CA23125 - The mETamaterial foRmalism approach to recognize cAncer (TETRA); Working Groups 2, 2026

Mathematical modelling of highly disordered anisotropic structures.

Part 2.3. Classification algorithm for automatic detection for cancerous zones in tissue.

Vladimir Mityushev

July 23, 2026

Abstract

The introduction to selected aspects of DIP (Digital Image Processing) is presented. Further we employ this knowledge to construct image processing pipelines for specific images. We illustrate all concepts using Python programming language. As a short introduction, we can recommend an appendix of our book [17]. You can also study free Scientific Python Lectures [3]. Complete examples of the code from this part can be downloaded from [1]. You can run it and experiment. We use the OpenCV (Open Computer Vision) library for image processing. We learn the specific algorithms in the library that are useful for image processing of tissues and selecting parts suspected to be cancer cells. We focus on cells in medical applications. The developed *a*RVE methodological, see Parts 2.1 and 2.2, and practical application enables a systematic treatment of the variability in random composites within the framework of homogenization principles and asymptotic analysis. Application to the diagnosis and classification of glioma structures requires a set of corresponding pictures to apply the developed machine learning classification algorithm. To summarize, the theoretical basis and algorithms have been developed.

In order to go further to practical problems we need:

- physical constants of the image components;
- a number of healthy and diseased samples to process their and use for the classification of new pictures by supervised machine learning.

1 Digital Image Processing (DIP)

We begin with the main concepts of DIP concerning thresholding, detection of zones and their classification. They will be used in practical applications. We want to be manageable due to a specific aim in our head, and since DIP is a vast discipline, there are already excellent resources that treat this topic in great generality. You can consult [12] for very precise mathematical exposition or more concise yet still extremely precise [16].

We will illustrate concepts constructing Python code with the help of OpenCV library [2], a robust and well-optimized set of image processing tools for real-world applications. The library is successively amended by new algorithms and methods and is a de facto standard in Image Processing. The library is written in C language due to optimization and possible applications in embedded systems. However, a version for Python language allows us to use the benefits of high-level language if we do not need real-time responses. There are many excellent and practical books that introduce to the library, e.g., [21, 13], and reference books, e.g., [15]. The package Mathematica is also used in DIP. It is worth noting that Python and Mathematica complement each other to construct algorithms for automatic detection for zones in tissue.

In the following subsections, you will be introduced to all necessary topics from DIP to gain working knowledge for processing issues images, materials images, and biological microscopic images.

1.1 Thresholding

Having a grayscale image, we want to extract some shapes of a given intensity level. This can be done by selecting pixels above the fixed (intensity) threshold. We can use histograms to specify the threshold or use automatic methods.

The general idea of thresholding θ_t is to modify pixel-wise the value depending on the value of threshold t and maximal value V (typically the maximal intensity $V = 255$). In a simple binary thresholding we have

$$\theta_t^V(f)(x,y) = \begin{cases} 0 & \text{if} \quad f(x,y) < t \\ V & \text{if} \quad f(x,y) \geq t \end{cases} \tag{1.1}$$

We can also invert the values. This is the most useful in creating masks.

First, we create an image that contains two types of granules/circles of different sizes. We, moreover, add a disturbance to the background. We then convert the image to the grayscale. In real life, the complexity is larger

```python
import numpy as np
import matplotlib.pylab as plt
import cv2

#prepare figure
image = np.zeros((512,512,3), np.uint8)
image = image + 255

#perturbation
cv2.rectangle(image,(100,0),(500,400),(240,240,240),-1)
cv2.rectangle(image,(0,100),(400,500),(0,240,240),-1)

#generate class 1
x = 512* np.random.rand(200)
y = 512* np.random.rand(200)
for i,j in zip(x,y):
    cv2.circle(image,(int(i),int(j)), 5, (100,100,100), -1)

#generate class 2
x = 512* np.random.rand(50)
y = 512* np.random.rand(50)
for i,j in zip(x,y):
    cv2.circle(image,(int(i),int(j)), 20, (10,10,10), -1)

imageGray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imwrite("output/initialImageThresholding.png", imageGray)
```

which produces Figure 1. The next is to construct histogram of intensity distribution

```python
histGray= cv2.calcHist([imageGray],[0],None,[256],[0,256])
plt.plot(histGray/max(histGray))
plt.xlabel("bin")
plt.ylabel("normalized cout")
plt.savefig("output/GrayHistogramThresholding.png")
plt.show()
```

that gives Figure 2. We can note in Figure 2 the bin close to zero representing bigger darker balls in Figure 1, and the second bin closer to 100 that represents smaller, lighter balls. These two kinds of balls are granules we

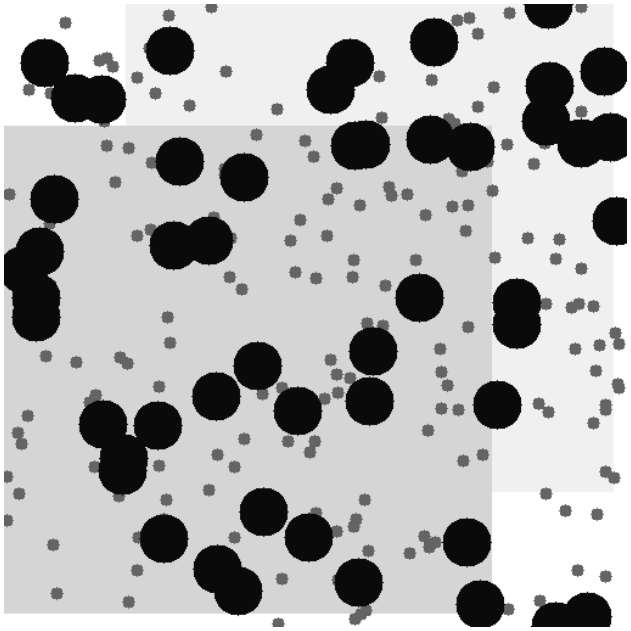


Figure 1: Initial image converted to grayscale.

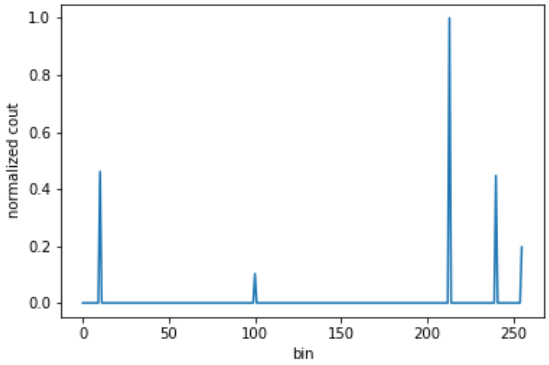


Figure 2: Intensity histogram for image 1.

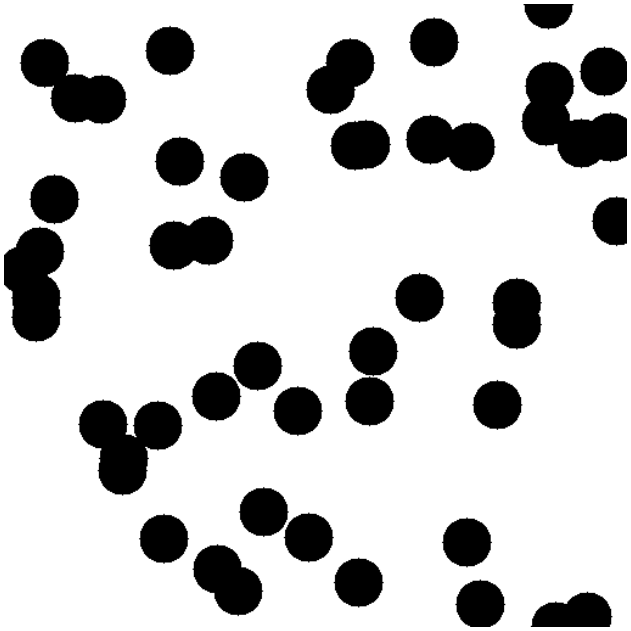


Figure 3: Extraction of larger granules from Figure 1.

want to extract. Moreover, we have three bigger peeks above 200: two light rectangles of big area and the last one at 255 represents the background white color.

The first kind of granules can be extracted easily by setting the threshold value to, e.g., 50. The pixels equal to or above this value will be set to white. We can do it using the following code

```
_ , mask1 = cv2.threshold(imageGray, 50, 255, cv2.THRESH_BINARY)
cv2.imwrite("output/MaskThresholding1.png", mask1)
```

which produces Figure 3, which can be used as a mask for selecting the first kind of balls.

For selecting the second kind of balls we have to subtract from Figure 1 the balls from image 3. That can be done by negating the mask 3 and using exclusive OR (XOR) binary operation with the original image 1. Then we can do thresholding below, e.g., 150

```
_ , mask2 = cv2.threshold(cv2.bitwise_xor(imageGray,
cv2.bitwise_not(mask1)), 150, 255, cv2.THRESH_BINARY)
cv2.imwrite("output/MaskThresholding2.png", mask2)
```

that produces Figure 4. This figure can also be used as a mask for extracting smaller balls. One can observe that some of these balls were shadowed by

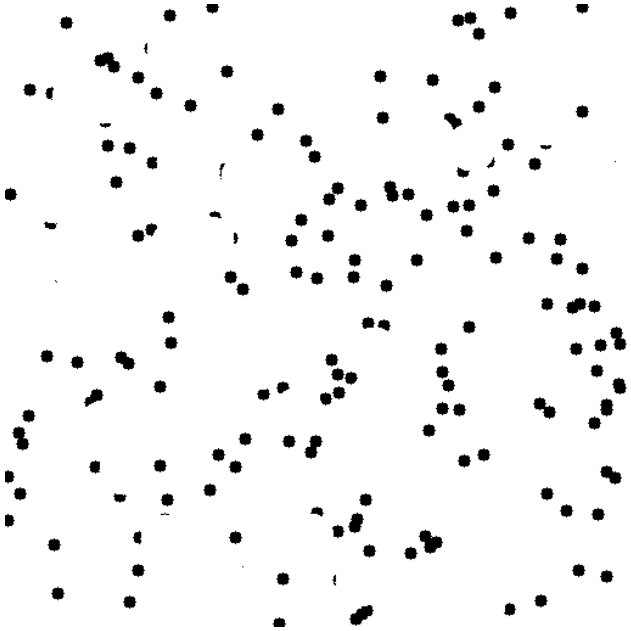


Figure 4: Extraction of smaller granules from Figure 1.

the darker, bigger ones, and therefore, they are only partially restored. We can remove these parts using morphological operations as described above, however, we will not do it here. At this point we have desired *segmentation* of the image into two classes of granules.

The same extraction can be done using `cv2.inRange` function that is supplied with the boundaries of the intensity interval from which we extract pixels as in the following code

```
mask2 = cv2.bitwise_not(cv2.inRange(imageGray, 50, 150))
cv2.imwrite("output/MaskThresholding2.png", mask2)
```

We also inverted colors to have granules black and color white (opposite to the colors used in a mask). The function `cv2.inRange` can also be used to threshold color images, and then the boundary points are BGR vectors for minimal and maximal values of each color channel.

We can now focus on examining automatic thresholding. We first examine Otsu's method [18], which tries to split pixels into different classes and tries to maximize in-between-class variance, see, e.g., [12]. It is realized, but the following code

```
_, imageOtsu = cv2.threshold(imageGray, 0, 255, cv2.THRESH_OTSU)
cv2.imwrite("output/MaskThresholdingOtsu.png", imageOtsu)
```

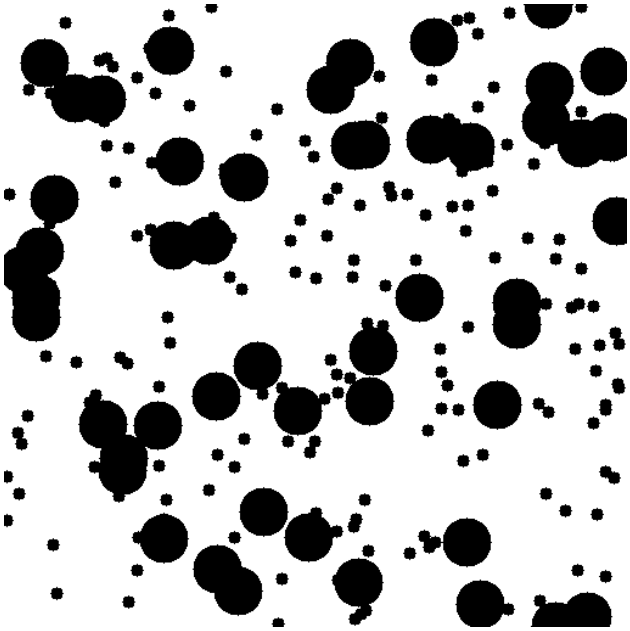


Figure 5: Otsu's method applied to Figure 1.

that produces Figure 5 that removes background.

We can also use adaptive thresholding using the mean local value of pixels minus a constant. We have to provide the size of the local neighborhood. This is realized by

```
imageAdaptiveMean = cv2.adaptiveThreshold(imageGray,255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY,51,1)
cv2.imwrite("output/MaskThresholdingAdaptiveMean.png",
imageAdaptiveMean)
```

that produces Figure 6. The last two parameters in `cv2.adaptiveThreshold` are the size of the local neighborhood and the constant C that is subtracted from the local mean, giving the local threshold.

One can note that some artifacts resulting from removing background rectangles are present.

We can also apply Gaussian adaptive thresholding, which determines the threshold as Gaussian-weighted local neighborhood values minus a constant. This algorithm is applied in the following code

```
imageAdaptiveGaussian = cv2.adaptiveThreshold(imageGray,255,
cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY,51,1)
cv2.imwrite("output/MaskThresholdingAdaptiveGaussian.png",
```

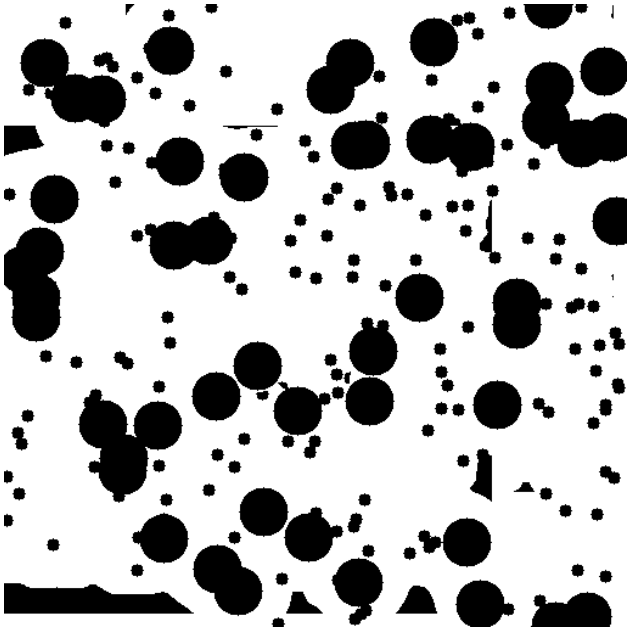


Figure 6: Adaptive mean thresholding applied to Figure 1.

```
4 imageAdaptiveGaussian)
```

that produces Figure 7. In this case, also, some artifacts are visible.

