Computer Vision

My Personal Notes

10.5 Zernike Moments

Like Hu Moments, we can use ***Zernike Moments*** to *characterize and quantify the shape* of an object. However, Zernike Moments are *more powerful* and generally *more accurate* image descriptors with very little additional computational cost.

- The shape in an image we wish to describe can be either the *outline* (i.e. "boundary") of the shape or *a mask* (i.e. "filled in boundary") of the shape we want to describe. In most real world applications, it's *common to use the shape mask* since it's less susceptible to noise.

Zernike Moments *are not* available in OpenCV. To utilize and extract Zernike Moments we'll be using the ***mahotas*** package.

What are Zernike Moments used to describe?

Zernike Moments are an image descriptor used to *characterize the shape of an object* in an image. The shape to be described can either be a *segmented binary image* or the *boundary of the object* (i.e. the "outline" or "contour" of the shape).

In most applications it is *preferable to use the segmented binary image* rather than just the outline since the segmented binary image is less susceptible to noise.

How do Zernike Moments work?

Zernike Moments were first introduced by ***Teague*** in the 1980 paper, *Image Analysis via General Theory of Moments* (the original paper can be found here; English version of Teague's paper Here). Up until this point, Hu Moments were primarily used as shape descriptors.

The work by Teague introduced a *new shape descriptor* that generally *out-performed Hu Moments*. This shape descriptor is called Zernike Moments and is *based on the theory* of ***orthogonal functions***.

Examples of orthogonal functions include the ***sine*** and ***cosine*** function:

Privacy & Cookies: This site uses cookies. By continuing to use this website, you agree to their use.
To find out more, including how to control cookies, see here: Cookie Policy

Close and accept

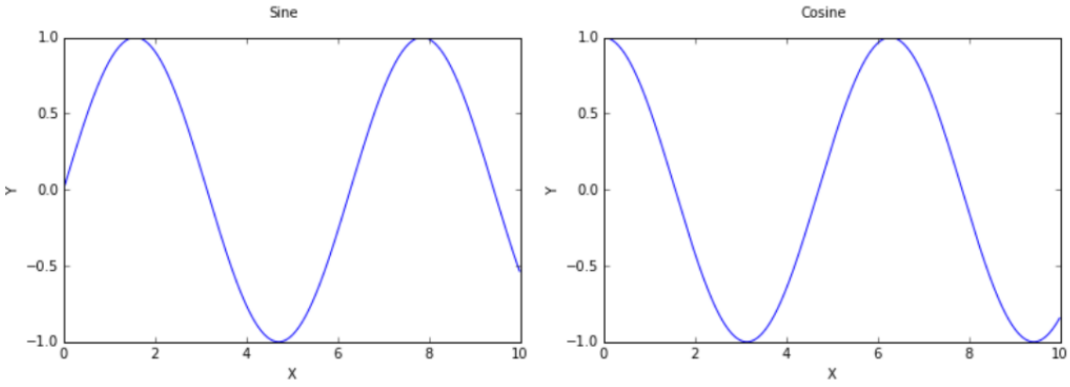


FIGURE 1: SINE AND COSINE ARE EXAMPLES OF ORTHOGONAL FUNCTIONS.

Two functions, such as *sine* and *cosine*, are considered orthogonal if their inner product (i.e. sum of elements) is zero (or sufficiently close to zero). For a thorough treatment of the sine and cosine orthogonal proof, take a look at this article.

The benefit of *orthogonal functions* is that there is ***no redundancy of information between moments***, making them more robust and discriminative than Hu Moments, which are based on simple statistical derivations.

