Université de Lille

BINARY SEGMENTATION

RECONNAISSANCE DES FORMES

Iacob Sergiu

13th February 2020

Contents

CHAPTER 1

INTRODUCTION

1.1 CONTEXT

This report is the result of a university assignment. It aims to prove that the student understood the motivation, goals and means of studying pattern recognition. Therefore, this is a way of summarizing a series of observations and experiments done on images containing shapes.

1.2 MOTIVATION

Shapes are everywhere and of many kinds. Bigger, smaller, rotated, symmetric or not and so on. But what about their contours? How would we know how exactly to draw such a shape? This is the motivation behind trying to code and decode a *shape's contour*, so one may easily reconstruct such a shape or, often times very important, *simplify* them.

1.3 GOALS

Suppose we have multiple shapes $S_1, \ldots, S_n$, each shape S_i described by the *outer points* of its contour. We need to come up with ways to code these points and also simplify them, that is to reduce the number of points. That would allow us to reduce the memory needed to serialise these shapes, but at the cost of precision. Therefore, if we can somehow *adjust* the trade-off between simplicity and precision, we would have a rather versatile method to work with shape contours that could fit our needs, based on the context we find ourselves in.

Chapter 2

Experiments

2.1 Loading data

For our following experiments, we will use the following textures:

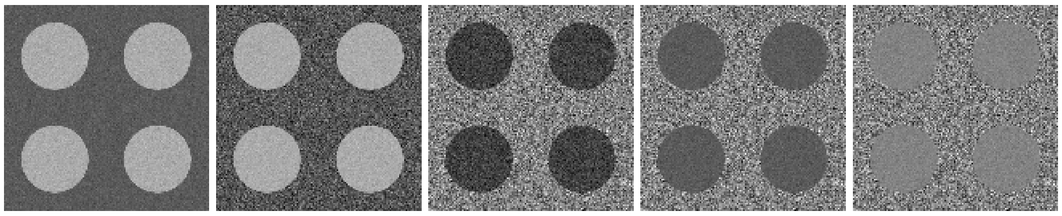

Figure 2.1
Images to experiment on

Our reference image (having a clear difference between objects and background) is the following:

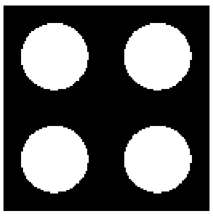

Figure 2.2
Reference image

These will be loaded and converted to gray images:

```r
rdfReadGreyImage <- function (nom) {
    image <- readImage (nom)
    if (length (dim (image)) == 2) {
        image
    } else {
        channel (image, 'red')
    }
}
```

Listing 2.1
Loading images

2.2 GRAY LEVELS

2.2.1 GRAY VALUES HISTOGRAMS

A first good step would be taking a look at how the gray values are distributed. We can use a *histogram* for this.

```
buildHistogram <- function (nom, nbins){
    image <- rdfReadGreyImage (nom)
    h <- hist (as.vector (image), breaks = seq (0, 1, 1 / nbins), main = nom)
}
```

