SHAPE CONTOURS

RECONNAISSANCE DES FORMES

Iacob Sergiu

4th 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 TECHNICAL ACKNOWLEDGEMENTS

The following code presented was written using *R*. The images from which the shapes were extracted were used as gray images. 2.1

2.2 LOADING DATA

In the first phase, we'll be working with files containing pre-defined contour points. These are just pairs (x, y) representing the coordinates of the outer points of a shape's contour. To simplify things, we'll transform the coordinates to complex numbers: Wikipedia 2020a.

```
# Lit un contour dans un fichier texte
rdfChargeFichierContour <- function (nom) {
    contour <- read.table (nom)
    complex (real = contour$V1, imaginary = contour$V2)
}
```

LISTING 2.1
Loading outer contour points

Images used are loaded then converted to grayscale. Right now, we are not interested in their color, so they'll only be black and white.

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

LISTING 2.2
Loading images

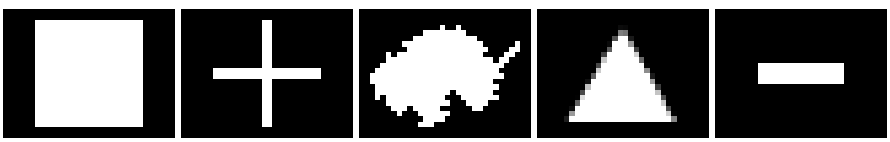

FIGURE 2.1
Shape images

2.3 FOURIER DESCRIPTORS

2.3.1 OUTER CONTOUR POINTS

Let's take a look at how one of our initial shape looks like. The red contour contains all of our points. The blue one is drawn between each 4th point, and the green one is drawn between each 8th point.

