
24             pix_seed = tuple(current_pix)
25             for P in (i for i in moore_neighborhood(
                   pix_seed,(pix_seed[0]+1,pix_seed)) if i
                   not in border_for_label): # henter piksel
                   inni contour, ellers så vil bakgrunnen bli
                    "fyllt" istedet.
26                 if contained(P, border_for_label):
27                     pix_seed = P
28                     break
29             updated_I = b_floodfill(border_for_label,
                   pix_seed, current_label, euclid_dist_point)
30             e_dist_map.update(updated_I)
31
32         elif I[tupel(current_pix)][1] in labels_taken:
33                 current_pix = max([[x,y] for x,y in I if
                       (I[(x,y)][1] == I[(col,row)][1]) and (
                       y == current_pix[1])],key= lambda P: P
                       [0])
34                 current_pix[0] += 1
35
36         current_pix[0] += 1
37         if current_pix[0] > max_width:
38             current_pix[0] = 0
39             current_pix[1] += 1
40     return e_dist_map
```


Figure 7: euclidean distance code

To make sure that the point that gets evaluated is inside the contour I use the algorithm described by Darel Rex Finley[1]. The algorithm is reflected in the figure 8.

```
1  def euclid_dist_point(I,X):
2      """
3          I = {(x,y):(V,label)}
4          X = (x,y)
5      """
6      print("euclid_dist_point")
7      min_dist = X[0]**2+X[1]**2
8      for Y in I:
9          calc = (X[0]-Y[0])**2 + (X[1]-Y[1])**2
10         if calc < min_dist:
11             min_dist = calc
12     return min_dist

13 def contained(X,I):
14     """
15         I, contour, {(x,y):(V,label)}
16         X, point, (x,y)
17     """
18     print("contained")
19     i = j = len(I)-1
20
21     polyX,ployY = [],[]
22
23     for P in I:
24         polyX.append(P[0])
25         polyY.append(P[1])
26
27     odd_nodes = False
28
29     for i in range(len(I)):
30         if ((ployY[i] < X[1]) and (polyY[j] >= X[1])) or
               ( (ployY[j] < X[1]) and (polyY[i] >= y) ):
31             if (polyX[i] + (y-polyY[i])/(polyY[j]-polyY[i
                   ]) * (polyX[j]-polyX[i])) < X[0]:
32                 odd_nodes = not(add_nodes)
33         j = i
34     return odd_nodes
```

Figure 8: euclidean distance code part 2, with check for containment of point.

4 The apology

Even though I have all the theory I need, did not have the time to solve a few problems with the written code, due to poor design chooses early in the development. Therefore I give you my process in ImageJ.

5 Analyses of picture

Steps taken in solving the problem of finding .

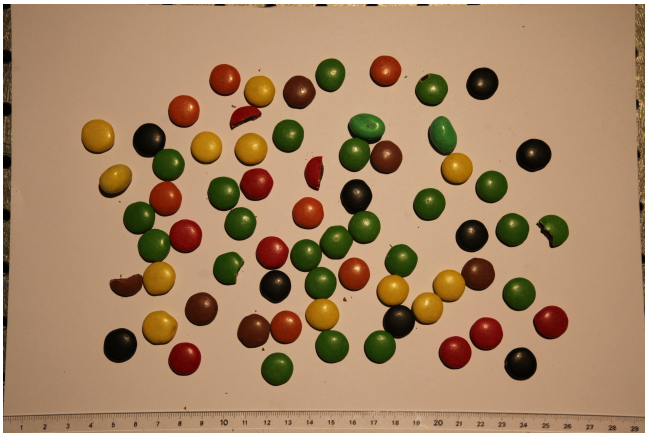
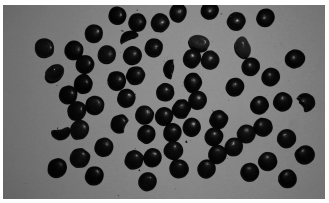
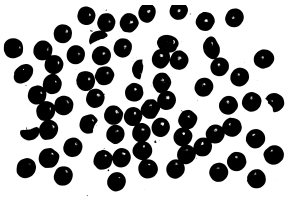


Figure 9: Picture before treatment.