LISTING 2.2
Creating histograms of gray values

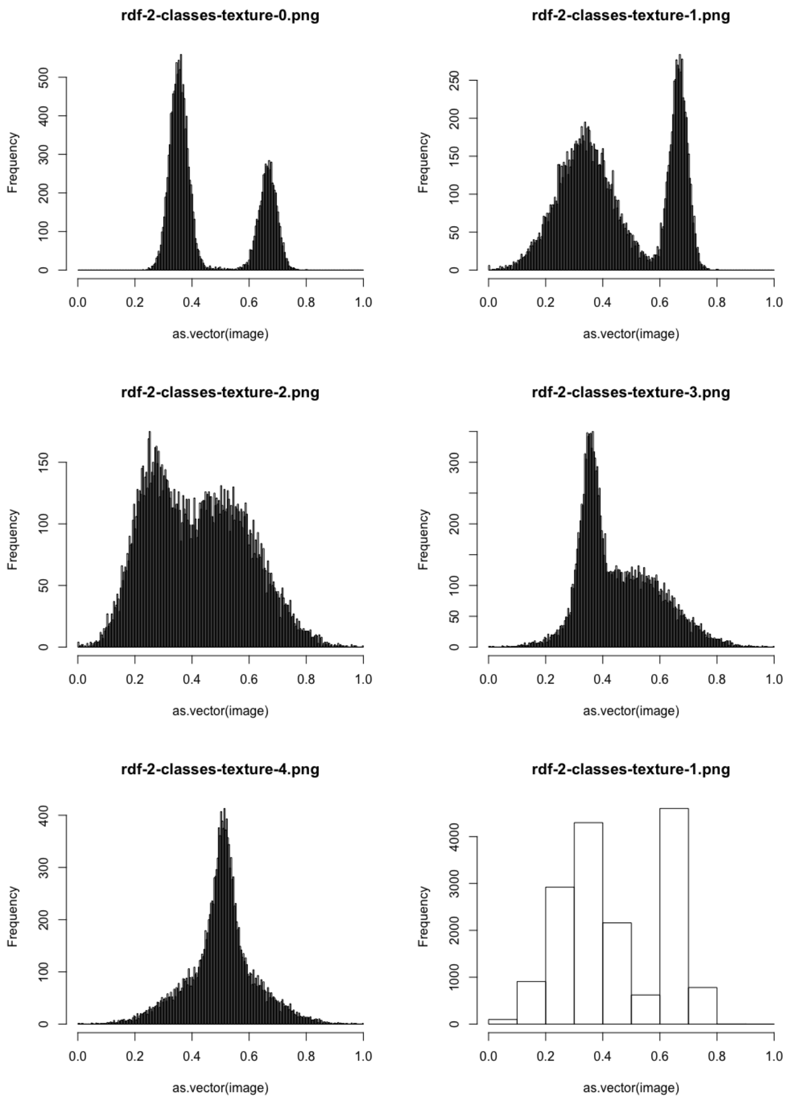

FIGURE 2.3
Histograms of gray values

Since each pixel can have a gray value from 0 to 255, we therefore have a total number of 256 values, so that is going to be our number of *bins* for the histogram. If we were to use, for example, only 10 bins, that would give the number of pixels with gray values between $[0, 0.1)$, $[0.1, 0.2)$ and so on, as it can be seen in the last image 2.3.

2.2.2 THRESHOLDING BASED ON GRAY VALUES

Based on the histograms above 2.3, we may have our first attempt at a simple form of *image segmentation*. By picking the gray value at which (as explained in the figure below), we can classify the pixels on the left as being part of objects, and the others on the right as being part of the background.

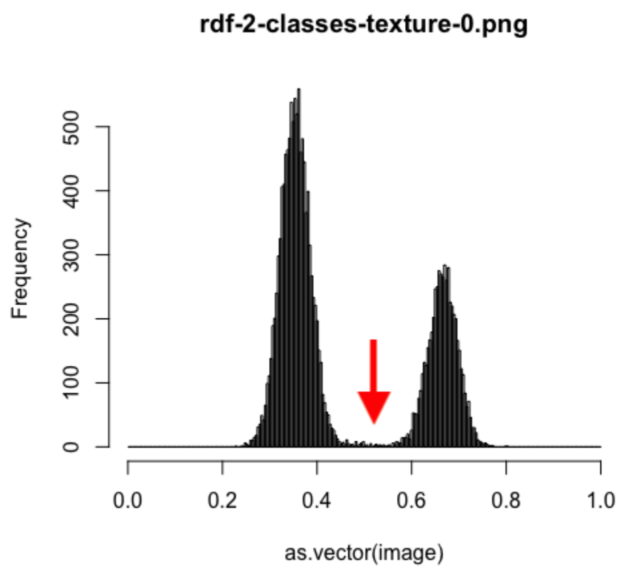

FIGURE 2.4
Choosing the threshold

```
1   buildBinaryImage <- function(nom, threshold){
2       image <- rdfReadGreyImage (nom)
3       binaire <- (image - threshold) >= 0
4       # a "trick" to make the error calculation more precise
5       bin_1 = sum(binaire)
6       bin_0 = dim(binaire)[1] * dim(binaire)[2] - bin_1
7       if (bin_0 < bin_1)
8           binaire = 1 - binaire
9
10      # reference image
11      reference <- rdfReadGreyImage ("rdf-masque-ronds.png")
12      # results
13      error = sum(binaire != reference) / (dim(reference)[1] * dim(reference)[2])
14      error = round (error, 4) * 100
15      info <- sprintf("%s\nthreshold=%s\nerror=%s%%", nom, threshold, error)
16      if (interactive()){
17          display (binaire, "image binaire", method="raster", all=TRUE)
18          legend(x=0.5, y=dim(binaire)[2]/2, info, bg = "lightgreen", box.col = "
    lightgreen", yjust=0.5)
19      }
20  }
```

LISTING 2.3
Segmenting based on the gray threshold

Observation: in the code above, we consider that the majority of pixels are for objects. Therefore, if in our segmentation it's the opposite way, we reverse the image (lines 5-8) so we have a more precise error [1] calculation.