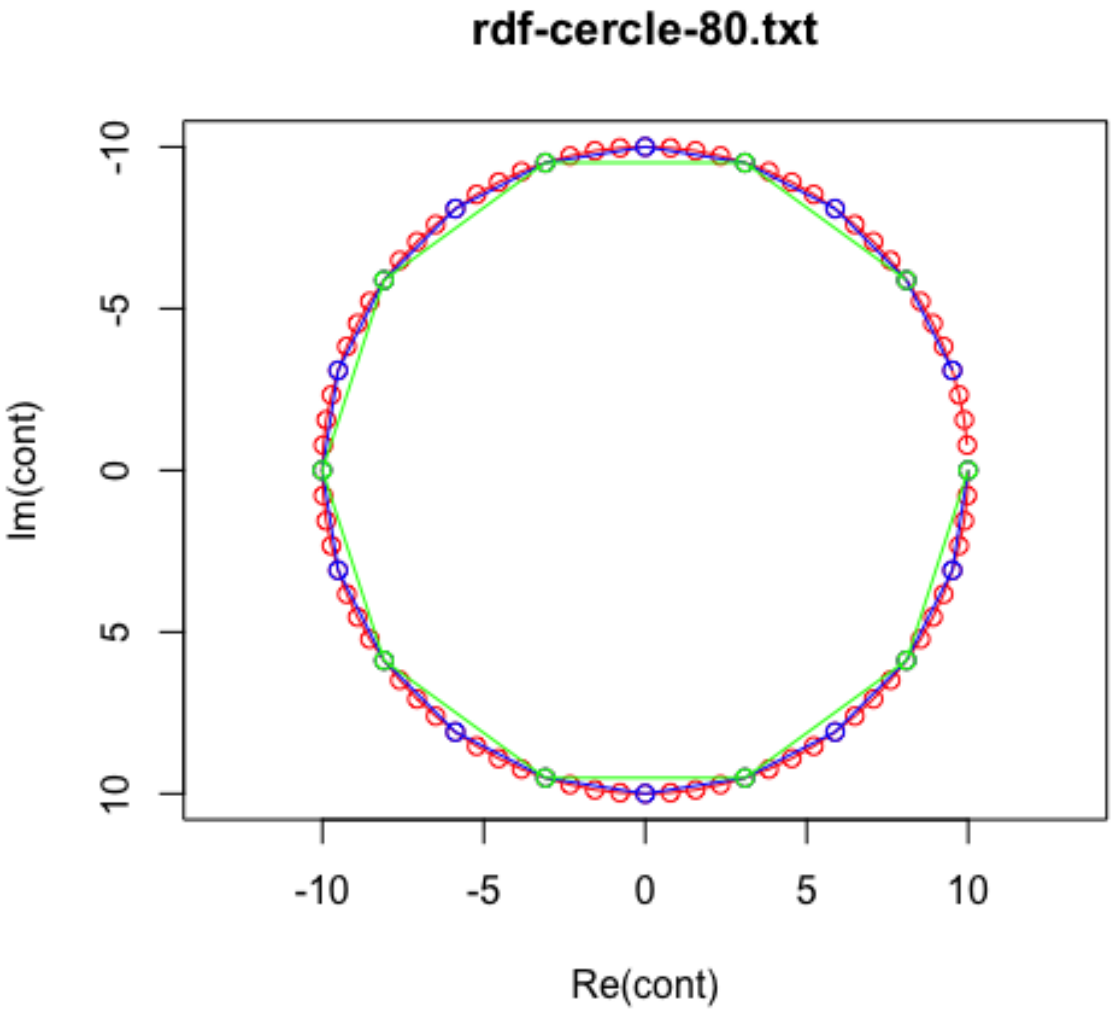

FIGURE 2.2
Circle contour

If we apply the *Fast Fourier Transform* on our contour points, we'll get our *Fourier descriptors*[1]. We can use these to describe our contour and we're even able to reconstruct our contour starting from fourier descriptors.

```
1  reconstructShapeWithFourier <- function(nom){
2      cont <- rdfChargeFichierContour(nom)
3      fourier <- fft(cont)/length(cont)
4      inversed <- fft(fourier, inverse=TRUE)
5      print ("Can we reconstruct our shape from Fourier descriptors?")
6      print (all.equal(inversed, cont))
7  }
```

LISTING 2.3
Calculating Fourier descriptors

[1]The Fourier descriptors were normalised by dividing them to the length of the total contour points

Here's how the Fourier descriptors look like calculated on the contour of the square:

```
> reconstructShapeWithFourier("rdf-carre-80.txt")
 [1]  0.00000000+0.00000000i -8.10986264-8.10986264i  0.00000000+0.00000000i  0.00000000+0.00000000i
 [5]  0.00000000+0.00000000i -0.32842678-0.32842678i  0.00000000+0.00000000i  0.00000000+0.00000000i
 [9]  0.00000000+0.00000000i -0.10434317-0.10434317i  0.00000000+0.00000000i  0.00000000+0.00000000i
[13]  0.00000000+0.00000000i -0.05235586-0.05235586i  0.00000000+0.00000000i  0.00000000+0.00000000i
[17]  0.00000000+0.00000000i -0.03261346-0.03261346i  0.00000000+0.00000000i  0.00000000+0.00000000i
[21]  0.00000000+0.00000000i -0.02318122-0.02318122i  0.00000000+0.00000000i  0.00000000+0.00000000i
[25]  0.00000000+0.00000000i -0.01808078-0.01808078i  0.00000000+0.00000000i  0.00000000+0.00000000i
[29]  0.00000000+0.00000000i -0.01515659-0.01515659i  0.00000000+0.00000000i  0.00000000+0.00000000i
[33]  0.00000000+0.00000000i -0.01349426-0.01349426i  0.00000000+0.00000000i  0.00000000+0.00000000i
[37]  0.00000000+0.00000000i -0.01267511-0.01267511i  0.00000000+0.00000000i  0.00000000+0.00000000i
[41]  0.00000000+0.00000000i -0.01251930-0.01251930i  0.00000000+0.00000000i  0.00000000+0.00000000i
[45]  0.00000000+0.00000000i -0.01299458-0.01299458i  0.00000000+0.00000000i  0.00000000+0.00000000i
[49]  0.00000000+0.00000000i -0.01420127-0.01420127i  0.00000000+0.00000000i  0.00000000+0.00000000i
[53]  0.00000000+0.00000000i -0.01642038-0.01642038i  0.00000000+0.00000000i  0.00000000+0.00000000i
[57]  0.00000000+0.00000000i -0.02026843-0.02026843i  0.00000000+0.00000000i  0.00000000+0.00000000i
[61]  0.00000000+0.00000000i -0.02712848-0.02712848i  0.00000000+0.00000000i  0.00000000+0.00000000i
[65]  0.00000000+0.00000000i -0.04049786-0.04049786i  0.00000000+0.00000000i  0.00000000+0.00000000i
[69]  0.00000000+0.00000000i -0.07131611-0.07131611i  0.00000000+0.00000000i  0.00000000+0.00000000i
[73]  0.00000000+0.00000000i -0.16965274-0.16965274i  0.00000000+0.00000000i  0.00000000+0.00000000i
[77]  0.00000000+0.00000000i -0.90481100-0.90481100i  0.00000000+0.00000000i  0.00000000+0.00000000i
[1] "Can we reconstruct our shape from Fourier descriptors?"
[1] TRUE
```