Luckily, *Zernike polynomials* are *orthogonal* over a disk with radius ***r*** (specified in polar coordinates), thus making them applicable to computer vision and shape description:

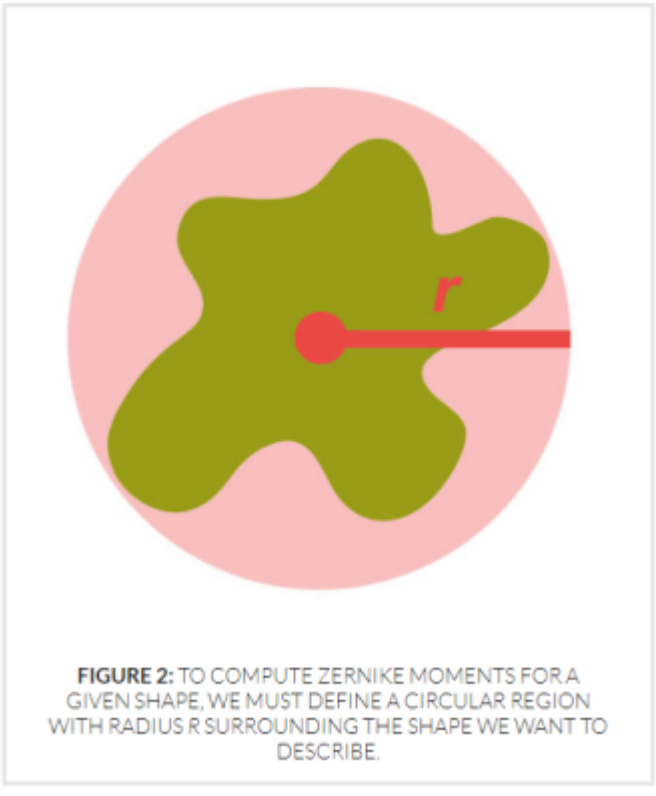


FIGURE 2: TO COMPUTE ZERNIKE MOMENTS FOR A GIVEN SHAPE, WE MUST DEFINE A CIRCULAR REGION WITH RADIUS R SURROUNDING THE SHAPE WE WANT TO DESCRIBE.

In the above image we have a green shape that we want to compute Zernike Moments for. We then place a disk surrounding our shape with a radius ***r***. The radius *r* should technically be set properly to include the en-

Privacy & Cookies: This site uses cookies. By continuing to use this website, you agree to their use.
To find out more, including how to control cookies, see here: Cookie Policy

Close and accept

To compute Zernike Moments we specify 2 parameters: *the **radius** of the disc* and the ***degree** of the polynomial*. The radius is thus the region of which the polynomials are defined.

First, the input image is mapped to a disc with radius *r*, where the center of the image is placed at the origin of the disc.

Pixels that fall outside the disc are ignored and not included in the computation, so be sure to choose your radius thoughtfully.

From there, Zernike Moments up to degree ***d*** are computed and utilized as the feature vector. The *size* of the returned *feature vector is directly controlled by the degree* of the polynomial. ***The larger the degree, the larger the feature vector***.