[1]The error represents the percent of missclassified pixels, relative to the reference image

Taking a look at the results, we realise it was not incorrect to call this a *simple* form of segmentation. It's rather effective to reach about the same results as our reference image 2.2 for the first 2 textures, having a low error. However, for the last 2, we don't have a clear "valley" 2.4 in the histogram that we can use as our threshold. For example, the histogram of the last texture resembles a Gaussian distribution, therefore the gray values are evenly distributed relative to their center: $\mu \approx 0.5$.

As a result, we receive an error of about 50% for the last texture when trying to somewhat resemble the reference image. Our lowest error will be achieved by setting the threshold to either 0 or 1, but that will not help us at all.

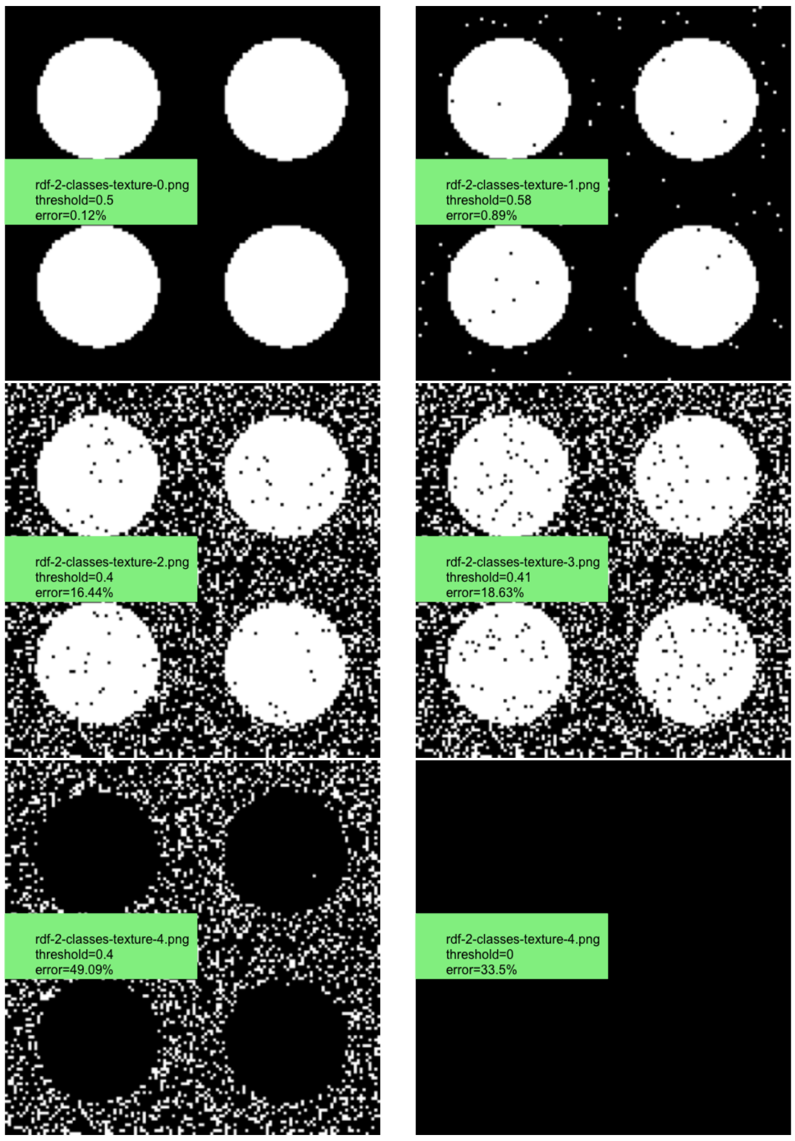
rdf-2-classes-texture-0.png
threshold=0.5
error=0.12%
rdf-2-classes-texture-1.png
threshold=0.58
error=0.89%
rdf-2-classes-texture-2.png
threshold=0.4
error=16.44%
rdf-2-classes-texture-3.png
threshold=0.41
error=18.63%
rdf-2-classes-texture-4.png
threshold=0.4
error=49.09%
rdf-2-classes-texture-4.png
threshold=0
error=33.5%