FIGURE 2.3
Reconstructing contour from Fourier descriptors

As we can see, we can safely reconstruct our initial contour thanks to the *Inverse Fast Fourier Transform*: Wikipedia 2020b, FFT. An interesting observation is that the first Fourier descriptor, Z_0, represents the center of the shape. If we modify this complex number (for example $Z_0 = Z_0 + 0.5 + 3i$) we'll actually do a *translation*.

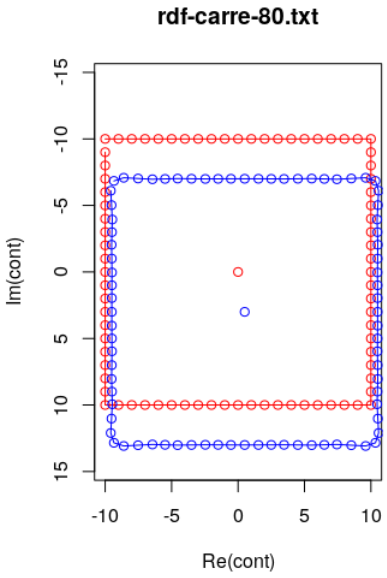

FIGURE 2.4
Modifying Z_0

2.3.2 CANCELLING FOURIER DESCRIPTORS

We can simplify our contour by "cancelling" some of the Fourier descriptors. We can do so by defining a *ratio* of descriptors that we'd like to keep. When doing so, it is important to know that we should start cancelling the descriptors from the middle of the descriptor list: Cabestaing 2020, RDF Course.

```
1   rdfAnnuleDescFourier <- function (desc, ratio){
2       if (ratio == 1.0)
3           desc
4
5       nb_delete <- length(desc) - length(desc) * ratio
6       middle <- length(desc)/2
7       start <- middle - nb_delete/2
8       end <- middle + nb_delete/2
9       desc[start:end] <- 0
10      desc
11  }
```