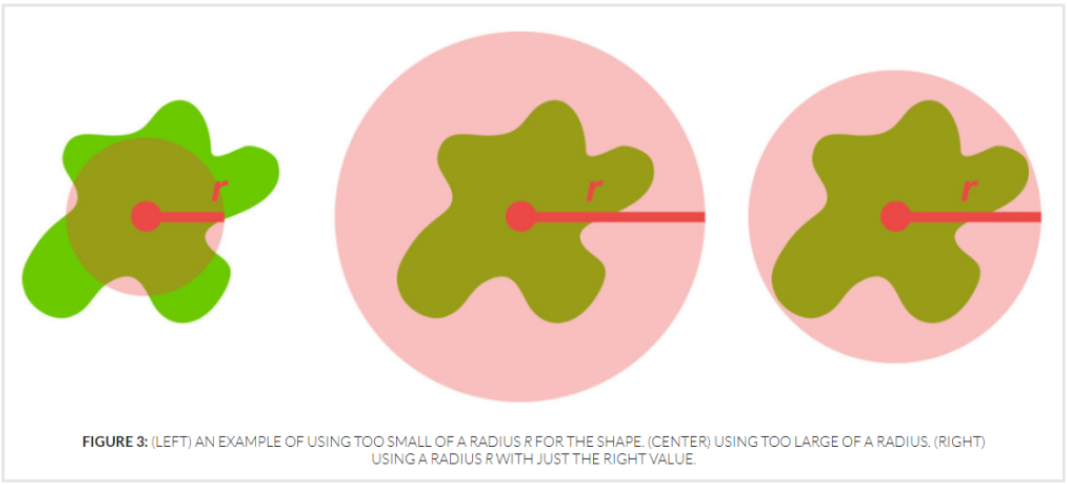


FIGURE 3: (LEFT) AN EXAMPLE OF USING TOO SMALL OF A RADIUS *R* FOR THE SHAPE. (CENTER) USING TOO LARGE OF A RADIUS. (RIGHT) USING A RADIUS *R* WITH JUST THE RIGHT VALUE.

Mathematical formulation is outside the scope of this post. However, if you are interested in learning more about the formulation of Zernike polynomials, be sure to take a look at the following articles:

- Moment-based approaches in imaging: Somewhat lengthy, but provides depth into the mathematical details behind Zernike Moments.
- Complex Zernike Moments: Quite short, but the mathematics are only briefly explained.
- Moments in Image Processing: My personal favorite article for explaining Zernike Moments. The formatting of the article is a bit funky with strange unicode characters, but overall it provides a good tradeoff between mathematical depth and high-level overview.

Where are Zernike Moments implemented?

Zernike Moments are implemented inside the **mahotas** Python package. Click here for the official documentation on the zernike_moments function.

How do I use Zernike Moments?

Privacy & Cookies: This site uses cookies. By continuing to use this website, you agree to their use.
To find out more, including how to control cookies, see here: Cookie Policy

Close and accept

In this post, we are going to learn to apply Zernike Moments (which are much more powerful than Hu Moments) to actually *recognize* an object in an image.

We'll require 2 images.

- The first image will be a *reference image* of the object we want to detect.
- The second image will be a *distractor image* containing (1) the object we want to find and identify and (2) a bunch of "distractor" objects mean to "confuse" our algorithm.
- **Our goal will be to successfully detect the reference image in the second image.**

Let's go ahead and introduce our two images. The first image, our reference image, can be seen below:

FIGURE 4: OUR OBJECT REFERENCE IMAGE. WE'LL BE CHARACTERIZING THE SHAPE OF THIS OBJECT USING ZERNIKE MOMENTS AND THEN FINDING THE CORRESPONDING SHAPE IN THE SECONDARY IMAGE BELOW.

Privacy & Cookies: This site uses cookies. By continuing to use this website, you agree to their use.
To find out more, including how to control cookies, see here: Cookie Policy

Close and accept

FIGURE 5: THIS IMAGE CONTAINS THE GAME BOY ROM CARTRIDGE WE WANT TO DETECT, ALONG WITH THREE OTHER "DISTRACTOR" OBJECTS MEAN TO "CONFUSE" OUR ALGORITHM.

In this image we can see that we have the Pokemon game cartridge. But also also have a bunch of other distractor objects, such as scissors, a highlighter, and a sticky note.

** Again, our goal is to be able to detect and recognize the Pokemon game cartridge while ignoring the other distractor objects by using simple shape descriptors — namely, Zernike Moments.