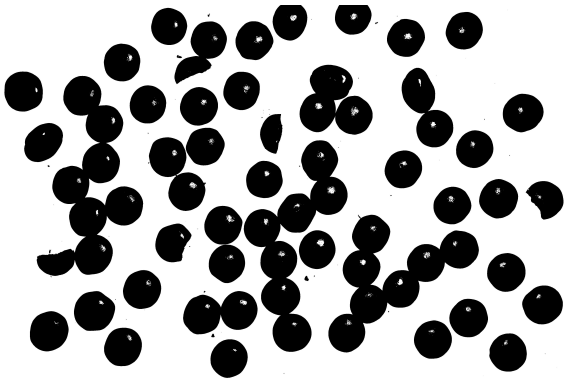
To start we crop the picture so we can focus on the chocolate. Even a ruler can throw our analyses off track. Once that is done we will make a binary of the image, this is done by taking the lowest value among the colour channels and then perform a Outsu-threshold. Further more we shall dilate and erode the picture to remove small particles.



(a) home-made greyscale



(b) binary treatment of Outsu-thresholding from home-made greyscale



(c) binary treatment of Outsu-thresholding from home-made greyscale

After that we will use ”flood fill” to remove the ”holes” in the picture. This step can be done with finding the contour of all figures, and then perform ”flood fill” to set every pixel to appropriate values. What remains is to label the areas that is a individual chocolate. This can be done with Erosion, as there are some chocolates that are too close together. We will perform this until all significant particles are removed, then we dilute equal number of times with the same filter. Although this will shrink the size of the chocolate, it will not remove the centre off the chocolate which can be used to identify the location of the chocolate. Once that is done we will use Watershed to separate the last individuals into separate chocolates. Once that is done we will perform our analyses of the chocolates.

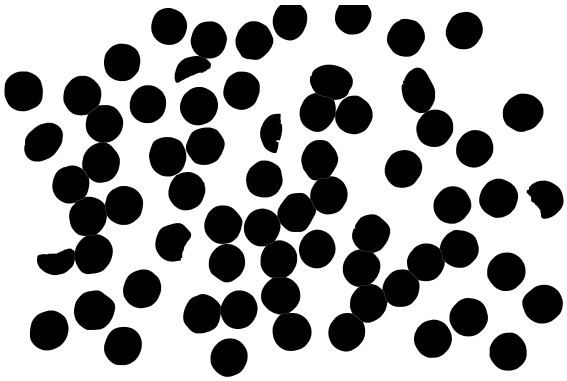


Figure 11: Result after several erosions, and dilations. This had a side-effect of removing holes. However this could had been done by a "fill-hole" procedure.