LISTING 2.4
Simplifying contour

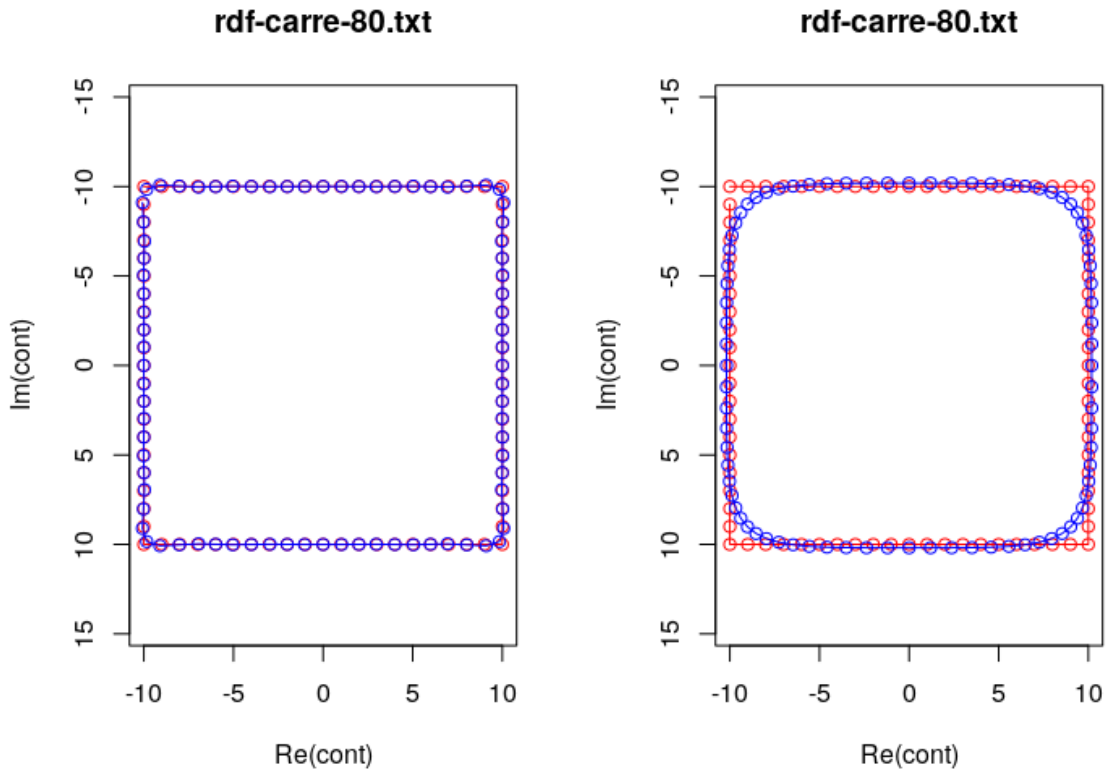

FIGURE 2.5
Cancelling Fourier descriptors, $ratio \in \{0.5, 0.075\}$

2.4 SIMPLIFYING THE SHAPE CONTOUR

We can make use of *circle's chord* to reduce the number of outer contour points. We'll reduce our shape to a sequence of segments between points. Given a sequence of contour points $S = P_1, \ldots, P_n$ and P_1P_n the line between the first and last point, we define X as the "furthest" point from this line and d_X the distance from X to P_1P_n. If d_X is less than some d_{max}, then we add the segment P_1P_n as part of our "simplified" contour. If not, we divide S into $S_1 = P_1, \ldots, X$ and $S_2 = X, \ldots, P_n$ and apply the same algorithm as above. Please refer to Wikipedia 2020c and Wikipedia 2020d for how the distances were calculated.

```
# Algorithme de la corde pour la reduction d'un contour
rdfAlgorithmeCorde <- function (cont, dmax) {
    # Calcul des distances
    d <- rdfDistances (cont)
    # Si distance maxi inferieur au seuil, ne garder que les extremites
    if (max (d) <= dmax) {
        c (head (cont, 1), tail (cont, 1))
    # Sinon decouper en deux parties
    } else {
        # Point le plus eloigne
        loin <- which.max (d)
        # Reduire les deux sous chaines
        cont1 <- rdfAlgorithmeCorde (cont[1:loin], dmax)
        cont2 <- rdfAlgorithmeCorde (cont[loin:length (cont)], dmax)
        # Enlever un point et contatener
        c (cont1, tail (cont2, -1))
    }
}
```

LISTING 2.5
Chord algorithm

```
# Calcul des distances entre les points et la corde
rdfDistances <- function (cont) {
    # Points extremes
    debut = head (cont, 1)
    fin = tail (cont, 1)
    distances <- rep (0, length (cont))
    for (i in 1:length(cont)){
        distances[i] <- rdfDistanceToLine(cont[i], debut, fin)
    }
    distances
}
```

LISTING 2.6
Calculating distances from P_1P_n to each point

```
rdfDistanceToLine <- function (p, p1, p2){
    x1 <- Re(p1)
    y1 <- Im(p1)
    x2 <- Re(p2)
    y2 <- Im(p2)

    a <- y2 - y1
    b <- -(x2 - x1)
    c <- -x1*(y2 - y1) + y1*(x2-x1)

    res <- abs(a * Re(p) + b * Im(p) + c)/sqrt(a**2 + b**2)
    res
}
```

LISTING 2.7
Distance from a point to a line

Being able to parametrise the algorithm with d_{max} will allow us to reach the trade-off between simplicity and precision that we mentioned in the introduction.1.3 Let's take a look at the results:

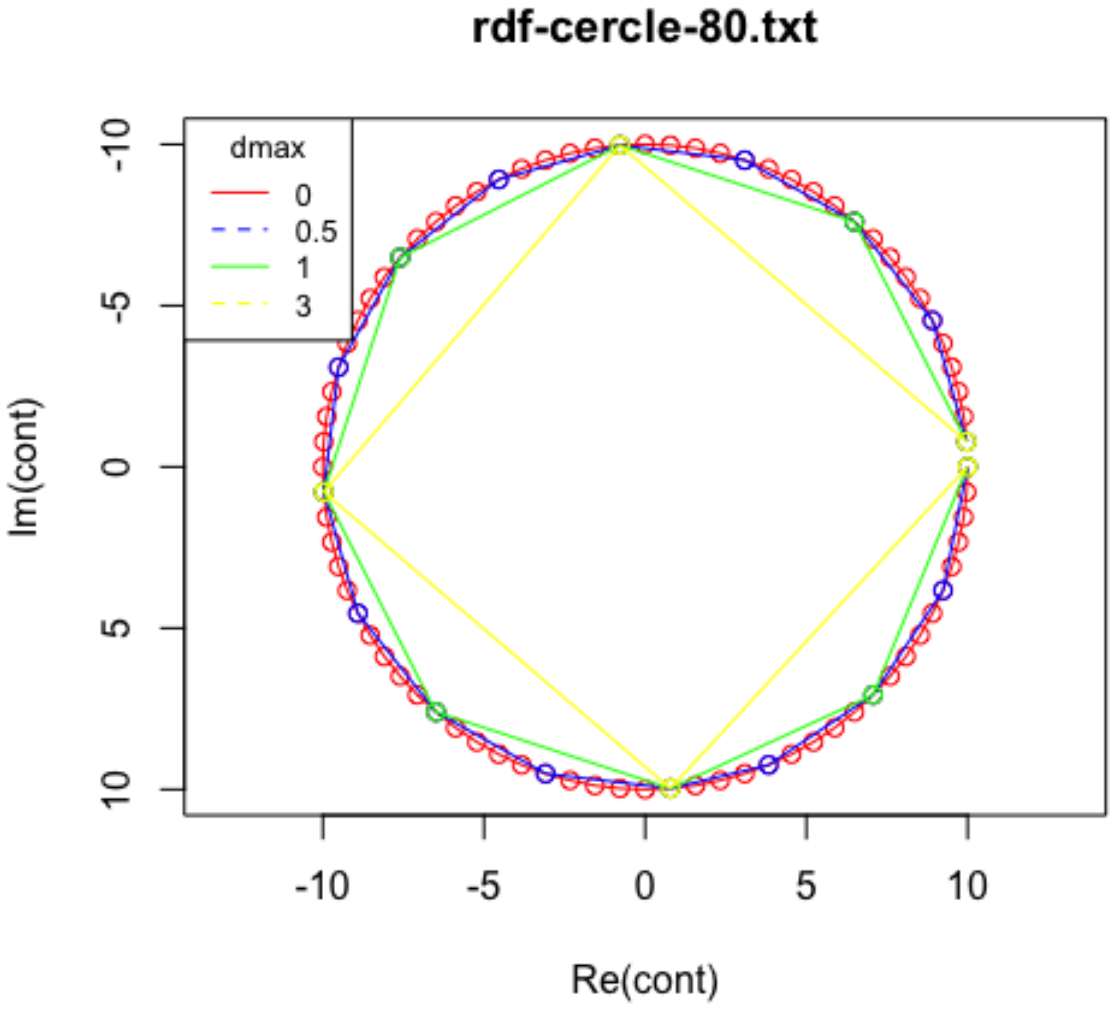

FIGURE 2.6
Chord algorithm, $d_{max} \in \{0, 0.5, 1, 3\}$

2.5 FOURIER DESCRIPTORS VS. CHORD ALGORITHM

A natural question we may ask ourselves now is: which method is better? Well, like in most Computer Science questions, the answer is the same: it depends. Let's have a look at how each method can be used to describe the contour of a shape:

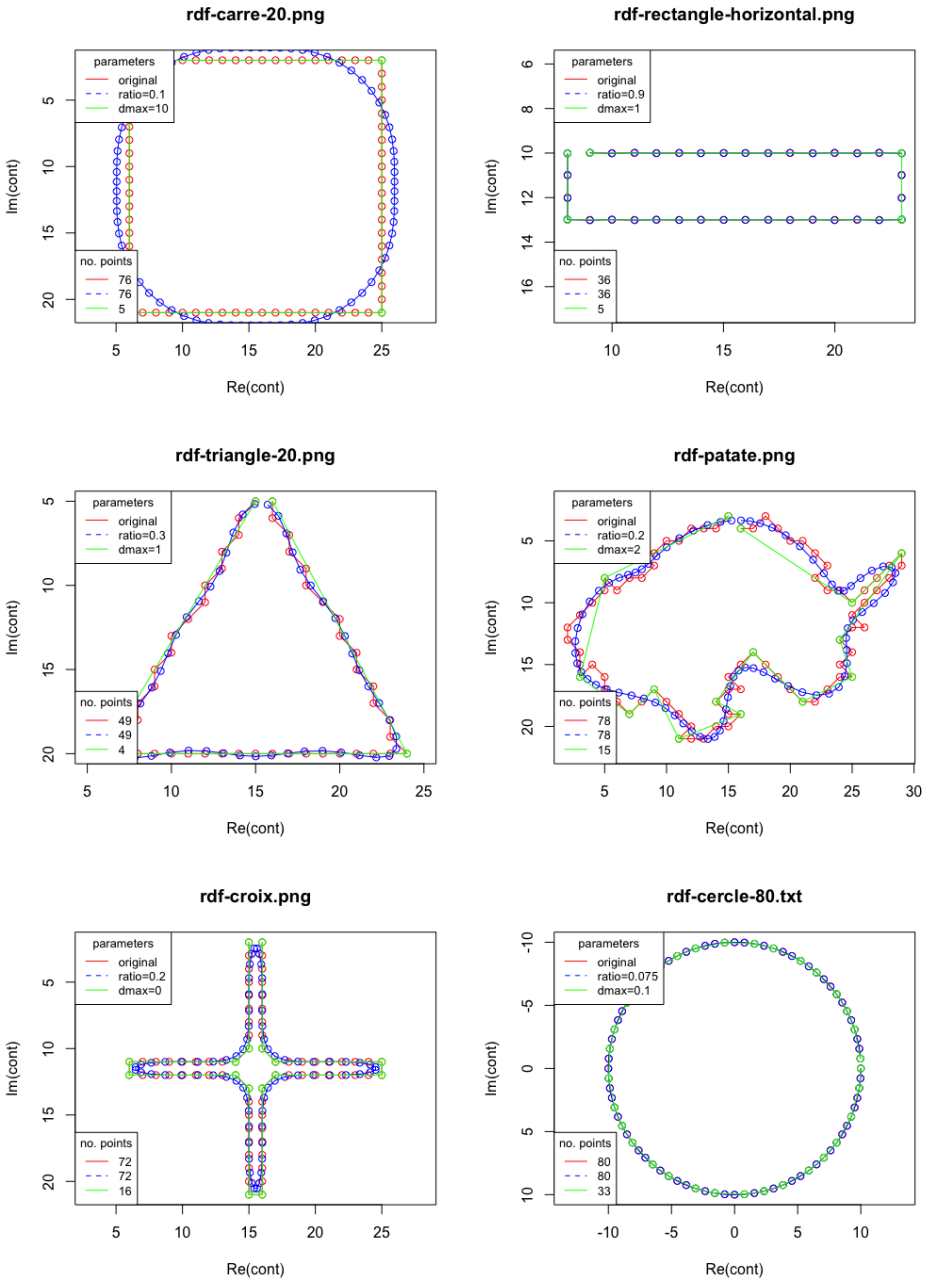

FIGURE 2.7
Comparison between encoding methods

If we take a good look at the figures2.7, we can clearly see that the *Chord Algorithm* is much better for shapes without curves. For example, for the square and for the rectangle, we can reduce the initial number of points to just *5*! The same goes for triangle and the cross, where we can drastically reduce the number of points necessary.

Taking the rectangle as an example, the *Fast Fourier Transform* method requires to keep around 90% of the descriptors to be able to properly reconstruct the shape. However, this method is much better for shapes with many curves. Taking the circle as an example, the Fourier method needs just 7.5% of the descriptors to be able to reconstruct the circle without any kind of problems. Also, for the irregular shape ("patate"), we are able to construct a much *smoother* shape contour than by using the Chord algorithm. Therefore, depending on our needs and on the shapes we're working with, both methods might bring advantages and disadvantages.

CHAPTER 3

CONCLUSION

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

Wikipedia (2020a). *Complex number*. Wikimedia Foundation. URL: https://en.wikipedia.org/wiki/Complex_number (visited on 4th Feb. 2020).

– (2020b). *Fast Fourier Transform*. Wikimedia Foundation. URL: https://en.wikipedia.org/wiki/Fast_Fourier_transform (visited on 4th Feb. 2020).

Cabestaing, François (2020). *Reconnaissance des formes*. Université de Lille. URL: http://master-ivi.univ-lille1.fr/Cours/RdF (visited on 4th Feb. 2020).

Wikipedia (2020c). *Distance from a point to a line*. Wikimedia Foundation. URL: https://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line (visited on 4th Feb. 2020).

– (2020d). *Line Equation for Complex Numbers*. Wikimedia Foundation. URL: https://proofwiki.org/wiki/Equation_for_Line_through_Two_Points_in_Complex_Plane/Examples/2%2Bi,_3-2i/Standard_Form (visited on 4th Feb. 2020).