Code : [***detect_game.py***]

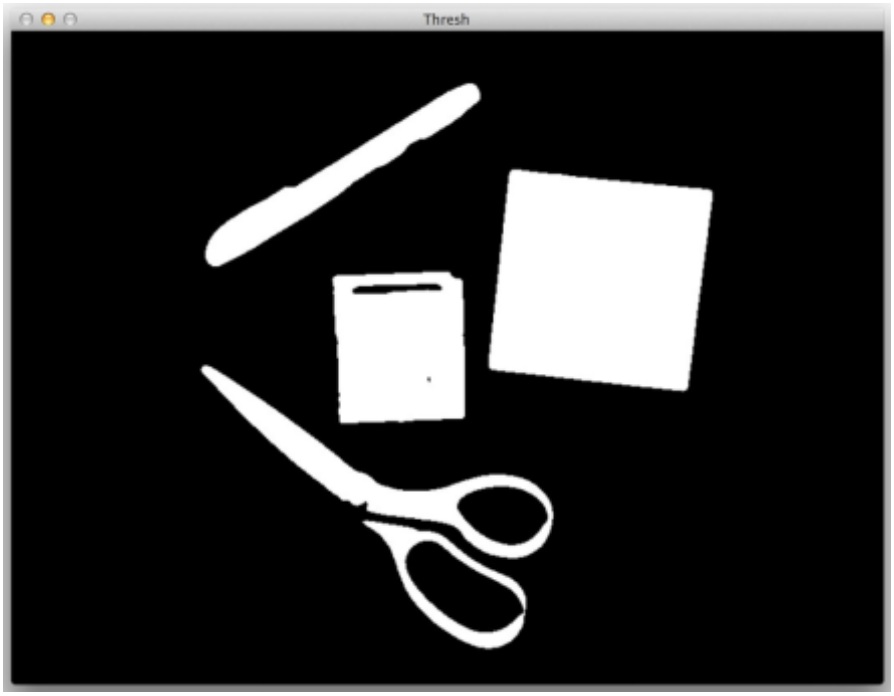
The first thing we'll do here is define a ***describe_shapes*** function. This function will be applied to the Pokemon game cartridge in the reference image along with *every* object in the second image. Given that this section will be called so many times, I thought it best to create a dedicated function to describing the shape regions.

- The *describe_shapes* function will take a single argument, an image that contains the objects and shapes we want to quantify using Zernike Moments.

We'll need to apply a little bit of pre-processing to our image before we can extract our shape features, so we'll convert the image to grayscale, blur it to remove high frequency noise and allow us to focus on the structural aspects of the image, followed by thresholding to segment the objects from the background.

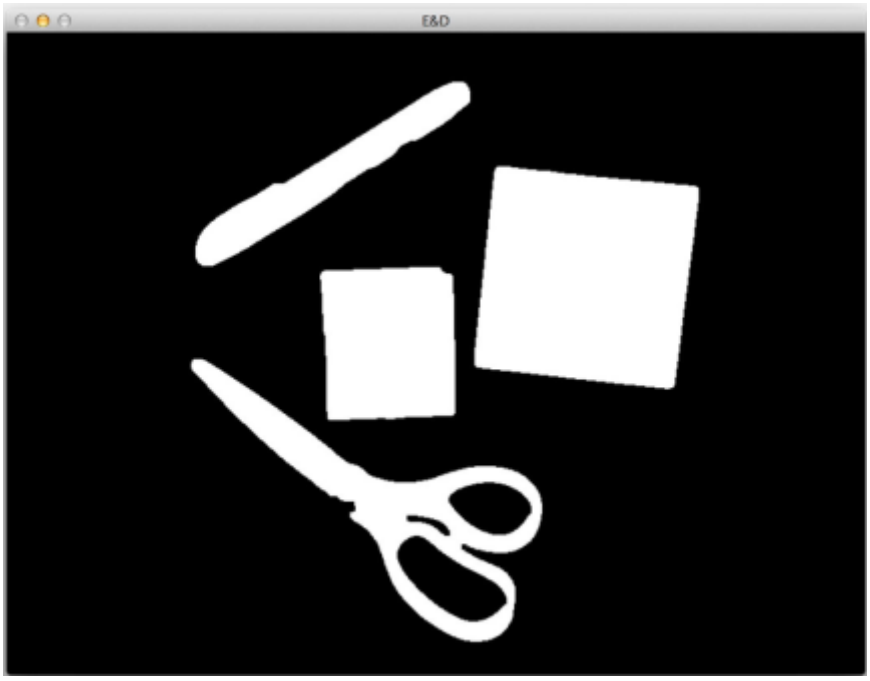
Privacy & Cookies: This site uses cookies. By continuing to use this website, you agree to their use.
To find out more, including how to control cookies, see here: Cookie Policy

Close and accept



APPLYING THRESHOLDING TO SEGMENT THE FOREGROUND OBJECTS FROM THE BACKGROUND.