Summing up, the basic automatic methods are less helpful in extracting granules in such a simple example.

1.2 Contours

Having extracted granules, we want to characterize them. In this section we will learn how to extract contours, fit geometric shapes to them, and compute moments of contours.

First, we must prepare the image. We use an academic example of random balls. The difficulty will be with overlapping balls. The following code

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pylab as plt
4 import cv2
5
6 #prepare figure
7 image = np.zeros((512,512,3), np.uint8)
8 image = image + 255
```

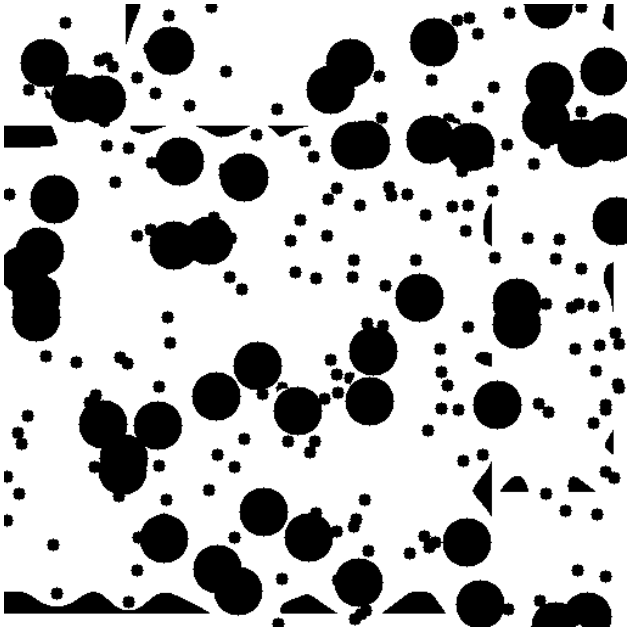


Figure 7: Adaptive gaussian thresholding applied to Figure 1.

```
9
10  #generate granules
11  np.random.seed(1)
12  x = 512* np.random.rand(50)
13  y = 512* np.random.rand(50)
14  for i,j in zip(x,y):
15      cv2.circle(image,(int(i),int(j)), 20, (255,0,0), -1)
16
17  cv2.imwrite("output/initialImageContours.png", image)
```

creates Figure 8.

First, we create a grayscale mask

```
1  imageGray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
2  imageGray = cv2.bitwise_not(imageGray)
3  cv2.imwrite("output/maskImageContours.png", imageGray)
```

that gives Figure 9. Since some granules are merged we can try to disconnect them using erosion multiple times. However, overlapping is too big in some cases to separate them by erosion without significantly removing white pixels from well-separated images. Moreover, in real applications, granules have

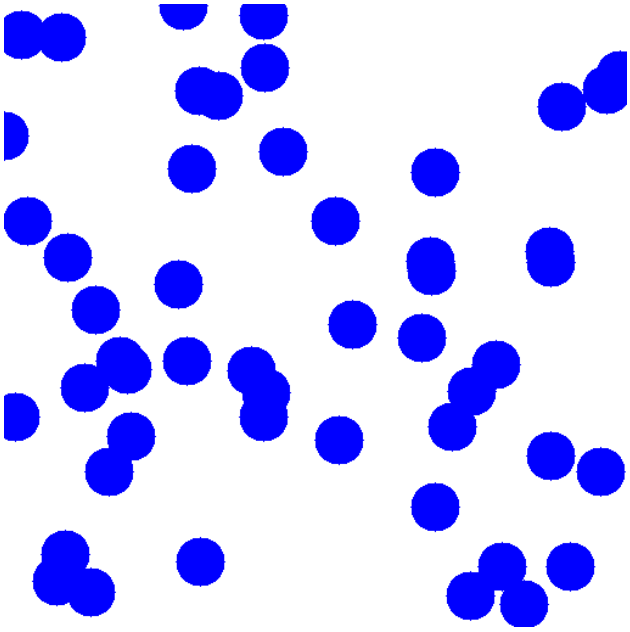


Figure 8: Initial image for detecting contours.

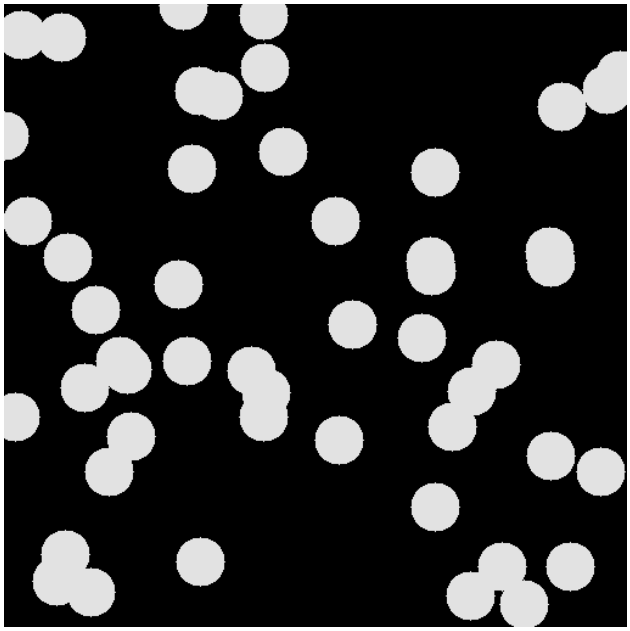


Figure 9: Grayscale image of Figure 8.

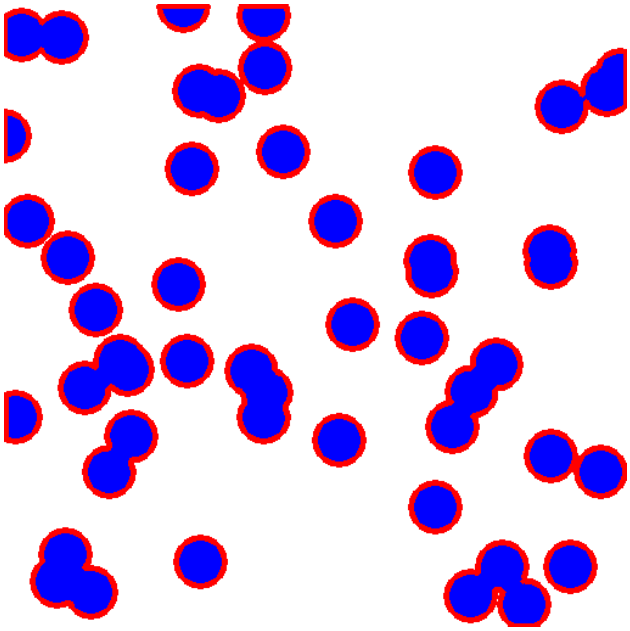


Figure 10: Detected contours (red) from Figure 8.

irregular shapes and are connected to such an extent that they are impossible to distinguish even by the human eye.

OpenCV has many specialized methods that can detect contours. Contours are lists of points that sit at the edge of shapes. In the simplest case, the algorithm tries to find the contours(borders) of connected components. We use a general one for detecting contours on grayscale image

```
contours, _ = cv2.findContours(imageGray, cv2.RETR_LIST,
cv2.CHAIN_APPROX_TC89_L1 )
#draw contours
imageGrayContours =  image.copy()
#imageGrayContours =  np.zeros(image.shape)+255
cv2.drawContours(imageGrayContours, contours,-1,(0,0,255),3)
cv2.imwrite('output/contoursImageContours.png',
imageGrayContours)
```

which gives Figure 10. One can observe that some blobs are treated as a single connected component.

The function `cv2.findContours` has various options and types of returned data. For detecting granules we need only a flat structure of the list (option `cv2.RETR_LIST`). In general, it can return a hierarchical structure of contours using `cv2.RETR_TREE` instead. We can also use various ap-

proximation methods that optimize the number of points. Since we need many points in order to match ellipses to the contours, we used option `cv2.CHAIN_APPROX_TC89_L1` which uses the Teh-Chin chain approximation algorithm [20]. We can use other Teh-Chin algorithm by option `cv2.CHAIN_APPROX_TC89_KCOS`, or simple approximation with a small number of points `cv2.CHAIN_APPROX_SIMPLE`, or no approximation `cv2.CHAIN_APPROX_NONE` - stores all points of the contours.

The contour is drawn on the image `imageGrayContours` all (-1 contours of all indices) detected contours using a red line of thickness 3.

Since we know that the contours are balls, we can apply more specialized algorithms to detect circles that are available in OpenCV. We use the method based on Hough transform [8]

```
img = imageGray.copy()
colorimg = image.copy()
circles = cv2.HoughCircles(img,cv2.HOUGH_GRADIENT, 1, 20,
param1=20, param2=10, minRadius=2, maxRadius=50)
circles = np.uint16(np.around(circles))
for circle in circles[0,:]:
    # draw the outer circle
    cv2.circle(colorimg,(circle[0],circle[1]),circle[2],
    (0,0,255),2)
    # draw the center of the circle
    cv2.circle(colorimg,(circle[0],circle[1]),3,(0,0,0),3)
cv2.imwrite('output/HoughCirclesDetectorImageContours.png',
colorimg)
```

that produces Figure 11. One can note that more balls are detected, however, the algorithm does not spot some overlapping granules.

The list of circles returned by the `cv2.HoughCircles` method represented by x and y coordinates (indices 0 and 1) of the center and the radius (index 2).

Let us return to contours, as they are more general types of closed curves used for bonding regions in the images. We want to draw around the contour bounding rectangles. Moreover, we cut the rectangular image enclosing the contours and save them in images, moreover, we enumerate detected contours. This is realized by the following code

```
img = image.copy()
for idx, contour in enumerate(contours):

```

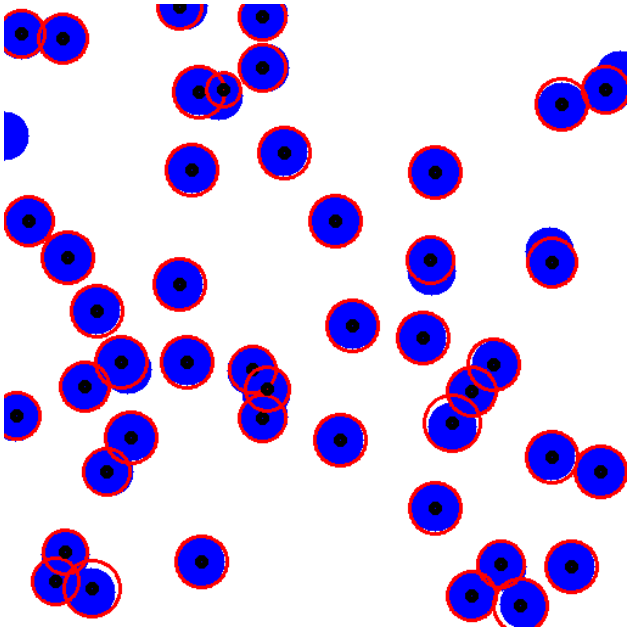


Figure 11: Hough algorithm for detecting circles applied to Figure 8.

```
    #draw minimum area rectangle
    minRect = cv2.minAreaRect(contour)
    box = cv2.boxPoints(minRect)
    box = np.int0(box)
    cv2.drawContours(img, [box], -1, (0,0,255), 1)
    #enumarate contours
    cv2.putText(img, '{}'.format(idx), (box[0][0], box[0][1]),
    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,0,0), 1, cv2.LINE_AA)

    #extract image from bounding rectangle and write to files
    x, y, w, h = cv2.boundingRect(contour)
    cropped = image[y:y + h, x:x + w, :].copy()
    feature = cv2.resize(cropped, (20, 20))
    cv2.imwrite("./output/figs/{}.png".format(idx), feature)
cv2.imwrite('output/MinAreaRectangleImageContours.png', img)
```

which produces Figure 12. Extracted images can be used for (unsupervised) training Machine Learning algorithms to discriminate their types if there are more types of them. We will return to Machine Learning later in this part.

When the granules have no circular symmetry, we sometimes want to know their orientation and non-symmetry. The easiest way is to fit ellipses and extract their parameters. OpenCV has a function that can fit an ellipse

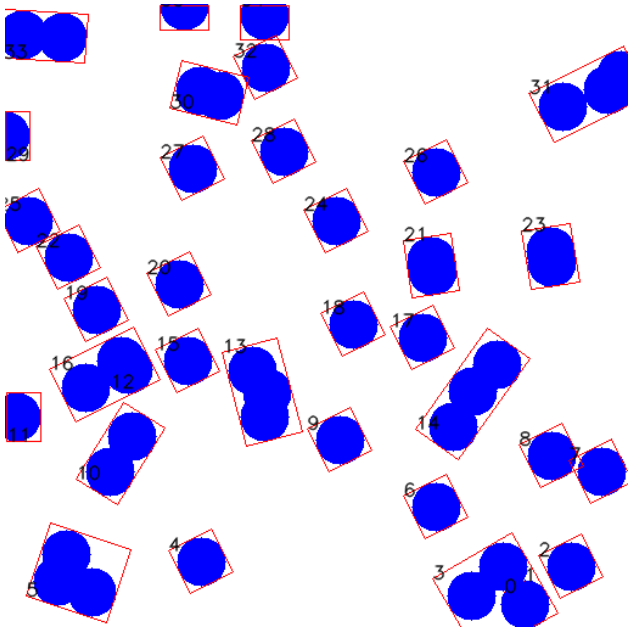


Figure 12: Minimal area rectangle for detected contours from Figure 10.

using at least five points of contours using the least square method. We do it in the following code

```
img = image.copy()
ellipses = [] #list for storing ellipsis parameters
for idx, contour in enumerate(contours):
    if contours[idx].size < 10 :
        continue
    ellipse = cv2.fitEllipse(contour) # ellypse is ((x,y),
    (majorAxis, minorAxis), angle)
    #remove bigger ellipses
    if max(ellipse[1]) < 200:
        ellipses.append({"id": idx, "x":ellipse[0][0],
        "y":ellipse[0][1], "majorAxis": ellipse[1][0],
        "minorAxis": ellipse[1][1], "angle": ellipse[2]})
        cv2.ellipse(img, ellipse, (0,0,255), 3, cv2.LINE_AA)
        cv2.putText(img, '{}'.format(idx), (int(ellipse[0][0]),
        int(ellipse[0][1]) ), cv2.FONT_HERSHEY_SIMPLEX,
        0.2, (0, 0, 0), 1, cv2.LINE_AA)
#save parameters of ellipses
df_ellipses = pd.DataFrame.from_records(ellipses)
df_ellipses.to_csv("output/ellipses.csv")
```

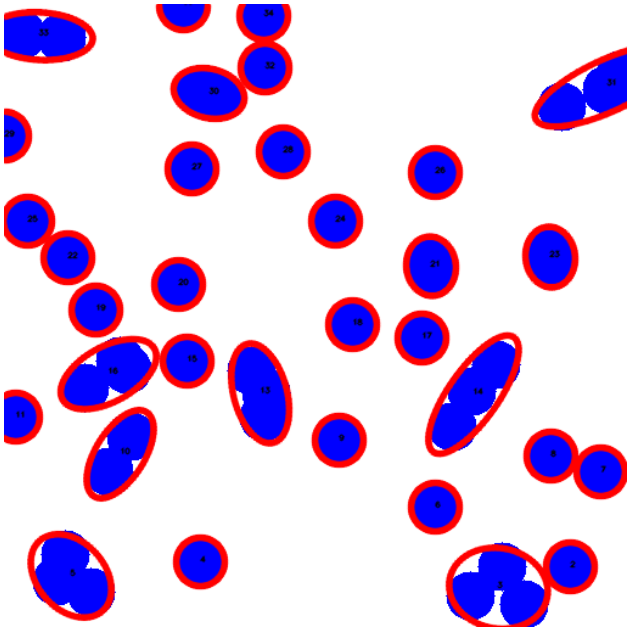


Figure 13: Fitted ellipses to contours from Figure 10.