FIGURE 2.5
Image segmentation based on gray threshold

We can conclude this experiment by saying that this method will not suffice, especially for "noisy" textures, as seen above. Even for the simple textures we still have some missclassified pixels, as it's hard (or often times impossible) to get the right threshold for a perfect segmentation. It is intuitive that the single white pixels (for example for the second texture) surrounded by black pixels should also be part of the background, so next we'll focus on trying to use a pixel's neighbours to gain more information.

2.3 TEXTURE LEVELS

2.3.1 CALCULATING TEXTURE LEVELS

We can define a pixel's *texture level* as being the standard deviation of the gray levels for the pixels located in a square neighborhood. Intuitively, a big value for the standard deviation will mean that the nearby pixels have very different gray values. We could therefore say that the current pixel is a *border* between an object and the background.

To calculate this, we first select the *neighbourhood size* we want to work with. That will represent the number of nearby pixels we'll take into consideration when calculating the standard deviation. A neighbourhood size of 4 will give us a square of $5x5$, since the square will be centered in our original pixel. From there on, we'll just "sweep a neighbourhood window" over each pixel to calculate the average gray level. This can be done with the help of the *filter2()* function from R, which uses fast 2D FFT convolution product. RDocumentation 2020

```r
    # Moyennage d'une image
    rdfMoyenneImage <- function (image, taille) {
        # cote du masque = 2 *taille + 1
        taille <- 2* taille + 1
        masque <- array (taille ^ -2, c (taille, taille))
        # image filtree
        filter2 (image, masque)
    }
```

LISTING 2.4
Calculating a pixel's average value

Taking a quick look at the results of the function *rdfMoyenneImage*, with a neighbourhood size of 5, we can say that it applies a "blur" over the image. That helps reducing the noise.

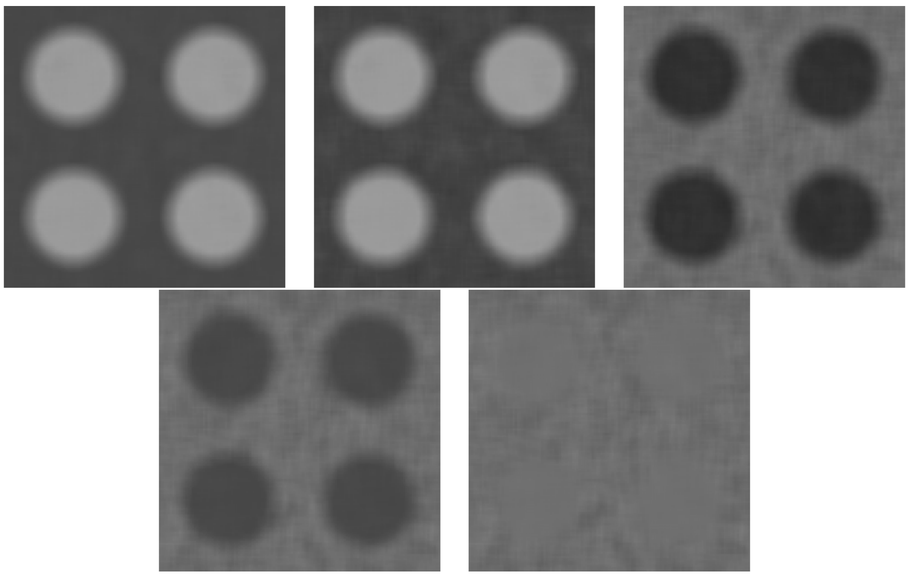

FIGURE 2.6
Result of filter2() function

Afterwards, we just follow the formula for the standard deviation for each pixel. Observation: the image is normalised to have values between 0 and 1, since we're working with gray values.

```r
# Ecart type normalise des voisinages carres d'une image
rdfTextureEcartType <- function (image, taille) {
    # carre de l'image moins sa moyenne
    carre = (image - rdfMoyenneImage (image, taille)) ^ 2
    # ecart type
    ecart = sqrt (rdfMoyenneImage (carre, taille))
    # normalise pour maximum a 1
    ecart / max (ecart)
}
```