We'll also perform a series of dilations and erosions to close gaps between any parts of objects.



APPLYING A SERIES OF EROSIONS CLOSES ANY GAPS AND HOLES BETWEEN PARTS OF OBJECTS.

Now that the **Morphological Operations** have been applied, we can then detect the contours of each of the objects in the image.

We start looping over each of the individual contours. And then for each of these contours, we allocate memory for a ***mask*** followed by drawing the contoured region on the mask.

Privacy & Cookies: This site uses cookies. By continuing to use this website, you agree to their use.
To find out more, including how to control cookies, see here: Cookie Policy

Close and accept

Given *the ROI of the shape* we can now extract *Zernike Moments* by making a call to the ***mahotas.features.zernike_moments*** function. This function accepts 3 parameters: *the image/ROI* that we want to quantify, the *radius* of the region, and the *degree* of the polynomial.

** ***Tuning the radius*** can be *non-trivial based on the sizes of the objects* in the image. If the sizes of the objects vary dramatically, as they do in our distractor image above, the value of this radius is often not immediately obvious.

To solve this problem, we have 2 solutions :

- The first solution is to simply ***resize the ROI*** to a known size. This ensures that we will be able to hard-code the radius value and that our images are described in a consistent manner.
    - The **problem** with this approach is that resizing to a fixed size ***can destroy the aspect ratio*** of the shape. This is especially troublesome if we are trying to quantify the shape of a region — by throwing away the aspect ratio, we throw away information regarding the dimensions of the shape, which is completely counterintuitive to our goal!

The other solution is what this example utilizes — we can *dynamically* compute the radius of the Zernike Moments simply by computing the ***cv2.minEnclosingCircle*** function (a **simple contour property**). This method will return the minimum size radius that can be used to enclose the *entire* object. By applying this method, we can ensure the radius of the moments covers the entire ROI.

- We also supply degree=8 to the *zernike_moments* function, which is the default degree of the polynomial. In most cases you'll need to tune this value until it obtains adequate results.

Finally, we can return a ***2-tuple*** from our *describe_shapes* function consisting of (1) the contours of the objects/shapes in the image, followed by (2) the Zernike Moments feature vectors corresponding to each shape.

So now that we have the *describe_shapes* function defined, let's see how we can use it to recognize the Pokemon game cartridge in an image:

We load our reference image of the Pokemon game cartridge from disk and describe shape of the game cartridge.

We load the distractor image from disk and quantify *all* of the shapes in the image using Zernike Moments.

To detect the actual game cartridge, we use SciPy's distance sub-module to compute the Euclidean distance between all pairs of the *gameFeatures* and *shapeFeatures*.

- Notice how we have only one row, which is the Zernike Moments associated with the game cartridge features. We also have multiple columns — one for each of the shapes in the shapeFeatures list

Privacy & Cookies: This site uses cookies. By continuing to use this website, you agree to their use.
To find out more, including how to control cookies, see here: Cookie Policy

Close and accept

Zernike Moments are used to quantify the shape of an object. If we assume shapes that have *similar feature vectors* also have *similar visual contents*, then the shape that minimizes the distance between Zernike Moments must be our reference image!

We'll start by looping over the contours in our distractor image. If the index of the current contour does not match the index of the contour with minimum distance in our distance matrix *D*, then we draw a rotated bounding box surrounding it in ***red***.