```
20 #write image with elipses
21 cv2.imwrite('output/EllipsesImageContours.png', img)
```

that produces the image in Figure 13. The ellipse is characterized by the center (x, y), major axis, minor axis, and orientation given by the angle between the positive $0X$ axis and the major axis. In the code, these data are packed into a Python `DataFrame` structure and saved in `ellipses.csv` file[1]

Sometimes, we want to characterize contours more precisely. We can use moments of distribution. For a contour $c(x, y)$ we can get *static real moments* (sometimes called '*spatial or raw moments*')

$$m_{ij} = \sum_{x,y} c(x, y) x^i y^j, \tag{1.2}$$

central moments

$$\mu_{ij} = \sum_{x,y} c(x, y)(x - \bar{x})^i (y - \bar{y})^j, \tag{1.3}$$

[1]CVS (comma-separated values) is a simple and portable file format for saving tabular data in a text file. Each row of the table corresponds to the row of the file, and columns are represented by data separated by a comma or other separator. These files can be easily imported into a spreadsheet program.

where the 'mass center' is characterized by

$$\bar{x} = \frac{m_{10}}{m_{00}}, \quad \bar{y} = \frac{m_{01}}{m_{00}}, \tag{1.4}$$

and *normalized central moments* ($\nu ='' nu''$) defined as

$$\nu_{ij} = \frac{\mu_{ij}}{m_{00}^{(i+j)/2+1}}. \tag{1.5}$$

Since $\mu_{00} = m_{00}$, $\nu_{00} = 1$, $\nu_{10} = \mu_{10} = \nu_{01} = \mu_{01} = 0$, so these values are not returned.

The considered static real moments correspond to the static complex moments introduced in Part 2.2. The complex Green's formulas allow us to treat line integral over a closed contour as a surface area over the area enclosed by the contour. The same holds for the static real moments.

The contours with self-intersections can have zero m_{00} even if the area is not zero due to the implicit orientation of areas in Green's formula. Moreover, the value depends on rasterization, and therefore, the computed values can be treated only as approximations. Moreover, the more points of the contour yield a more accurate approximation.

The contours (viewed as continuous curves/before discretization) can be transformed by scaling, translations, and rotations (after undoing discretization on the raster lattice). We are interested in having the same moments for two of the same contours, even if one is a transformed version of the second one. Therefore, in analyses, we should use moments that are invariant under specific symmetry groups. A well-known fact is that the central moments are invariant with respect to translations. The moments ν_{ij} are invariant with respect to translations as defined by central moments, and also with respect to scaling. The natural next step is to define the invariant moments concerning scaling, rotations, and translations. These seven moments were defined by Hu [14], and then amended [9] and generalized for spatial objects

[10]. *Hu moment invariants* are defined as

$$
\begin{aligned}
I_1 &= \nu_{20} + \nu_{02}, \\
I_2 &= (\nu_{20} - \nu_{02})^2 + 4\nu_{11}^2, \\
I_3 &= (\nu_{30} - 3\nu_{12})^2 + (3\nu_{21} - \nu_{03})^2, \\
I_4 &= (\nu_{30} + \nu_{12})^2 + (\nu_{21} - \nu_{03})^2, \\
I_5 &= (\nu_{30} - 3\nu_{12})(\nu_{30} + \nu_{12})[(\nu_{30} + \nu_{12})^2 - 3(\nu_{21} + \nu_{30})^2] + \\
&+ (3\nu_{12} - \nu_{03})(\nu_{21} + \nu_{03})[3(\nu_{30} + \nu_{12})^2 - (\nu_{21} + \nu_{03})^2], \\
I_6 &= (\nu_{20} - \nu_{02})[(\nu_{30} + \nu_{12})^2 - (\nu_{21} + \nu_{03})^2] + 4\nu_{11}(\nu_{30} + \nu_{12})(\nu_{21} + \nu_{03}), \\
I_7 &= (3\nu_{21} - \nu_{03})(\nu_{30} + \nu_{12})[(\nu_{30} + \nu_{12})^2 - 3(\nu_{12} + \nu_{03})^2] - \\
&(\nu_{30} - 3\nu_{12})(\nu_{21} + \nu_{03})[3(\nu_{30} + \nu_{12})^2 - (\nu_{21} + \nu_{03})^2].
\end{aligned}
\tag{1.6}
$$

They are not all independent. Moreover, I_7 is reflection antisymmetric and the remaining ones reflection symmetric, they can be used to detect mirror symmetry of the contour. In addition, there are more invariants of fourth order [9].

The Hu moment invariants can be computed in OpenCV using `cv2.HuMoments` function using the result of `cv2.moments` that computes m_{ij}, μ_{ij} and ν_{ij} moments. The following code, being a continuation of the previous code, computes moments and Hu moment invariants and saves it to a CSV file

```
import math
huMomentsList = []
for idx, contour in enumerate(contours):
    #calculate moments
    moments = cv2.moments(contour)
    print("contour nr ", idx, ", moments:", moments)
    #calculate Hu moments
    huMoments = cv2.HuMoments(moments)
    #scaling Hu moments
    for i in range(0,7):
       if huMoments[i] == 0:
           huMoments[i] = 0.0
       else:
           huMoments[i] = -1* math.copysign(1.0, huMoments[i])
           * math.log10(abs(huMoments[i]))
```

	h0	h1	h2	h3	h4	h5	h6
0	[0.]	[0.]	[0.]	[0.]	[0.]	[0.]	[0.]
1	[0.]	[0.]	[0.]	[0.]	[0.]	[0.]	[0.]
2	[0.79814845]	[26.81188262]	[24.41845927]	[23.47695713]	[47.42470121]	[-36.88289946]	[49.31560298]
3	[0.70538293]	[2.69093786]	[2.58304728]	[4.38886088]	[-7.8800969]	[-5.73709618]	[-8.68443074]
4	[0.79814845]	[26.85279981]	[24.42416307]	[24.02403254]	[48.2512061]	[-37.47078364]	[-49.17406799]
5	[0.74091863]	[2.46926156]	[2.9935646]	[4.60553714]	[-8.84461097]	[-6.23294822]	[8.43585707]
6	[0.79814845]	[0.]	[25.18292176]	[24.10978444]	[-48.98377938]	[0.]	[48.8498556]
7	[0.79814845]	[26.81178664]	[23.9546497]	[23.7333268]	[-48.094335]	[-37.32975677]	[-47.59838247]
8	[0.79814845]	[27.41384663]	[24.16494398]	[23.87036614]	[-48.0824341]	[-37.63872767]	[-48.00203766]
9	[0.79814845]	[28.01652069]	[24.77083858]	[24.68023155]	[49.49318743]	[-38.7606507]	[49.64558229]
10	[0.67523629]	[1.79176439]	[25.59654951]	[25.54615833]	[51.11772537]	[26.92941533]	[52.6216787]
11	[0.76012002]	[2.48727811]	[3.35418212]	[4.80944317]	[-8.89125582]	[-6.05308223]	[0.]
12	[0.]	[0.]	[0.]	[0.]	[0.]	[0.]	[0.]
13	[0.70056534]	[1.91643844]	[3.56724453]	[4.68988083]	[-8.82139728]	[-5.65043331]	[9.75310793]
14	[0.58037321]	[1.39557297]	[3.782052]	[4.72392509]	[-9.11828063]	[-5.50417661]	[9.13697904]
15	[0.79814845]	[29.21941298]	[26.6618193]	[26.11570643]	[52.50449099]	[-40.7895659]	[54.5047043]
16	[0.6857275]	[1.89187407]	[3.40205217]	[4.3795409]	[8.2942191]	[5.35392537]	[8.76151372]
17	[0.79814845]	[28.01652069]	[24.51249657]	[24.61855795]	[49.28468392]	[-38.85433458]	[-49.39952709]
18	[0.79814845]	[0.]	[26.02654629]	[25.700419]	[-51.80239727]	[0.]	[-51.65197979]
19	[0.79814845]	[27.7213525]	[25.73798887]	[25.60301727]	[51.27462302]	[-39.46378692]	[-52.42122381]
20	[0.79814845]	[27.82192699]	[26.37755539]	[25.60589352]	[53.07603682]	[-39.58323299]	[-51.59785794]
21	[0.78997163]	[3.06604495]	[25.77686473]	[25.79253896]	[52.34820172]	[27.85054456]	[-51.58356701]
22	[0.79814845]	[29.02657497]	[26.31685638]	[26.26031951]	[52.54917525]	[-40.7864668]	[54.00352057]
23	[0.78905807]	[3.00941968]	[25.17852668]	[26.13276919]	[52.34125909]	[27.75796314]	[-51.80614589]
24	[0.79814845]	[0.]	[26.69939242]	[25.85411951]	[53.36018816]	[0.]	[-52.13163211]
25	[0.79804425]	[5.66204838]	[6.19735014]	[9.43787949]	[-17.25549431]	[-12.26890368]	[-25.32645678]
26	[0.79814845]	[27.82245498]	[24.73266168]	[24.70911716]	[49.430966]	[-38.63688209]	[-50.60786036]
27	[0.79814845]	[0.]	[29.45039529]	[27.1276657]	[56.57571274]	[0.]	[55.41774276]
28	[0.79814845]	[27.91895058]	[25.97686679]	[25.63567101]	[51.4647807]	[-39.61840774]	[51.94228181]
29	[0.69109948]	[1.88105976]	[2.92548302]	[4.13412645]	[-7.66393119]	[-5.07465633]	[-18.75625795]
30	[0.77060218]	[2.50929185]	[27.41107951]	[27.55199248]	[55.03535016]	[28.84551973]	[-56.07258]
31	[0.60488307]	[1.57827313]	[3.01942336]	[4.14726545]	[-7.81259446]	[-4.94539935]	[7.98182496]
32	[0.79814845]	[29.02657497]	[26.31685638]	[26.26031951]	[52.54917525]	[-40.7864668]	[-54.00352057]
33	[0.69716981]	[1.92795111]	[4.89335164]	[6.67083021]	[13.07703084]	[7.90232684]	[12.46554216]
34	[0.76012002]	[2.48727811]	[3.35418212]	[4.80944317]	[-8.89125582]	[-6.05308223]	[-22.07892886]
35	[0.69109948]	[1.88105976]	[2.92548302]	[4.13412645]	[-7.66393119]	[-5.07465633]	[18.75278852]

Figure 14: Hu moments for contours from Figure 10.

```
16
17      huMomentsList.append({'h0':huMoments[0], 'h1':huMoments[1],
18                            'h2':huMoments[2], 'h3':huMoments[3],
19                            'h4':huMoments[4], 'h5':huMoments[5],
20                            'h6':huMoments[6]})
21
22  df_HuMoments = pd.DataFrame.from_records(huMomentsList)
23  df_HuMoments.to_csv("output/HuMoments.csv")
```

Since Hu moment invariants can have arbitrary values, we adjust their range using log_{10} transform and preserve their sign. It is standard operation, and it can be helpful when we use moments for Mchine Learning algorithms that require data to be properly normalized/standardized to have a similar range. This code generates the file presented in Figure 14.

In the present section, the discussed static moments are used for the image analysis. It is worth noting that the static moments affect the physical properties in combination with structural sums. This is explicitly shown by

the formulas from Part 2.1 and 2.2.

1.3 Classifying

Once we 'segmented' image into granules and identify potentially interesting ones, the next step is to identify them. The most reliable is the 'hand-made' classification by a skilled human expert. On the other hand, this is the slowest and least automatic way of classification. In some important cases where the error should be minimized, like cancer cell detection, the human expert should have the last word in such classification. However, automatic methods can help in such classification and speed up the process.

In this part, we make a short review of Machine Learning (ML) methods that can be applied to the classification of images. We focus mostly on the methods where no expert insight is available (unsupervised learning). We also explain how to use other types of Machine Learning where expert data is present for learning before classification (supervised learning). The basics of ML in the context of mathematical modeling can be learned in our book [17]. Moreover, a more formal yet practical introduction to Machine Learning are the books [19, 11]. This part cannot be treated as an introduction to the vast area of ML. However, you will learn the idea and see an example that can be generalized to any other, more advanced ML method.

Since computers are good only at crunching numbers, all data, before processing, must be converted into numbers or vectors of numbers. The general idea of ML methods is to distinguish separate classes in the data and associate class labels to them. We distinguish two main cases:

- **supervised learning** - when class labels are present in the training data; Then ML algorithm learns on examples to associate labels to the new data on training. This type of algorithms focus on making precise borders between areas with points associated with different class labels (decision regions). In this class we have algorithms like SVM (Support Vector Machines) and various architectures of Artificial Neural Networks.

- **unsupervised learning** - when that data does not contain class labels, the ML algorithm must infer how many classes there are and label them. The classes are typically determined as clusters in the data space. All kinds of clustering algorithms like k-Means or Gaussian Mixture Model represent this class.

Supervised learning can be made with the data that are labeled. If no such data are available for our case, the way to detect differences in data is

to use unsupervised learning. In the later part we will focus on unsupervised learning. Already trained supervised learning algorithms can be treated as a replacement in the code encountered in this part.

Many ML algorithms are implemented in the Scikit Learn library [4]. The work with classifiers is rather standard and consists of two steps:

1. `fit(...)` - learn classifier;
2. `predict(...)` - use a classifier to predict classes; usually used for new data not used in the learning process;

A prominent example of unsupervised ML algorithms is the k-means method. It tries associating data to k clusters, minimizing WSSE (clusters within the sum of square error). The input is the number of clusters, however, it can be determined. In order to understand the idea, we present the following toy example, where we create artificial data with some clusters and then apply the k-Means algorithm. More details can be found in our book [17].