LISTING 2.5
Calculating a pixel's average value

2.3.2 HISTOGRAMS OF TEXTURE LEVELS

Applying the methods above (neighbourhood size = 4), we can build the following histograms of texture levels:

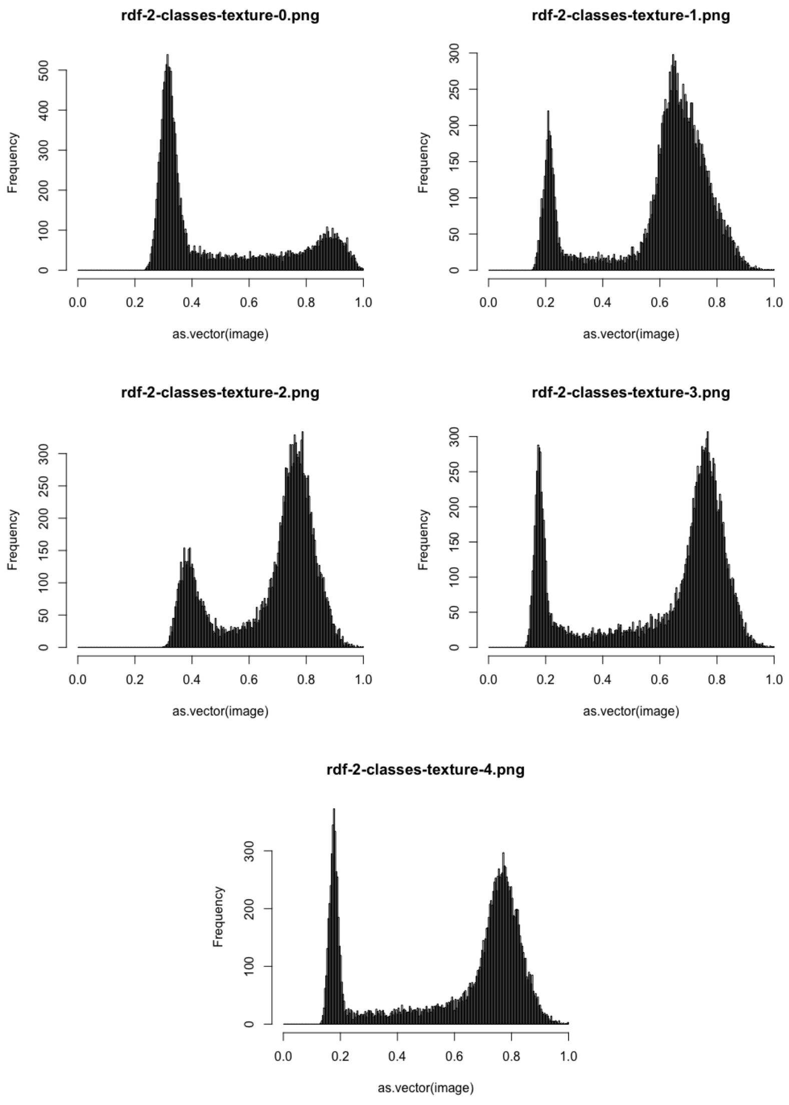

FIGURE 2.7
Histograms of gray values for each texture

An interesting observation is that we now have a "valley" for each texture, even for the noisy ones. Next, we can use the same "thresholding" method as in our previous experiment 2.2.2, but applied on the images obtained from *rdfTextureEcartType()*.