We then then draw a ***green*** rotated bounding box surrounding our identified game cartridge region, followed by displaying the text "FOUND!" directly above the bounding box.

```
# import the necessary packages
from scipy.spatial import distance as dist
import numpy as np
import mahotas
import cv2
import imutils

def describe_shapes(image):
    # initialize the list of shape features
    shapeFeatures = []

    # convert the image to grayscale, blur it, and threshold it
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    blurred = cv2.GaussianBlur(gray, (13, 13), 0)
    thresh = cv2.threshold(blurred, 50, 255, cv2.THRESH_BINARY)[1]

    # perform a series of dilations and erosions to close holes
    # in the shapes
    thresh = cv2.dilate(thresh, None, iterations=4)
    thresh = cv2.erode(thresh, None, iterations=2)

    # detect contours in the edge map
    cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)
    cnts = imutils.grab_contours(cnts)

    # loop over the contours
    for c in cnts:
        # create an empty mask for the contour and draw it
        mask = np.zeros(image.shape[:2], dtype="uint8")
        cv2.drawContours(mask, [c], -1, 255, -1)

        # extract the bounding box ROI from the mask
        (x, y, w, h) = cv2.boundingRect(c)
        roi = mask[y:y + h, x:x + w]

        # compute Zernike Moments for the ROI and update the list
        # of shape features
        features = mahotas.features.zernike_moments(roi, cv2.minEnclosingCirc
        shapeFeatures.append(features)

    # return a tuple of the contours and shapes
    return (cnts, shapeFeatures)

# load the reference image containing the object we want to detect,
# then describe the game region
refImage = cv2.imread("pokemon_red.png")
(_, gameFeatures) = describe_shapes(refImage)

```

Privacy & Cookies: This site uses cookies. By continuing to use this website, you agree to their use.
To find out more, including how to control cookies, see here: Cookie Policy

Close and accept

```
D = dist.cdist(gameFeatures, shapeFeatures)
```

```
i = np.argmin(D)

# loop over the contours in the shapes image
for (j, c) in enumerate(cnts):
    # if the index of the current contour does not equal the index
    # contour of the contour with the smallest distance, then draw
    # it on the output image
    if i != j:
        box = cv2.minAreaRect(c)
        box = np.int0(cv2.cv.BoxPoints(box) if imutils.is_cv2() else cv2.boxP
        cv2.drawContours(shapesImage, [box], -1, (0, 0, 255), 2)

# draw the bounding box around the detected shape
box = cv2.minAreaRect(cnts[i])
box = np.int0(cv2.cv.BoxPoints(box) if imutils.is_cv2() else cv2.boxPoints(bo
cv2.drawContours(shapesImage, [box], -1, (0, 255, 0), 2)
(x, y, w, h) = cv2.boundingRect(cnts[i])
cv2.putText(shapesImage, "FOUND!", (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9
    (0, 255, 0), 3)

# show the output images
cv2.imshow("Input Image", refImage)
cv2.imshow("Detected Shapes", shapesImage)
cv2.waitKey(0)
```

To see Zernike Moments in action, just execute the following command:

```
$ python detect_game.py
```

Output :