```
import matplotlib.pylab as plt

#preparing data
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=1000, centers=3, n_features=3,
random_state=0)

# plot data
plt.scatter(X[:,0], X[:,1], c=y)
plt.xlabel("$x1$")
plt.ylabel("$x2$")
plt.savefig("output/data-kMeans.png")
```

which produces three clusters in Figure 15.

Three clusters are clearly visible, and we expect to distinguish them. The easiest way to determine the number of clusters is to plot WSSE vs the number of clusters (the scree plot) and identify characteristic elbow. The following code

```
from sklearn.cluster import KMeans
wsse = []
for k in range(1, 20):
    kmeans = KMeans(n_clusters=k, n_init=1)
```

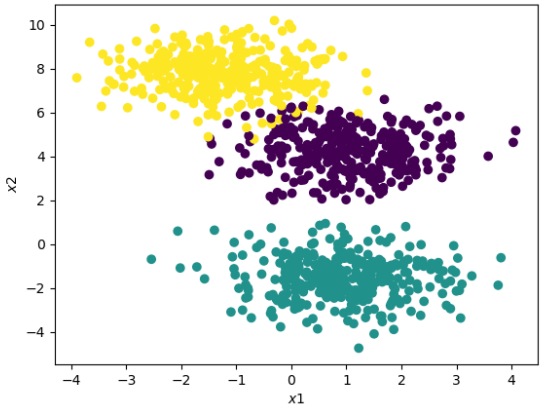


Figure 15: Initial data for k-Means algorithm.

```
    kmeans.fit(X)
    wsse.append(kmeans.inertia_)

#make plot
plt.plot(range(1, 20), wsse)
plt.xlabel("Clusters")
plt.ylabel("WSSE")
plt.savefig("output/kMenas_Scree.png")
```

gives Figure 16. One can see 'elbow' with 'joint' at $k = 3$ clusters. Above this number of clusters, increase does not improve WSSE much.

We can now make k-Means clustering using $k = 3$ clusters

```
kmeans = KMeans(n_clusters=3, n_init=1)
kmeans.fit(X)
##clusters labels
labels = kmeans.predict(X)
##locations of centroids
centroids  = kmeans.cluster_centers_

#plot
plt.scatter(X[:,0], X[:,1], c=labels)
plt.scatter(centroids[:,0],centroids[:,1], c='r', marker='v',
label="centers")
plt.legend()
plt.xlabel("x1")
```

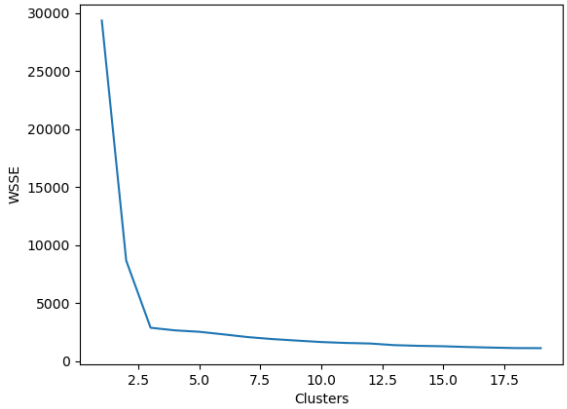


Figure 16: Scree plot.

```
plt.ylabel("x2")
plt.savefig("output/kMeans-clustering.png")
plt.show()
```

that gives plot in Figure 17. The above example was an ideal one. Real-world data are typically not so well separable.

The next example will present the k-means algorithm for image classification, which will be useful in the next part of this part. We use handwritten digits [5] as input data. They are loaded by the following code

```
import numpy as np
import matplotlib.pylab as plt

from sklearn.datasets import load_digits

digits = load_digits()
X = digits.data
Y = digits.target

#reverse color
X = 255-X

#example digit
plt.imshow(X[100].reshape(8,8))
plt.savefig("output/digitSingleOne.png")
plt.show()
```

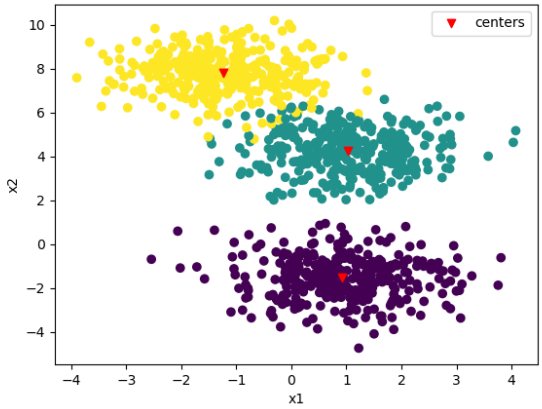


Figure 17: Clustering result.

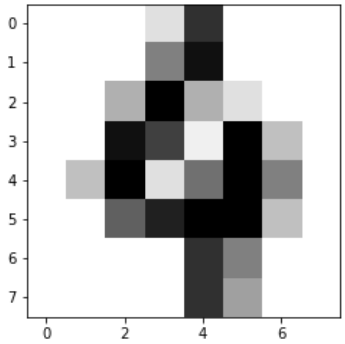


Figure 18: An example of handwritten digit.

that produces Figure 18. The image is encoded as a vector of 64 entries and has to be converted to 8×8 grayscale array.

The next step is to estimate the number of clusters

```
from sklearn.cluster import KMeans
k = range(1,20)
WSSE = []
for i in k:
    model = KMeans(n_clusters = i, n_init='auto')
    model.fit_predict(X)
    WSSE.append(model.inertia_)

```

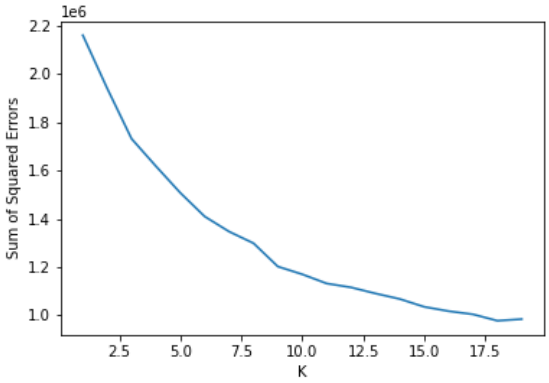


Figure 19: Scree plot for handwritten digits data.

```
plt.plot(k,WSSE)
plt.xlabel('K')
plt.ylabel('Sum of Squared Errors')
plt.savefig("output/digitsScree.png")
plt.show()

```

that gives Figure 19. One can note that the elbow joint is about 10, which is expected. Therefore in the next step we can use the k-Means classifier to cluster digits into similar sets

```
clusters = 10
kmeans = KMeans(n_clusters=clusters, n_init='auto')
kmeans.fit(X)
Z = kmeans.predict(X)

#print selected class
##get images of numbers from given class
clusterIndex = 2
#get images with a specific number of cluster
imageIndex = np.where(Z==clusterIndex)[0]
#plt.figure(figsize=(10,10))
for k in range(0, imageIndex.shape[0]):
    image = X[imageIndex[k],:]
    image = image.reshape(8, 8)
    plt.imshow(image, cmap='gray')
    plt.savefig("output/digits/{}_{}.png".
```

```
17      format(clusterIndex, k))    plt.show()
```

The results assigned to the class 2 can be inspected visually. It is not perfect, however, many images represents the same digit.

Instead of the k-means algorithm, one can use any other unsupervised ML algorithm or trained, supervised ML algorithm.

1.4 Segmentation using Means algorithm

One of the standard techniques of segmentation of an image is related to similar colors of objects we want to extract. If this is the case, we can do clusterization in color space, and then extract objects belonging to the same cluster (similar color). Clusterization can be performed using an arbitrary unsupervised ML algorithm. We present an example that uses the k-means algorithm.

We start by preparing an image

```
1  import numpy as np
2  import pandas as pd
3  import matplotlib.pylab as plt
4  import cv2
5
6  #prepare figure
7  image = np.zeros((512,512,3), np.uint8)
8  image = image + 255
9
10 #perturbation
11 cv2.rectangle(image,(100,0),(500,400),(240,240,240),-1)
12 cv2.rectangle(image,(0,100),(400,500),(0,240,240),-1)
13
14 #set reproducibility of initial image
15 np.random.seed(1)
16
17 #generate class 1
18 x = 512* np.random.rand(200)
19 y = 512* np.random.rand(200)
20 for i,j in zip(x,y):
21     cv2.circle(image,(int(i),int(j)), 5, (100,100,100), -1)
22
23 #generate class 2
24 x = 512* np.random.rand(50)
```

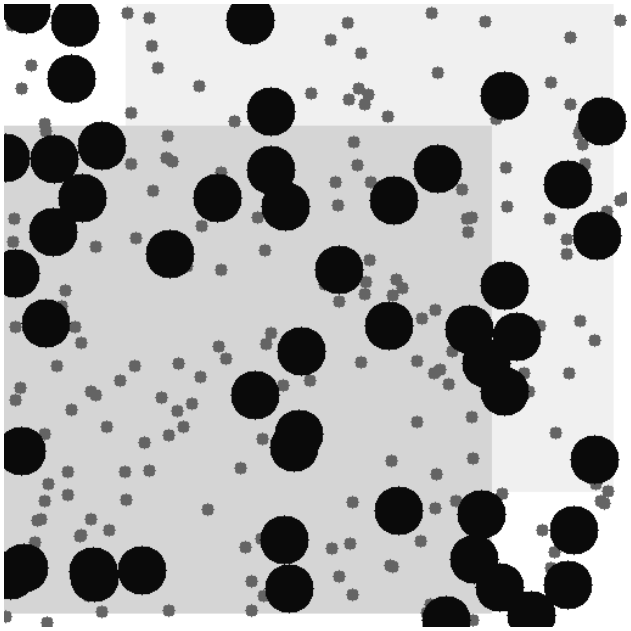


Figure 20: Initial image.

```
y = 512* np.random.rand(50)
for i,j in zip(x,y):
    cv2.circle(image,(int(i),int(j)), 20, (10,10,10), -1)
```

that produces an image from Figure 20.

Next, we localize the optimal number of classes

```
 from sklearn.cluster import KMeans

k = range(1,5)
WSSE = []

for i in k:
    model = KMeans(n_clusters = i, n_init='auto')
    model.fit_predict(df)
    WSSE.append(model.inertia_)

plt.plot(k,WSSE)
plt.xlabel('K')
plt.ylabel('WSSE')
plt.grid(True)
plt.savefig("output/kmeansSegmentationColor.png")
```

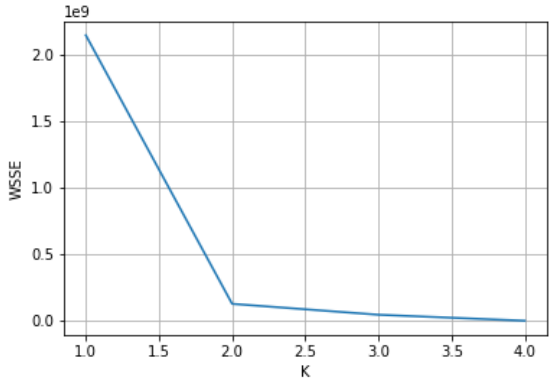


Figure 21: Scree plot for Figure 20.

```
plt.show()
```

that produces Figure 21.

One can observe that 2 clusters are enough. The following code clusters data with respect to 2 clusters

```
NClusters = 2

km = KMeans( n_clusters= NClusters, n_init='auto')
labels = km.fit_predict(df)
#create mask from class labels
mask = labels.reshape(xShape, yShape, 1)

imgLabelled = []
masks = []
for label in range(NClusters):
    maskClass = np.where(mask == label, 255, 0)
    maskClass = maskClass.astype(np.uint8)
    masks.append(maskClass)
    cv2.imwrite("output/SegmentationColorMask{}.png".
    format(label), maskClass)
    imgLabel = cv2.bitwise_and(imageGray, imageGray,
    mask = maskClass)    imgLabelled.append(imgLabel)
    cv2.imwrite("output/SegmentationColorLabel{}.png".
    format(label), imgLabel)
```

that produces segments in Figure 22, which were made using masks Figure 23 applied to Figure 20. We can see that the masks are inverse to each

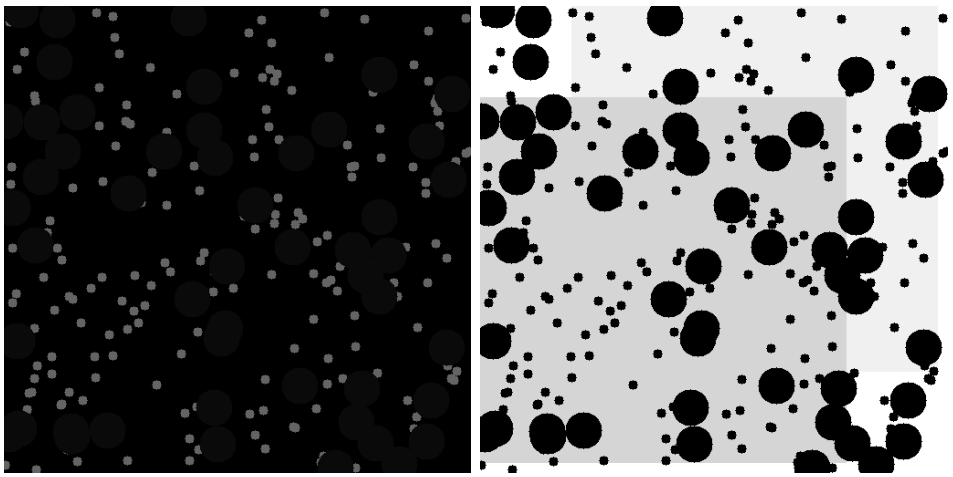


Figure 22: Segmentated image for Figure 20.

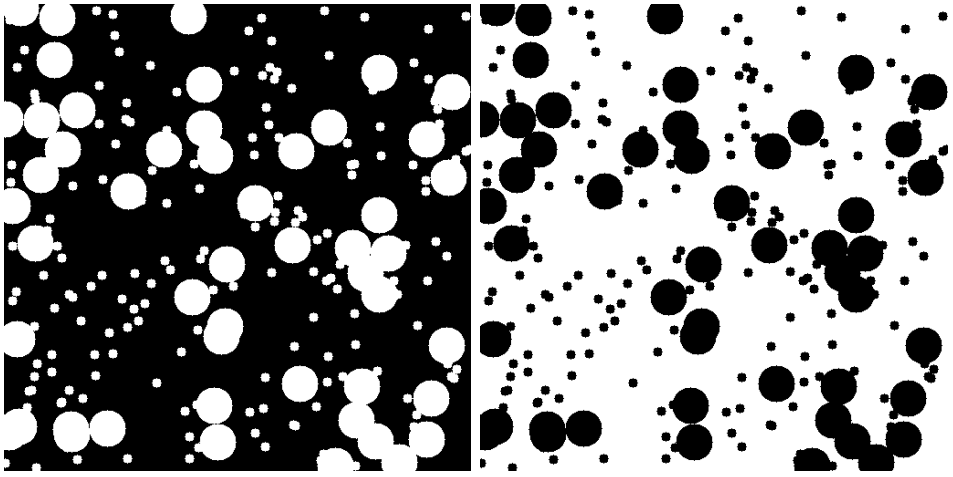


Figure 23: Masks that applied to Figure 20 result in Figure 22.

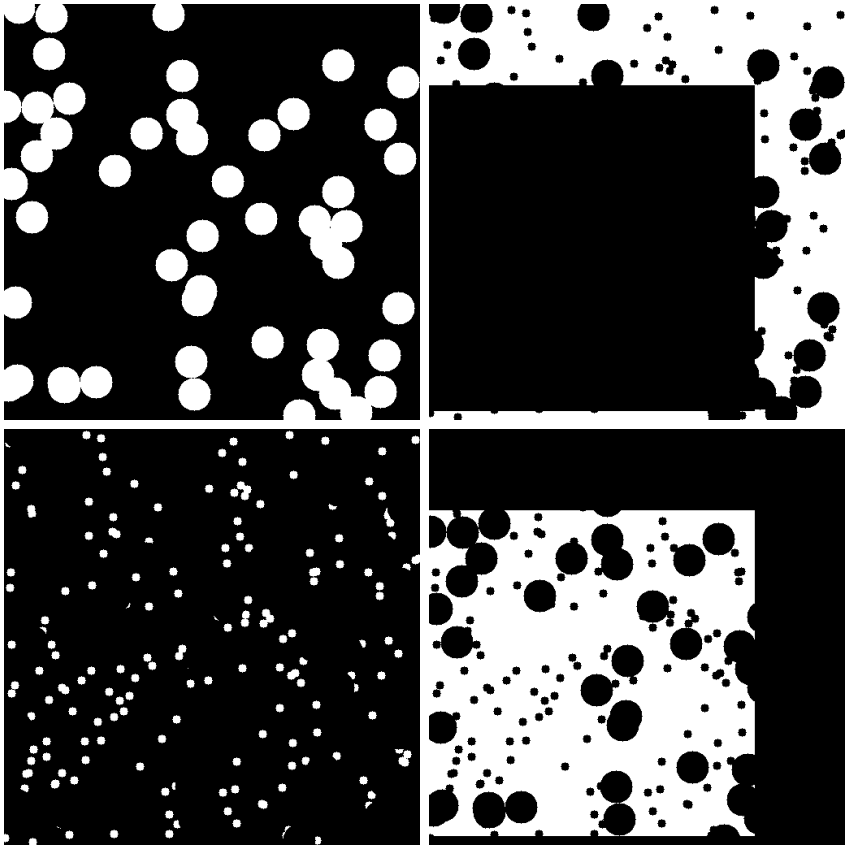


Figure 24: Masks for $k = 4$ clusters.

other.

It is instructive to consider a higher number of clusters. Figure 24 represents masks for `NClusters = 4` clusters in the k-Means algorithm. We observe that the k-Means algorithm separates small and big granules, as well as elements in the shaded region and its complement.

1.5 Summary

Until now, we have presented many essential methods of image manipulation with the OpenCV library that can be used for extracting and classifying granules in an image. They mustn't be used as a black box. They must be used in a specific order, and parameters must be adjusted to a particular situation/type of image. Some experience is needed, however, thanks to them you can automatize granulometry.

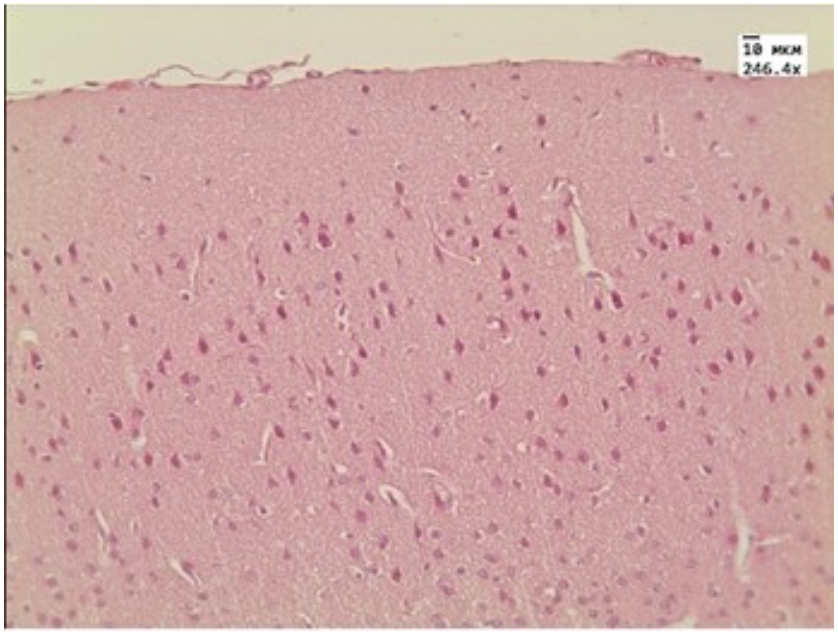


Figure 25: Initial image. Microscope image of cancerous mouse brain biopsy.

2 Examples

In this section, we will describe more advanced examples that use methods from the previous part. We present four examples.

2.1 Example 1

In the first example we want to select darker granules from Figure 25. This is a microscope image of a tissue.

In the first step we read the image and cut out the top part of the image