When analysing the tabell generated from "analyze particles" menu we get the following table 1

Label	Roundness	Label	Roundness
1	0.930	35	0.775
2	0.824	36	0.964
3	0.969	37	0.949
4	0.924	38	0.909
5	0.949	39	0.923
6	0.950	40	0.970
7	0.957	41	0.952
8	0.964	42	0.954
9	0.500	43	0.795
10	0.789	44	0.936
11	0.711	45	0.988
12	0.970	46	0.896
13	0.934	47	0.949
14	0.935	48	0.978
15	0.948	49	0.953
16	0.962	50	0.594
17	0.944	51	0.942
18	0.909	52	0.975
19	0.956	53	0.949
20	0.961	54	0.955
21	0.944	55	0.947
22	0.538	56	0.959
23	0.742	57	0.940
24	0.947	58	0.960
25	0.938	59	0.928
26	0.934	60	0.925
27	0.899	61	0.971
28	0.931	62	0.914
29	0.928	63	0.972
30	0.967	64	0.950
31	0.958	65	0.977
32	0.939	66	0.931
33	0.962	67	0.979
34	0.964	68	0.941

Table 1: Table extracted from ImageJ

The labels refers to figure 12

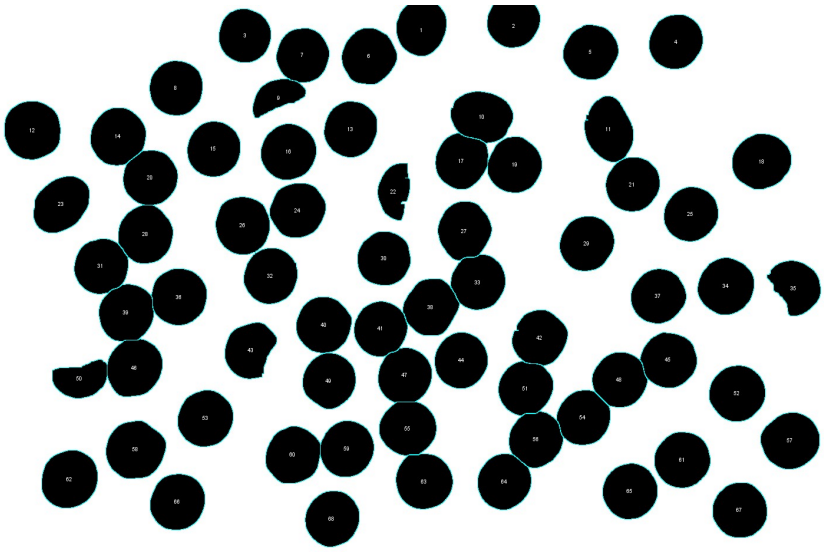


Figure 12: Labeled figure from imageJ

If we set the limit roundness to be 0.8 we find that we get the following tabel:

label	Roundness
9	0.5
10	0.789
11	0.711
22	0.538
23	0.742
35	0.775
43	0.795
50	0.594

Table 2: constricted table

From figure 12 compared with table 2 we find this believable as we know that 10, 11, and 23 are M&M's. The other pieces are broken pieces, some do vividly look elliptical which makes the analyses a great deal harder. The pieces in the 0.5 to 0.6 range are clearly broken Non-stop pieces.

6 Conclusion

From the analyses we can conclude that examinating Roundness (section 2.5) we can narrow down which pieces are NoN-stop, broken NoN-stop, and M&M's. In this case the pieces not fit for export are the pieces found in table 2.

References

[1] Darel Rex Finley. *Determining Whether A Point Is Inside A Complex Polygon*. Point-In-Polygon Algorithm — Determining Whether A Point Is Inside A Complex Polygon. 2007. URL: http://alienryderflex.com/polygon/ (visited on 10/31/2021).

[2] Abeer George Ghuneim. *Contour Tracing*. Contour Tracing. 2000. URL: http://www.imageprocessingplace.com/downloads_V3/root_downloads/tutorials/contour_tracing_Abeer_George_Ghuneim/moore.html (visited on 10/31/2021).

[3] Paul J. Shin et al. "An efficient algorithm for the extraction of contours and curvature scale space on SIMD-powered Smart Cameras". In: *2008 Second ACM/IEEE International Conference on Distributed Smart Cameras*. 2008 Second ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC). Palo Alto, CA, USA: IEEE, Sept. 2008, pp. 1–10. ISBN: 978-1-4244-2664-5. DOI: 10.1109/ICDSC.2008.4635714. URL: http://ieeexplore.ieee.org/document/4635714/ (visited on 11/01/2021).

[4] *watershed algorithm : definition of watershed algorithm and synonyms of watershed algorithm (English)*. 2012. URL: http://dictionary.sensagent.com/watershed%20algorithm/en-en/ (visited on 10/31/2021).

[5] Wilhelm Burger and Mark J. Burge. *Digital Image Processing*. 2nd ed. Texts in Computer Science. London: Springer, London. 811 pp. ISBN: 978-1-4471-6684-9.