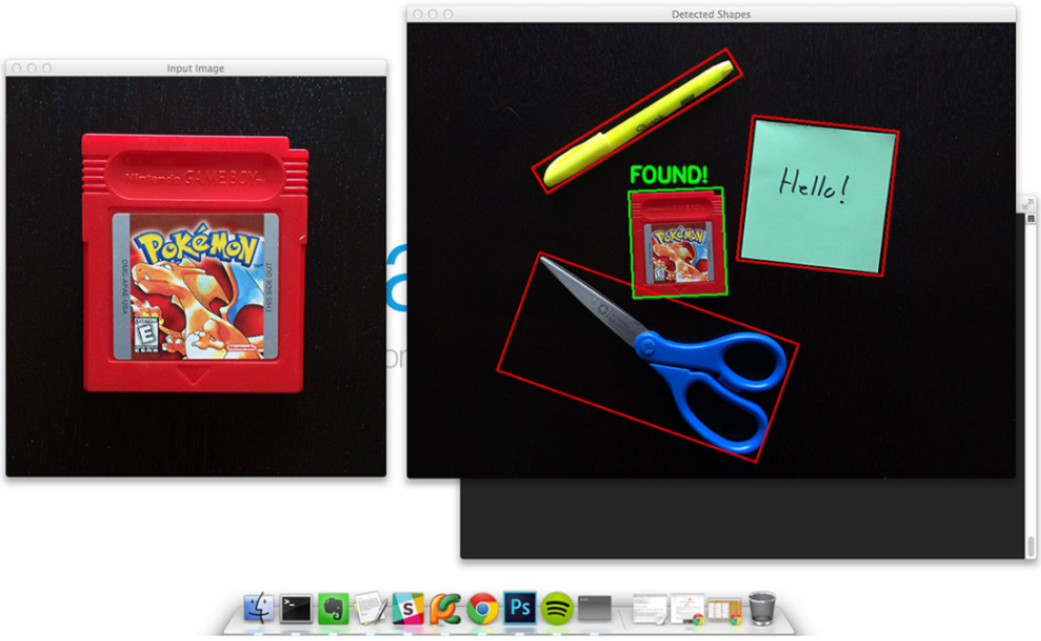


FIGURE 9: OUR ORIGINAL REFERENCE OBJECT IMAGE OF THE GAME BOY ROM CARTRIDGE ON THE *LEFT*, FOLLOWED BY DETECTING THE GAME BOY CARTRIDGE IN OUR SECOND IMAGE ON THE *RIGHT*.

On the *left* we have our original reference image of the Pokemon game cartridge. And on the *right* we have

Privacy & Cookies: This site uses cookies. By continuing to use this website, you agree to their use.
To find out more, including how to control cookies, see here: Cookie Policy

Close and accept

However, by applying Zernike Moments features we were able to discard the distractor shapes and detect the actual game cartridge shape in the image.

Suggestions when using Zernike Moments:

If you need to describe *multiple* shapes in an image, be sure to extract the ROI of each object, *and then extract Zernike Moments from each ROI*.

Secondly, be sure to pay attention to the radius and degree parameters. Zernike Moments map the shape you want to describe onto a disc with radius of *r* pixels. If there are any pixels of your shape that fall outside the radius *r*, then ***these pixels will be ignored!*** Take special care to ensure your radius is sufficiently large before extracting Zernike Moments.

Also be sure to pay attention to the degree parameter. The degree parameter directly affects the dimensionality of your resulting feature vector. The larger your value of degrees, *d*, the larger (and hopefully more discriminative) your feature vector is. However, as your increase the number of degrees *d* of the polynomial, your computational cost will increase.

When setting the *radius* and *degree* parameters of Zernike Moments, take special care to consider the radius first. Reflect on how large the radius should be to sufficiently capture the shape of the objects in your dataset. From there it becomes much easier to tune the degree parameter which can influence the dimensionality of the feature vector. I personally like to start with degree=8 and increase/decrease as necessary.

Pros:

- Very fast to compute.
- Low dimensional.
- Very good at describing simple shapes.
- Fairly simple to tune the *radius* and *degree* parameters.

Cons:

- Requires a very precise segmentation of the object to be described.
- Normally *only used for simple 2D shape segmentation* — as shapes become more complex, Zernike Moments often do not perform well.
- Just like Hu Moments, Zernike Moments are calculated based on the initial centroid computation — if this centroid is not repeatable for similar shapes, the Zernike Moments will not obtain good matching accuracy.

This entry was posted in Uncategorized on July 21, 2020

Privacy & Cookies: This site uses cookies. By continuing to use this website, you agree to their use.
To find out more, including how to control cookies, see here: Cookie Policy

Close and accept