```
1  import numpy as np
2  import pandas as pd
3  import matplotlib.pylab as plt
4  import cv2
5
6  img1 = cv2.imread("Picture1.png")
7
8  #cut out printout
9  img1 = img1[50:-10,10:]
10 cv2.imwrite("output/Example1Cut.png", img1)
```

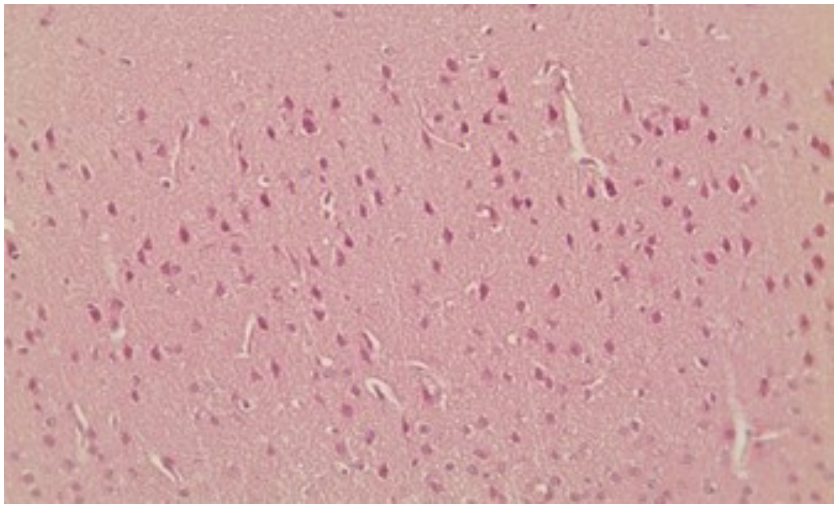


Figure 26: Cropped image that will be further processed.

which gives Figure 26.

We now improve contrast by histogram equalization

```
#histograms of colors
hist= cv2.calcHist([img1],[0],None,[256],[0,256])
hist2=cv2.calcHist([img1],[1],None,[256],[0,256])
hist3=cv2.calcHist([img1],[2],None,[256],[0,256])
##plot histograms
plt.subplot(1, 3, 1)
plt.plot(hist/max(hist))
plt.title('Hist Blue : ')
plt.subplot(1, 3, 2)
plt.plot(hist2/max(hist2),color="green")
plt.title('Hist Green : ')
plt.subplot(1, 3, 3)
plt.plot(hist3/max(hist3),color="red")
plt.title('Hist Red : ')
plt.savefig("output/Example1ColorHistograms")
plt.show()

#histogram equalization
src = cv2.cvtColor(img1, cv2.COLOR_BGR2HSV)
h,s,v = cv2.split(src)
v = cv2.equalizeHist(v)
src = cv2.merge([h,s,v])
```

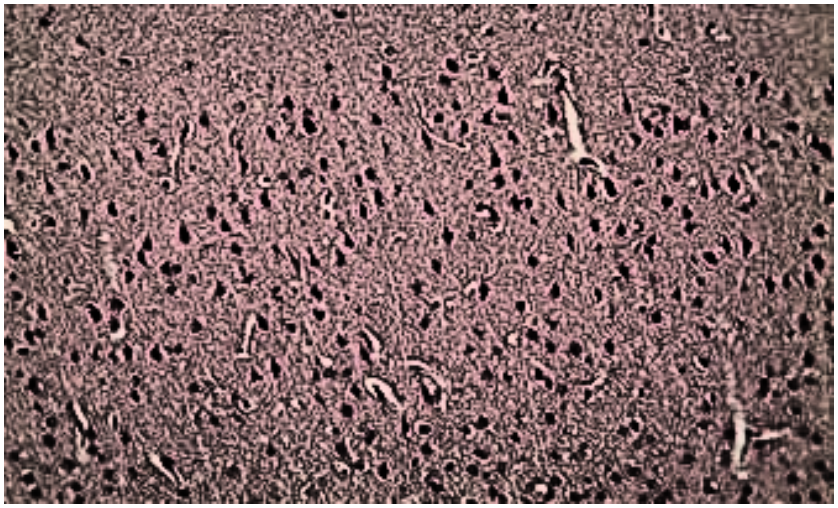


Figure 27: Result of histogram equalization applied to Figure 26.

```
23 img1 = cv2.cvtColor(src, cv2.COLOR_HSV2BGR)
24 cv2.imwrite("output/Example1Equalization.png", img1)
25
26 #histograms of colors after normalization
27 hist= cv2.calcHist([img1],[0],None,[256],[0,256])
28 hist2=cv2.calcHist([img1],[1],None,[256],[0,256])
29 hist3=cv2.calcHist([img1],[2],None,[256],[0,256])
30 plt.subplot(1, 3, 1)
31 plt.plot(hist/max(hist))
32 plt.title('Hist Blue : ')
33 plt.subplot(1, 3, 2)
34 plt.plot(hist2/max(hist2),color="green")
35 plt.title('Hist Green : ')
36 plt.subplot(1, 3, 3)
37 plt.plot(hist3/max(hist3),color="red")
38 plt.title('Hist Red : ')
39 plt.savefig("output/Example1ColorHistogramsEqualization.png")
40 plt.show()
```

that apart of color histograms, produces histogram-equalized Figure 27.

In order to enhance contours, we apply bilateral filtering, convert to grayscale, and compute histogram in order to determine the threshold. The following code

```
1 #scale figure
```

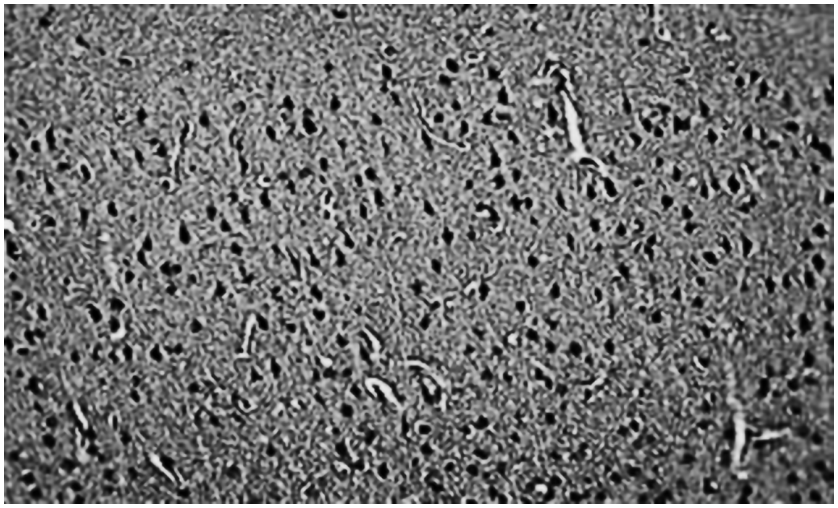


Figure 28: Grayscale image after bilateral filtering to enhance gradients.

```
img1 = cv2.resize(img1,None,fx=4, fy=4, interpolation =
cv2.INTER_CUBIC)
# convert to BW
img1_BW = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
cv2.imwrite("output/Example1BW.png", img1_BW)

#bilateral filtering
img1_BW = cv2.bilateralFilter(img1_BW,9,75,75)
cv2.imwrite("output/Example1BWFiltering.png", img1_BW)

#compute gray histogram
hist1 = cv2.calcHist([img1_BW], [0], None, [256], [0,256])
plt.plot(hist1.ravel()/hist1.max())
plt.savefig("output/Example1HistogramBW.png")
plt.show()
```

which produces Figs. 28, 29. One can note that the black dots are represented by the pixels with intensity below 40. We use this information in thresholding and below. We also do morphological operations to delete small granules in the following code

```
ret1m, thresh1 = cv2.threshold(img1_BW, 40, 256,
cv2.THRESH_BINARY)

#remove small dots
```

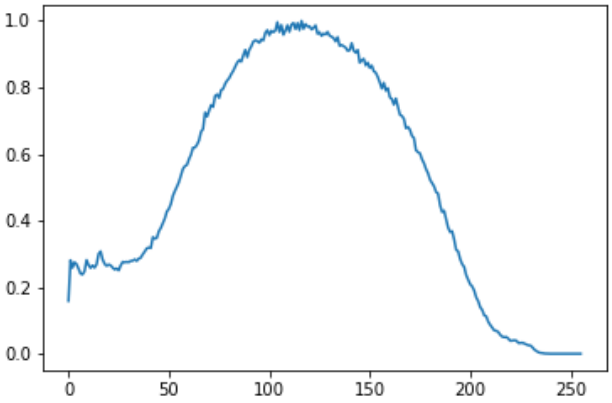


Figure 29: Histogram for Figure 28.

```
thresh1 = cv2.dilate(thresh1, np.ones((5, 5), np.uint8))
thresh1 = cv2.erode(thresh1, np.ones((5, 5), np.uint8))

cv2.imwrite("output/Example1BWThresholding.png", thresh1)
```

that produce Figure 30.

We are ready to detect contours and draw minimum area rectangle

```
#reverse colors in mask
thresh2 = abs(thresh1 - 255)

#find contours
contours, _ = cv2.findContours(thresh2, cv2.RETR_LIST,
cv2.CHAIN_APPROX_TC89_L1 )
blank = np.ones(thresh2.shape)*255
cv2.drawContours(blank, contours, -1, (0, 0, 0), 3)
print("# of contours found: ", len(contours))
cv2.imwrite("output/Example1BWCountour.png", blank)

#minmimal area rectangle
img1_tmp1 = img1.copy()
img1_tmp2 = img1.copy()
for idx, cnt in enumerate(contours):
    minRect = cv2.minAreaRect(cnt)
    box = cv2.boxPoints(minRect)
```



Figure 30: The result of thresholding in Figure 28.