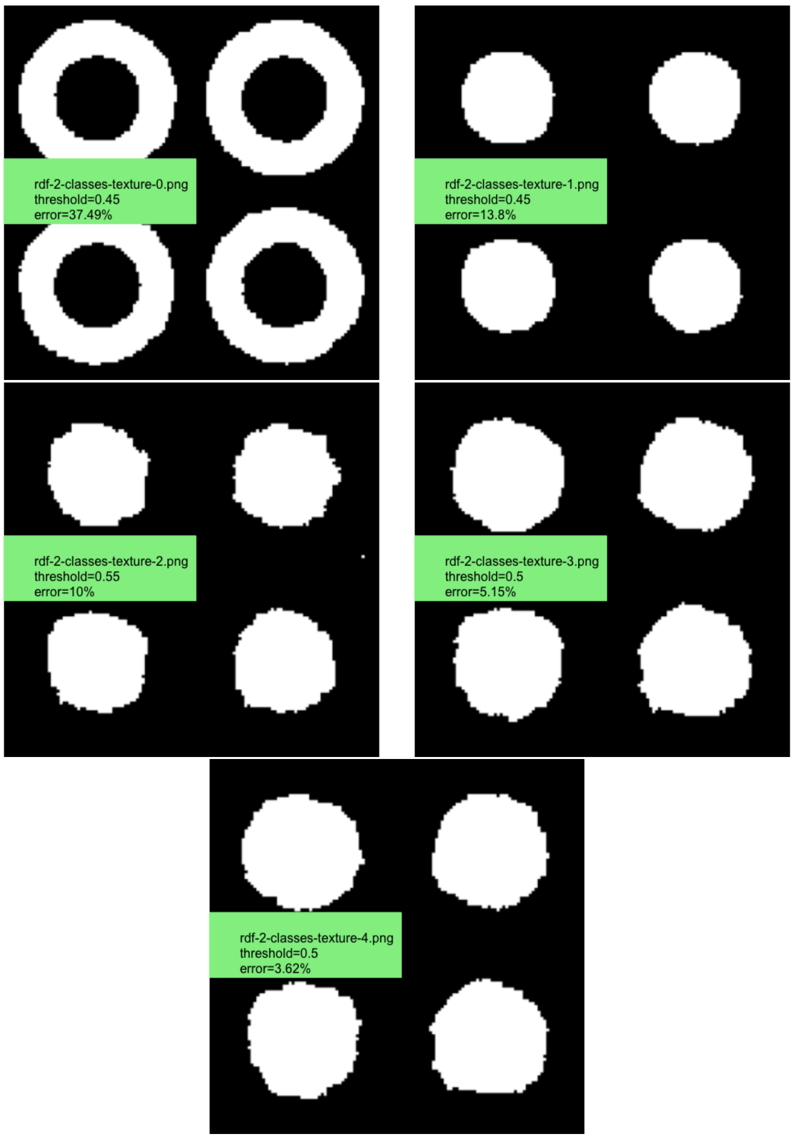
rdf-2-classes-texture-0.png
threshold=0.45
error=37.49%
rdf-2-classes-texture-1.png
threshold=0.45
error=13.8%
rdf-2-classes-texture-2.png
threshold=0.55
error=10%
rdf-2-classes-texture-3.png
threshold=0.5
error=5.15%
rdf-2-classes-texture-4.png
threshold=0.5
error=3.62%

FIGURE 2.8
Thresholding based on texture

Our biggest win here, using texture levels, is that for the last 2 textures, the segmentation works much better, due to the "mathematical blur" we applied by calculating the average gray values. However, this is still far from perfect, especially for the first texture, where our segmentation resulted in... donuts.

As a conclusion for this experiment, it seems like what "gray thresholding" does bad, "texture thresholding" does good and viceversa. Given this, it might be a good idea to try and combine the two methods.

CHAPTER 3

CONCLUSION

3.0.1 LIMITS AND ACKNOWLEDGEMENTS

In order to achieve a “perfect” binary image, one must properly set the threshold for the pixels’ values. In this report, we did that by hand, which is not ideal, especially given the fact that we would most likely want to work with many images. Therefore, developing an algorithm for setting this threshold would be ideal. It should automatically detect the value for which the “level of noise” is minimum. How we’d define the latter is, as well, a problem of discussion.

Taking advantage of the methods presented in this report, one is now able to code and decode a shape’s contour with ease. Shapes that resemble a sequence of concatenated segments can be drastically simplified with the Chord algorithm and irregular shapes can be “smoothened” using the Fourier method. Nevertheless, both methods can be used to simplify the shapes in some way.

We may even state that we’re able to do some kind of “shape compression”, for example by only saving the non-zero Fourier descriptors (for the first method), and only the points resulted from the Chord algorithm (for the second one). We can further take advantage of the parametrisation of the methods (i.e. *ratio* and d_{max}) to best suit our needs. In conclusion, these allow us to now work with a shape’s contour.

Bibliography

RDocumentation (2020). *filter2 function documentation*. DataCamp. URL: `https://www.rdocumentation.org/packages/EBImage/versions/4.14.2/topics/filter2` (visited on 4th Feb. 2020).