```
    box = np.int0(box)
    cv2.drawContours(img1_tmp1, [box], 0, (0,0,255), 2)
    cv2.drawContours(img1_tmp2, [box], 0, (0,0,255), 2)
    cv2.putText(img1_tmp2, '{}'.format(idx), (int(box[0][0]),
    int(box[0][1]) ), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
    (255, 255, 255), 1, cv2.LINE_AA)
cv2.imwrite("output/Example1BWMinAreaRectangle.png", img1_tmp1)
cv2.imwrite("output/Example1BWMinAreaRectangleWithIndices.png",
img1_tmp2)
```

That produces Figure 31. We can see that most of the granules were detected by this (semi) automatic procedure.

The final step is to fit ellipses to the contours, and save their parameters in CSV file, that is realized by the following code

```
blank = np.ones(thresh2.shape)*255
ellipsesData = []
img_tmp = img1.copy()
for cnt in contours:
    #if less points preventing fitting ellipse
    if cnt.size < 10 :
        continue
    x,y,w,h = cv2.boundingRect(cnt)
    #remove small conoturs
    if w < 2 or h < 2:
```

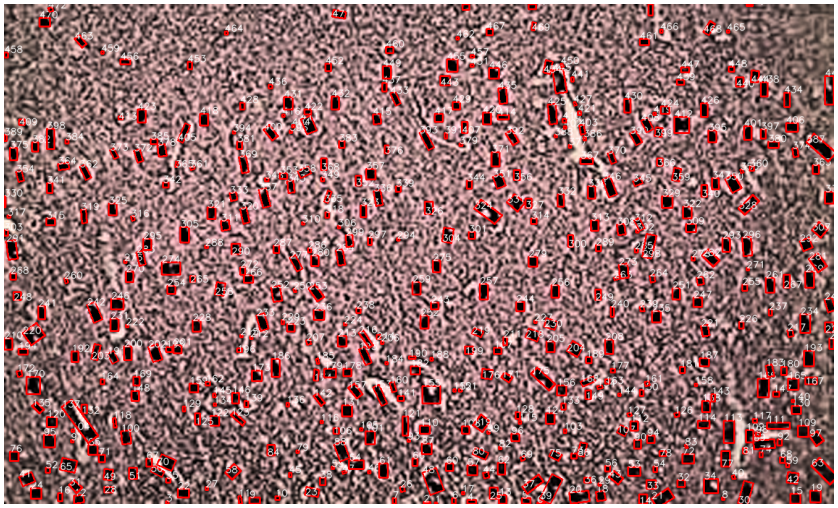


Figure 31: Minimal area rectangles enclosing detected contours with enumeration.

```
        continue
    ellipse = cv2.fitEllipse(cnt)
    cv2.ellipse(blank, ellipse, (0,255, 255), 1, cv2.LINE_AA)
    cv2.ellipse(img_tmp, ellipse, (0,255, 255), 1, cv2.LINE_AA)
    ellipsesData.append({"id": idx, "x":ellipse[0][0],
    "y":ellipse[0][1], "majorAxis": ellipse[1][0], "minorAxis":
    ellipse[1][1], "angle": ellipse[2]})

cv2.imwrite("output/Example1BWEllipses.png", blank)
cv2.imwrite("output/Example1BWEllipsesColor.png", img_tmp)

df_ellipses = pd.DataFrame.from_records(ellipsesData)
df_ellipses.to_csv("output/ellipses.csv")

plt.hist(np.array(df_ellipses.angle)%90)
plt.xlabel("angle[deg]  mod 90")
plt.ylabel("No. of ellipses")
plt.savefig("output/Example1AngleDistibution.png")
plt.show()
```

This produces Figure 32. Moreover, when we are looking for the orientation of ellipses that can represent the orientation of non-spherical granules, we get from the above code Figure 33.

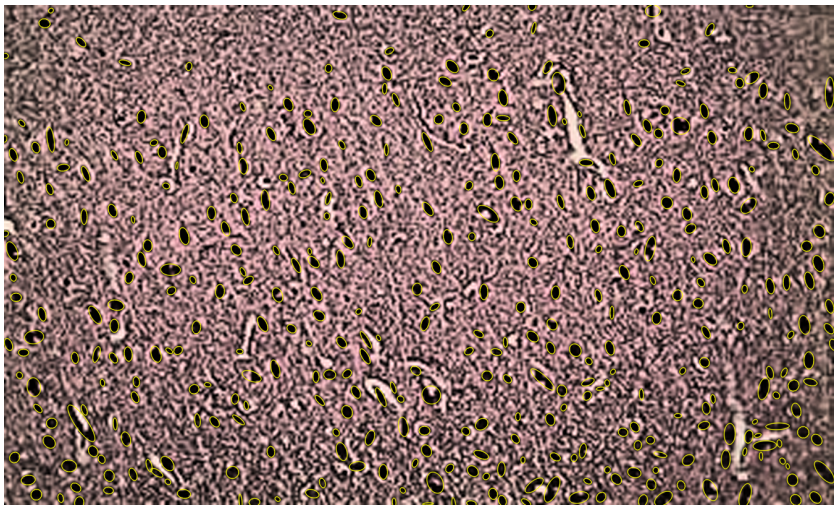


Figure 32: Ellipses fit to the contours.

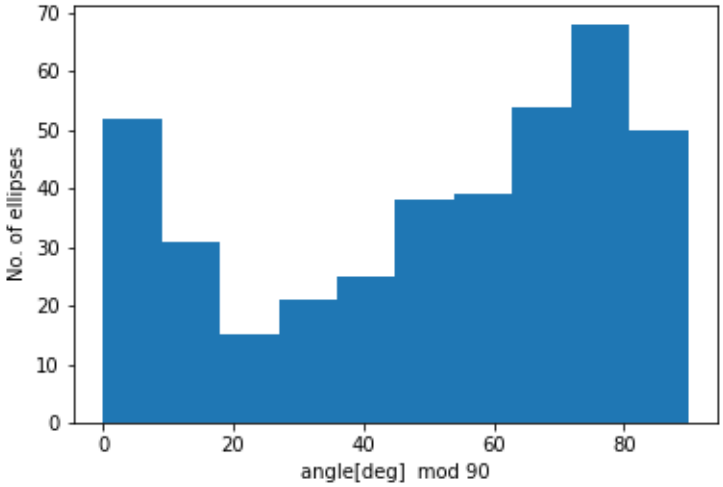


Figure 33: Angle distributions for ellipses.

2.2 Example 2 - classification using images

We now want to classify contours from the previous example using the k-means algorithm. As mentioned above, the classification results must be consulted with an expert. The first part of the code related to preparing images and creating masks is the same as before. We provide it for completeness

```
import numpy as np
import pandas as pd
import matplotlib.pylab as plt
import cv2

img1 = cv2.imread("Picture1.png")

#cut out printout
img1 = img1[50:-10,10:]
cv2.imwrite("output/Example2Cut.png", img1)

#histograms of colors
hist= cv2.calcHist([img1],[0],None,[256],[0,256])
hist2=cv2.calcHist([img1],[1],None,[256],[0,256])
hist3=cv2.calcHist([img1],[2],None,[256],[0,256])
##plot histograms
plt.subplot(1, 3, 1)
plt.plot(hist/max(hist))
plt.title('Hist Blue : ')
plt.subplot(1, 3, 2)
plt.plot(hist2/max(hist2),color="green")
plt.title('Hist Green : ')
plt.subplot(1, 3, 3)
plt.plot(hist3/max(hist3),color="red")
plt.title('Hist Red : ')
plt.savefig("output/Example2ColorHistograms")
plt.show()

#histogram equalization
src = cv2.cvtColor(img1, cv2.COLOR_BGR2HSV)
h,s,v = cv2.split(src)
v = cv2.equalizeHist(v)
src = cv2.merge([h,s,v])
img1 = cv2.cvtColor(src, cv2.COLOR_HSV2BGR)
```

```
35 cv2.imwrite("output/Example2Equalization.png", img1)
36
37 #histograms of colors after normalization
38 hist= cv2.calcHist([img1],[0],None,[256],[0,256])
39 hist2=cv2.calcHist([img1],[1],None,[256],[0,256])
40 hist3=cv2.calcHist([img1],[2],None,[256],[0,256])
41 plt.subplot(1, 3, 1)
42 plt.plot(hist/max(hist))
43 plt.title('Hist Blue : ')
44 plt.subplot(1, 3, 2)
45 plt.plot(hist2/max(hist2),color="green")
46 plt.title('Hist Green : ')
47 plt.subplot(1, 3, 3)
48 plt.plot(hist3/max(hist3),color="red")
49 plt.title('Hist Red : ')
50 plt.savefig("output/Example2ColorHistogramsEqualization.png")
51 plt.show()
52
53 #scale figure
54 img1 = cv2.resize(img1,None,fx=4, fy=4, interpolation =
55 cv2.INTER_CUBIC)
56 # convert to BW
57 img1_BW = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
58 cv2.imwrite("output/Example2BW.png", img1_BW)
59
60 #bilateral filtering
61 img1_BW = cv2.bilateralFilter(img1_BW,9,75,75)
62 cv2.imwrite("output/Example2BWFiltering.png", img1_BW)
63
64 #compute gray histogram
65 hist1 = cv2.calcHist([img1_BW], [0], None, [256], [0,256])
66 plt.plot(hist1.ravel()/hist1.max())
67 plt.savefig("output/Example2HistogramBW.png")
68 plt.show()
69
70 #tresholding - we select 40 for selecting black granules
71 ret1m, thresh1 = cv2.threshold(img1_BW, 40, 256,
72 cv2.THRESH_BINARY)
73
74 #remove small dots
75 thresh1 = cv2.dilate(thresh1, np.ones((5, 5), np.uint8))
```

```
thresh1 = cv2.erode(thresh1, np.ones((5, 5), np.uint8))

cv2.imwrite("output/Example2BWThresholding.png", thresh1)

#reverse colors in mask
thresh2 = abs(thresh1 - 255)

#find contours
contours, _ = cv2.findContours(thresh2, cv2.RETR_LIST,
cv2.CHAIN_APPROX_TC89_L1 )
blank = np.ones(thresh2.shape)*255
cv2.drawContours(blank, contours, -1, (0, 0, 0), 3)
print("# of contours found: ", len(contours))
cv2.imwrite("output/Example2BWCountour.png", blank)

#minmimal area rectangle
img1_tmp1 = img1.copy()
img1_tmp2 = img1.copy()
for idx, cnt in enumerate(contours):
    minRect = cv2.minAreaRect(cnt)
    box = cv2.boxPoints(minRect)
    box = np.int0(box)
    cv2.drawContours(img1_tmp1, [box], 0, (0,0,255), 2)
    cv2.drawContours(img1_tmp2, [box], 0, (0,0,255), 2)
    cv2.putText(img1_tmp2, '{}'.format(idx), (int(box[0][0]),
    int(box[0][1]) ), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
    (255, 255, 255), 1, cv2.LINE_AA)
cv2.imwrite("output/Example2BWMinAreaRectangle.png", img1_tmp1)
cv2.imwrite("output/Example2BWMinAreaRectangleWithIndices.png",
img1_tmp2)
```

We now crop from the initial image as bounding rectangles surrounding contours. These images will be reshaped into vectors to be an input for the ML algorithm. The following code prepares the data

```
image = img1.copy()
img_tmp = img1.copy()
filtered_contours = []
selected_img = []
for idx, contour in enumerate(contours):
    #draw selected contour
```

```
    minRect = cv2.minAreaRect(contour)
    box = cv2.boxPoints(minRect)
    box = np.int0(box)
    cv2.drawContours(img_tmp, [box], -1, (0,0,255), 1)

    #extract image from bounding rectangle
    x, y, w, h = cv2.boundingRect(contour)
    cropped = image[y:y + h, x:x + w, :].copy()
    feature = cv2.resize(cropped, (20, 20))
    cv2.imwrite("output/figs/{}.png".format(idx), feature)

    #scale cropped image to 1D array
    feature = feature.reshape(-1)
    selected_img.append(feature)
cv2.imwrite("output/Exmaple2SelectedContours.png", img_tmp)
df = pd.DataFrame(selected_img)
```

We now use these data to determine optimal number of clusters (similar granules)

```
from sklearn.cluster import KMeans
k = range(1,20)
WSSE = []
for i in k:
    model = KMeans(n_clusters = i, n_init='auto')
    model.fit_predict(df)
    WSSE.append(model.inertia_)
plt.plot(k,WSSE)
plt.xlabel('K')
plt.ylabel('WSSE')
plt.grid(True)
plt.savefig("output/Example2kmeans.png")
plt.show()
```

that produces Figure 34. From the 'elbow curve', we can observe that $k = 4$ clusters are sufficient. We therefore do classification using this value. Moreover, we fit ellipses to the contours and assign them to the class labels. The following lengthy code does the mentioned operations

```
km = KMeans( n_clusters=4, n_init='auto')
df['label'] = km.fit_predict(df)
```

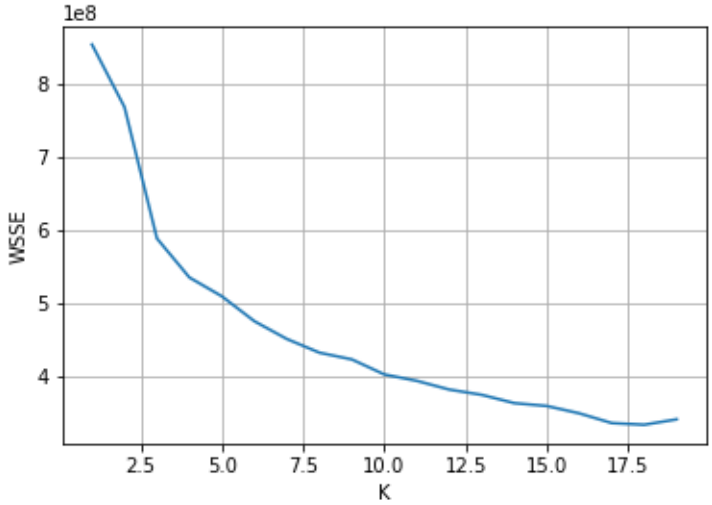


Figure 34: Angle distributions for ellipses.

```
3
4  df.to_csv("output/LabelledContours.csv")
5
6  img_tmp = image.copy()
7  img_ellipse = image.copy()
8  img_ellipse_closed = image.copy()
9  ellipsesData = []
10 for label, df_grouped in df.groupby('label'):
11     Ngranules = len(df_grouped)
12     print(Ngranules, " in label ", label)
13     mask = np.zeros_like(image[:, :, 0])
14     cv2.drawContours(mask, [contours[i] for i in
15     df_grouped.index], -1, (255), -1)
16     masked_image = cv2.bitwise_and(image, image, mask=mask)
17     masked_image[mask==0] = (255,255,255)
18     cv2.imwrite("output/Example2label{}.png".format(label),
19     masked_image)
20
21     print("fitting for label = ", label)
22     for  idx in df_grouped.index:
23         minRect = cv2.minAreaRect(contours[idx])
24         box = cv2.boxPoints(minRect)
25         box = np.int0(box)
```

```
        cv2.drawContours(img_tmp, contours[idx], -1,
        (0,0,label*50), 1)

        #fit ellypses
        if contours[idx].size < 10 :
            continue
        if w < 2 or h < 2:
            continue
        ellipse = cv2.fitEllipse(contours[idx])
        ellipsesData.append({"id": idx, "x":ellipse[0][0],
        "y":ellipse[0][1], "majorAxis": ellipse[1][0],
        "minorAxis": ellipse[1][1], "angle": ellipse[2],
        "class": label})
        if max(ellipse[1]) < 50:
            cv2.ellipse(img_ellipse, ellipse, (0,0,label*50),
            1, cv2.LINE_AA)
            cv2.putText(img_ellipse, '{}_{}'.format(idx, label),
            (int(box[0][0]), int(box[0][1]) ),
            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1,
            cv2.LINE_AA)
            cv2.ellipse(img_ellipse_closed, ellipse,
            (0,0,label*50), -1, cv2.LINE_AA) #open ellipses
            cv2.putText(img_ellipse_closed, '{}_{}'.format(idx,
            label), (int(box[0][0]), int(box[0][1]) ),
            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1,
            cv2.LINE_AA)

df_ellipses = pd.DataFrame.from_records(ellipsesData)
df_ellipses.to_csv("output/ellipses.csv")

cv2.imwrite('output/Example2_color_segmentation.png', img_tmp)
cv2.imwrite('output/Example2_color_ellypses.png', img_ellipse)
cv2.imwrite('output/Example2_color_ellypses_closed.png',
img_ellipse_closed)
```

which produces figures marking classified ellipses, as Figure 35.

2.3 Example 3

In this example, we do a similar analysis to the previous one using Figure 36. After reading and cropping the image by the following code snippet

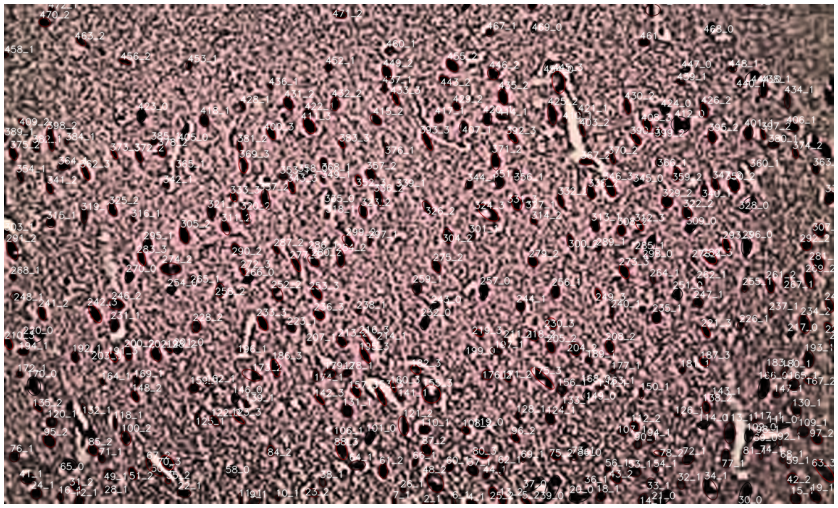


Figure 35: Fitted ellipses with numeration and class in the format: number_label.

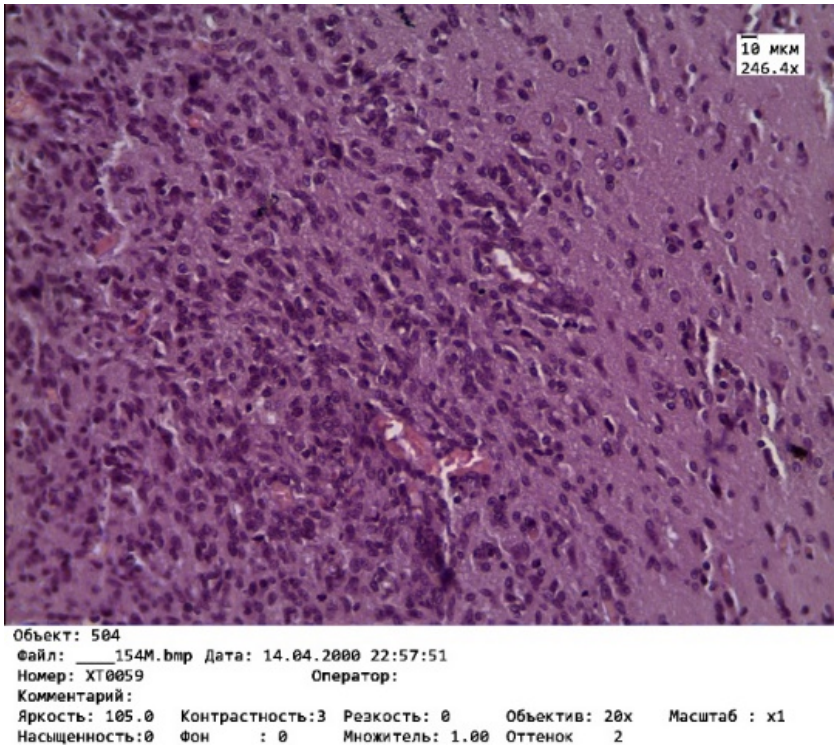


Figure 36: Initial image. Microscope image of cancerous mouse brain biopsy.

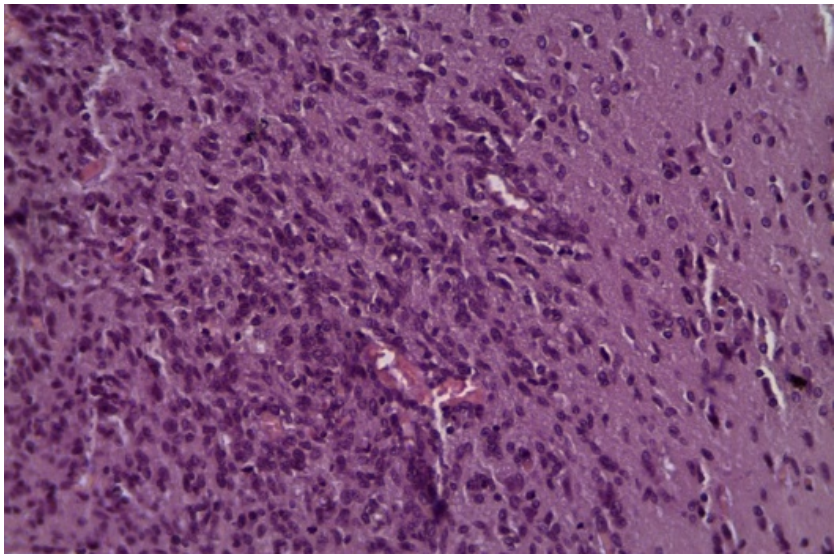


Figure 37: Cropped image.

```
1 img1 = img1[50:-80,10:]
```

that result in Figure 37. The histogram of intensity for a grayscale image, Figure 38 suggests the threshold at about 50. The identified contours are presented in Figure 39.

Finally, on classifying and fitting ellipse we get Figure 40.

2.4 Example 4 - classification using mean color and moments of contours

We use again the image from Figure 36 and the code from the previous example. However, instead of using a full image representing a granule cropped from the original image as data for the classifier, we use mean color (each color channel) and Hu moment invariants. These parameters can be important if we expect that the geometry of granules, as well as their (average) color, are parameters that are important to discriminating different classes.

The change in the previous code that prepares the data is as follows

```
1 import math
2 image = img1.copy()
3 img_tmp = img1.copy()
4 filtered_contours = []
```

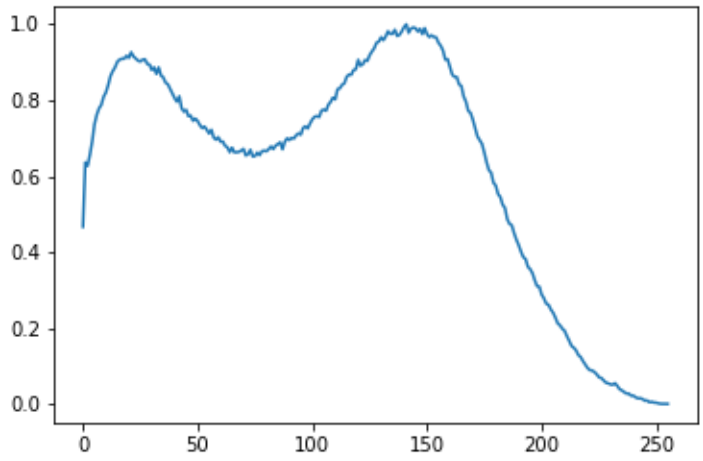


Figure 38: Intensity histogram.

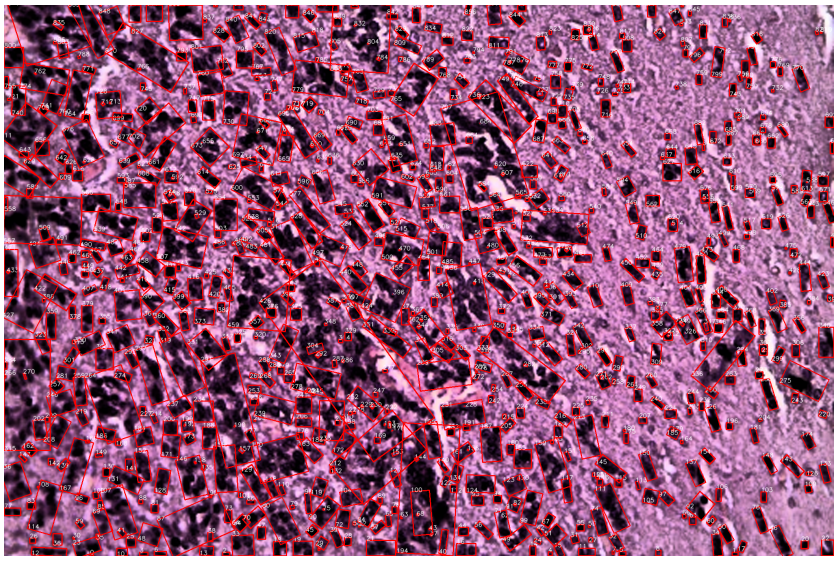


Figure 39: Contours marked by minimal area rectangles and enumerated.

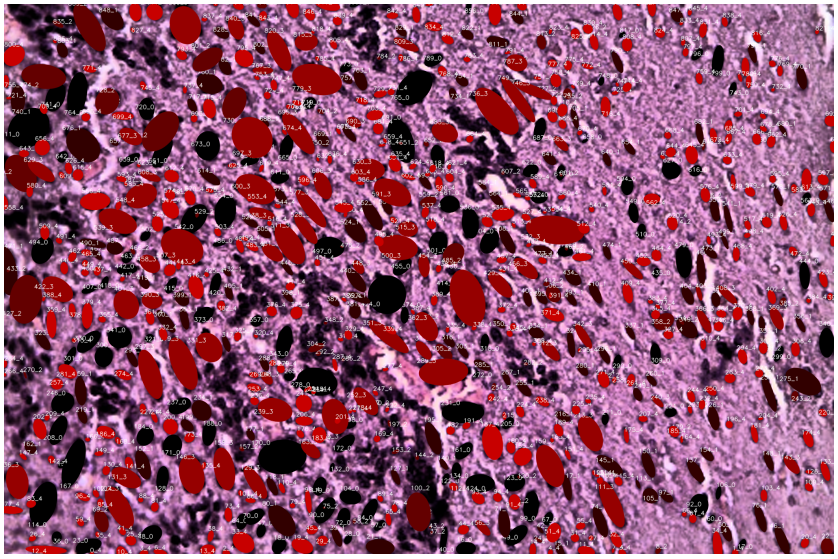


Figure 40: Ellipses fitted to contours with classes marked by different colors and enumeration in the format: number_label.

```
selected_data = []
for idx, contour in enumerate(contours):
    # get mean color of contour:
    masked = np.zeros_like(image[:, :, 0])
    cv2.drawContours(masked, [contour], 0, 255, -1)

    B_mean, G_mean, R_mean, _ = cv2.mean(image, mask=masked)

    #calculate moments
    moments = cv2.moments(contour)
    # Calculate Hu Moments
    huMoments = cv2.HuMoments(moments)

    #scaling Hu moments
    for i in range(0,7):
        #print("hu moment ", i, " ", huMoments[i])
        if huMoments[i] == 0:
            huMoments[i] = 0.0
        else:
            huMoments[i] = -1* math.copysign(1.0, huMoments[i])
            * math.log10(abs(huMoments[i]))

    selected_data.append({'B_mean': B_mean, 'G_mean': G_mean,
```

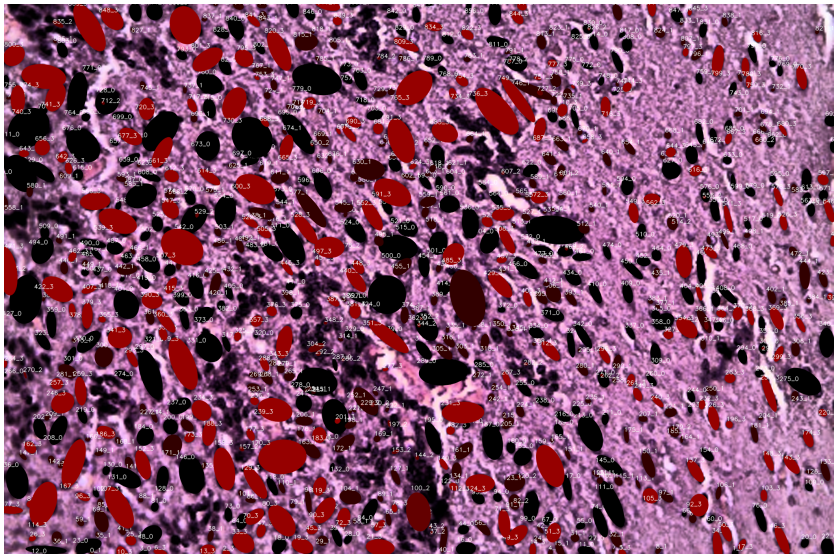


Figure 41: Ellipses fitted to contours with classes marked by different colors and enumeration in the format: number_label.

```
    'R_mean': R_mean, 'h0':huMoments[0], 'h1':huMoments[1],
    'h2':huMoments[2], 'h3':huMoments[3], 'h4':huMoments[4],
    'h5':huMoments[5], 'h6':huMoments[6]})
cv2.imwrite("output/Exmaple2SelectedContours.png", img_tmp)
df = pd.DataFrame(selected_data)
```

For comparison, fitted ellipses are presented in Figure 41.

3 Automatic detection for zones in tissue

The structural sums serve as the cornerstone for mathematical models of random structures within the framework of the *analytical* Representative Volume Element (*a*RVE) theory. We quantified the inter-phase interactions within media and derived analytical formulas for their macroscopic properties based on the Riemann-Hilbert and $\mathbb{R}$-linear problems for analytic functions.

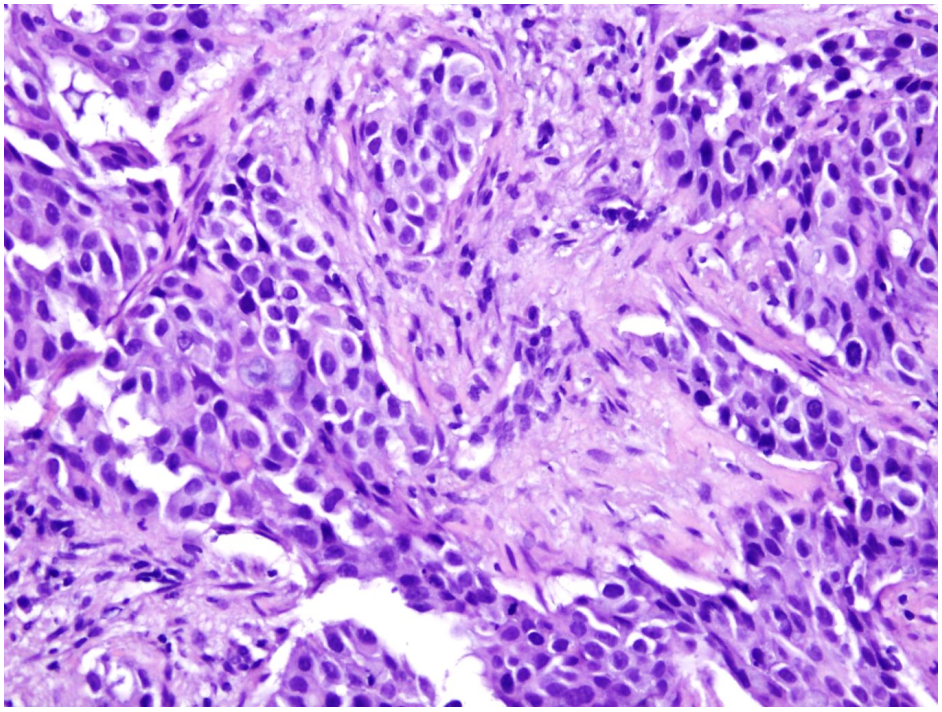


Figure 42: Cancerous tissues. The picture is due to Pablo Losa represented in the COST Action CA23125 workshop in Brcelona 2026.

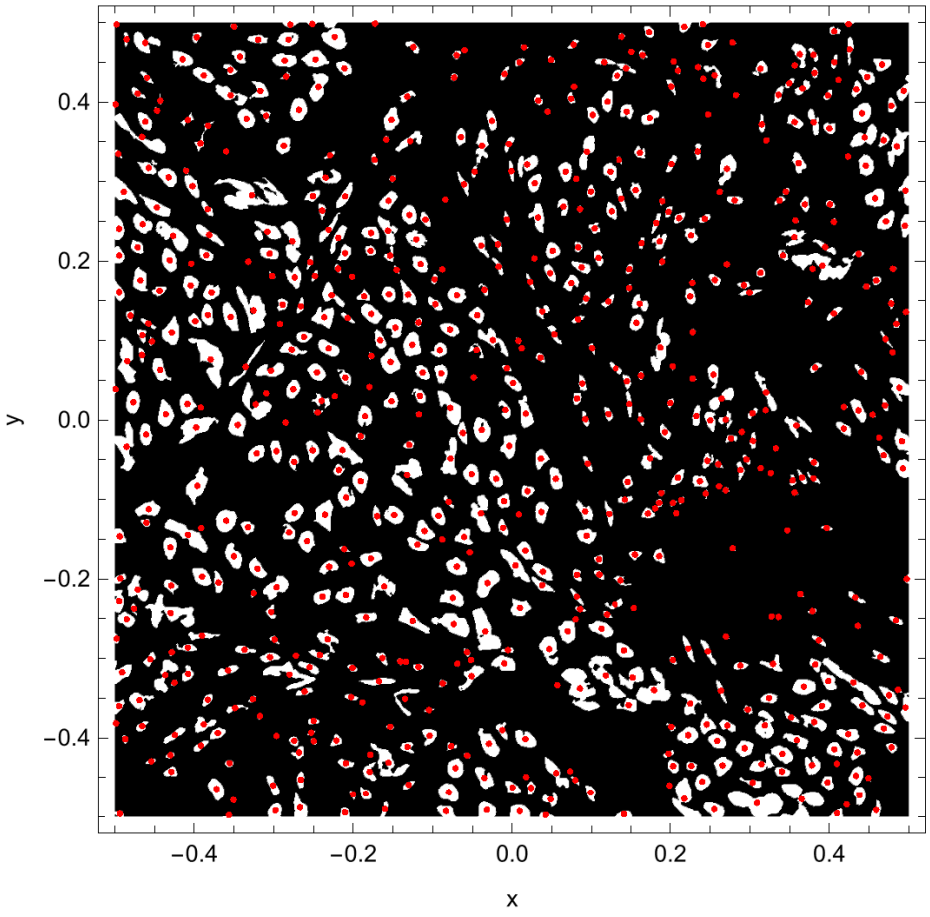


Figure 43: The processing of the central square in Figure 42. White areas and their gravitational centers (red dots).

The main problem of homogenization (global/local) is to replace a heterogeneous medium with an equivalent homogeneous medium and to determine its overall behavior.

Following Parts 2.1 and 2.2 consider a doubly periodic multi-phase composite when the host $G+m_1\boldsymbol{\omega_1}+m_2\boldsymbol{\omega_2}$ and the inclusions $G_k+m_1\boldsymbol{\omega_1}+m_2\boldsymbol{\omega_2}$ are occupied by *dielectric* materials. Let the *permittivity* of the host be normalized to unity, and the permittivity of the kth inclusion be a complex number ε_k. Let u and u_k denote the potentials in G and G_k, respectively. The complex functions u and u_k satisfy Laplace's equation in the corresponding domains and are continuously differentiable in their closures. The perfect contact between the components is expressed by the boundary condition

$$u(t)=u_k(t), \quad \frac{\partial u}{\partial \mathbf{n}}(t)=\varepsilon_k\frac{\partial u_k}{\partial \mathbf{n}}(t), \quad t\in\partial G_k \quad (k=1,2,\dots,N). \tag{3.1}$$

Following Parts 2.1 and 2.2 consider the contrast parameter

$$\varrho_k = \frac{\varepsilon_k - 1}{\varepsilon_k + 1}. \tag{3.2}$$

Introduce the averaged contrast parameter over inclusions

$$\langle \varrho \rangle = \sum_{k=1}^{N} \varrho_k |G_k|.$$

Let a_k be the complex coordinate of the gravity center of G_k; $E_2(z) = \wp(z) + \pi$ the Eisenstein function; $E_{mk} := E_2(a_m - a_k)$ for $m \neq k$. Introduce

$$\mathcal{L} = \frac{1}{\pi} \sum_{k,m=1}^{N} \varrho_k \varrho_m |G_k||G_m| E_{mk}. \tag{3.3}$$

Introduce the shape cumulative parameter

$$\mathcal{J} = \sum_{k=1}^{N} \varrho_k^2 J_k, \quad J_k = \frac{1}{4\pi} \int_{L_k} \mathrm{d}\bar{t} \int_{L_k} \frac{\overline{\tau}}{\tau - t}\, \mathrm{d}\tau. \tag{3.4}$$

The above sums can be written in dimensionless form using the total fraction f of inclusions

$$\mathcal{J}_0 = f^{-1}\mathcal{J}, \quad \mathcal{L}_0 = f^{-2}\mathcal{L}.$$

The permittivity tensor can be calculated by the formula

$$\begin{array}{l} \boldsymbol{\varepsilon}_\perp = (1 + 2\langle \varrho \rangle) I + 2 \sum_{k=1}^{N} \varrho_k^2 \begin{pmatrix} \mathrm{Re}\ \mathcal{J}_k & -\mathrm{Im}\ \mathcal{J}_k \\ -\mathrm{Im}\ \mathcal{J}_k & -\mathrm{Re}\ \mathcal{J}_k \end{pmatrix} \\ + \frac{2}{\pi} \sum_{k,m=1}^{N} \varrho_k \varrho_m |G_k||G_m| \begin{pmatrix} \mathrm{Re}\ E_{mk} & -\mathrm{Im}\ E_{mk} \\ -\mathrm{Im}\ E_{mk} & 2 - \mathrm{Re}\ E_{mk} \end{pmatrix} + O(\varrho_0^3 f^3), \end{array} \tag{3.5}$$

where $\varrho_0 = \max_k |\varrho_k|$.

We us the *structural sums*

$$\begin{array}{l} e_{m_1,\ldots,m_q} = \frac{1}{N^{1+\frac{1}{2}(m_1+\cdots+m_q)}} \times \\ \sum_{k_0,\ldots,k_q} E_{m_1}(a_{k_0} - a_{k_1}) \mathbf{C} E_{m_2}(a_{k_1} - a_{k_2}) \ldots \mathbf{C}^{q+1} E_{m_q}(a_{k_{q-1}} - a_{k_q}), \end{array} \tag{3.6}$$

where $E_p(z)$ are the Eisenstein functions, $\mathbf{C}$ the complex conjugation. For instance, e_2 and e_{22} take the following form

$$\begin{aligned} e_2 &= \frac{1}{N^2} \sum_{k_0=1}^{N} \sum_{k_1=1}^{N} E_2(a_{k_0} - a_{k_1}), \\ e_{22} &= \frac{1}{N^3} \sum_{k_0=1}^{N} \sum_{k_1=1}^{N} \sum_{k_2=1}^{N} E_2(a_{k_0} - a_{k_1}) \overline{E_2(a_{k_1} - a_{k_2})}. \end{aligned} \tag{3.7}$$

The structural sum (3.6) is a discrete multiple convolution of the Eisenstein functions. A fast, almost linear algorithm in N for its computation was developed in [6, 7].

The theory of aRVE can be considered as a constructive application of the Decomposition Theorem. One of aRVE applications is outlined below. Let us have at our disposal two plane pictures of dispersed composites and their digital treatment in the form of two sets of the centers of inclusions $\mathbf{a} = \{a_1, a_2, \ldots, a_N\}$ and $\mathbf{a}' = \{a'_1, a'_2, \ldots, a'_{N'}\}$ scaled to the periodic unit cell. We want to know whether these two composites belong to the same class of random composites.

First, we must calculate all the structural sums $e_{m_1,\ldots,m_s}$ for two sets of points $\mathbf{a}$ and $\mathbf{a}'$. The coefficient A_k on f^{k+1} is a linear combination of the structural sums $e_{m_1,\ldots,m_s}$ with $m_1 + \ldots + m_s = 2k$. Therefore, we have to compare the set of complex values $\mathcal{E} = \{\mathcal{E}_1, \mathcal{E}_2, \ldots\}$ where

$$\mathcal{E}_1 = \{e_2\},\ \mathcal{E}_2 = \{e_{22}\},\ \mathcal{E}_3 = \{e_{33}, e_{222}\},\ \mathcal{E}_4 = \{e_{44}, e_{332}, e_{233}, e_{2222}\}, \ldots. \tag{3.8}$$

constructed for $\mathbf{a}$ and the set $\mathcal{E}'$ for $\mathbf{a}'$.

The set $\mathcal{E}$ is infinite and may be truncated by the concentration tolerance. Let it be equal to $O(f^8)$ for definiteness. Then, we have to calculate the finite set of structural sums $e_{m_1,\ldots,m_s}$

$$\mathcal{E}_1 = \{e_2\},\ \mathcal{E}_2 = \{e_{22}\}, \ldots, \mathcal{E}_7 = \{e_{77}, e_{266}, \ldots, e_{2222222}\}. \tag{3.9}$$

It is worth noting that the 2-point correlation functions yield the same result as only one constant e_2; the 3-point correlation functions yield the the same result as the infinitely many elements e_{pp} $(p = 2, 3, \ldots)$; the 4-point correlation functions yield the same result as the infinitely many triple elements e_{m_1,m_2,m_3} from $\mathcal{E}$ and so forth. Using aRVE is equivalent to the usage of the multi-point correlation functions.

The structural sums for the random i.i.d. simulations

$$e'_2 = 3.14,\ e'_{22} = 11.5249,\ e'_{33} = -23.3673, e'_{44} = 7.9885. \tag{3.10}$$

The structural sums for the considered structure has the form

$$e_2 = 3.3876 - 0.3697i,\ e_{22} = 41.335,\ e_{33} = -338.484, e_{44} = 6541.22. \tag{3.11}$$

4 Conclusion and practical application

The theory of an aRVE bridges the gap between classical homogenization theory and the incorporation of randomness in heterogeneous media. Its

methodological and practical application enables a systematic treatment of the variability in random composites within the framework of homogenization principles and asymptotic analysis.

Application to the diagnosis and classification of glioma structures requires a set of corresponding pictures to apply the developed machine learning classification algorithm.

The following strategy of cancer detection can be applied by by the developed algorithms to obtain practical results:

- images of cancer samples with the precise description of zones with their permittivity data;
- processing the pictures by the presented algorithm;
- application of math formulas to permittivity and structural sums for a classification.

In order to perform the above strategy step we need:

- physical constants (perhaps as functions of frequency) of the image components (2 or multi-phase). It must be precisely described (by color) or by direct indication, which domains have the given constant.
- to present at least 10 pictures of healthy and 10 pictures of diseased samples. 10 + 10 pictures should present one type of healthy - diseased samples to process their and use in the next classification of new pictures by supervised machine learning.

References

[1] Example code from this book:. `https://github.com/rkycia/AnisotropyOfMetamaterialsBook`. Accessed: 27.09.2024.

[2] Opencv library. `https://opencv.org/`. Accessed: 29.07.2024.

[3] Scientific python lectures. `https://lectures.scientific-python.org/`. Accessed: 29.07.2024.

[4] Scikit learn library. `https://scikit-learn.org`. Accessed: 29.07.2024.

[5] Uci ml hand-written digits datasets. `https://archive.ics.uci.edu/dataset/80/optical+recognition+of+handwritten+digits`. Accessed: 29.07.2024.

[6] R. Czapla, W. Nawalaniec, and V. Mityushev. Effective conductivity of random two-dimensional composites with circular non-overlapping inclusions. *Computational Materials Science*, 63:118–126, 2012.

[7] P. Drygaś, S. Gluzman, V. Mityushev, and W. Nawalaniec. *Applied analysis of composite media: analytical and computational results for materials scientists and engineers*. Woodhead Publishing, 2019.

[8] R.O. Duda and P.E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[9] J. Flusser. On the independence of rotation moment invariants. *Pattern recognition*, 33(9):1405–1410, 2000.

[10] J. Flusser and T. Suk. Rotation moment invariants for recognition of symmetric objects. *IEEE Transactions on Image Processing*, 15(12):3784–3790, 2006.

[11] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

[12] R.C. Gonzalez. *Digital image processing*. Pearson education india, 2009.

[13] Joseph Howse and Joe Minichino. *Learning OpenCV 4 Computer Vision with Python 3: Get to grips with tools, techniques, and algorithms for computer vision and machine learning*. Packt Publishing Ltd, 2020.

[14] Ming-Kuei Hu. Visual pattern recognition by moment invariants. *IRE transactions on information theory*, 8(2):179–187, 1962.

[15] A. Kaehler and G. Bradski. *Learning OpenCV 3: computer vision in C++ with the OpenCV library*. " O'Reilly Media, Inc.", 2016.

[16] R. Klette. *Concise computer vision*, volume 233. Springer, 2014.

[17] V. Mityushev, Kycia R., W. Nawalaniec, and N. Rylko. *Introduction to Mathematical Modeling and Computer Simulations*. Francis & Taylor, 2nd edition, 2025.

[18] N. Otsu et al. A threshold selection method from gray-level histograms. *Automatica*, 11(285-296):23–27, 1975.

[19] S. Raschka, Yuxi Hayden Liu, and V. Mirjalili. *Machine Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python*. Packt Publishing Ltd, 2022.

[20] C.-H. Teh and R.T. Chin. On the detection of dominant points on digital curves. *IEEE Transactions on pattern analysis and machine intelligence*, 11(8):859–872, 1989.

[21] A.F. Villán. *Mastering OpenCV 4 with Python: a practical guide covering topics from image processing, augmented reality to deep learning with OpenCV 4 and Python 3.7*. Packt Publishing Ltd, 2019.