[ELE00142M] Machine Vision and Human Machine Interaction

UNIVERSITY of York

Lab 1: Image and Feature Processing with OpenCV

Part 1: Filtering and Segmentation

Lecturer: Mark A Post <mark.post@york.ac.uk>

Demonstrator: John Bateman <john.bateman@york.ac.uk>

Technician: Alejandro Pascual San Roman <alejandro.pascualsanroman@york.ac.uk>

Aims and Objectives

In this lab session you will be experimenting with the Open Source Computer Vision Library (OpenCV) using a USB camera on a laboratory desktop workstation. The focus of this lab will be the processing of image data to extract features from, segment, and characterize the image using common image processing algorithms implemented in OpenCV.

It is expected that you will complete all of these activities in two weeks of labs (Week 2 - Week 3). The P/T/410 and P/T/411 laboratories are open for your use outside of scheduled lab times in case you need extra time to finish. You can also work at home by installing the Python language and OpenCV-Python on your own computer.

Learning outcomes

- Opening images in OpenCV and converting them to matrix data
- Use of colour masking, filtering, and segmentation features in OpenCV
- Identification of line, corner, and descriptor features in images at pixel level

Hardware and Software required

- Python 3 with OpenCV-Python and numpy libraries installed
- Logitech C930 or similar USB camera with autofocus
- Test image files *gradient.png*, *shapes.png*, and *aruco.png* provided on the module webpage. They are also provided at the end of this document.

Tasks

1. Python Software Setup

We will be using the Python language in this module, as it is faster and easier to program with than C/C++ and has become the de-facto standard language for high-level machine vision, human-machine interaction, and artificial intelligence programming.

If you are not familiar with Python, you can find a quick introductory tutorial to the language in this Python Hands-On Tutorial Session, and there are many good tutorials available across the Internet that are easy to find through searching.

You can install OpenCV-Python using pip or pip3 with “pip3 install opencv-python --user” or “pip install opencv-python --user” (if your python version is only version 3). Numpy should be installed automatically as a dependency. If Python cannot find cv2 afterwards, try entering “help("modules")” in your Python console. Also, make sure that the version of Python that pip has installed to is the same as the version of Python used by your IDE, e.g. if you are using Spyder. If not you need to change Spyder or your IDE to use the same version of Python in the settings. You may also need to pip install spyder-kernels so that Spyder can use a kernel in your system version of Python.

There are also many Python Integrated Development Environments available for you to choose from, such as:

- ipython is just an enhanced python console/command line with additional features

- IDLE comes with Python by default, simple to use but with limited features.
- PyCharm is a well developed commercial IDE, the community version is free.
- Jupyter Notebook is web-based and portable, and supports rich output formats such as equations, visualisations, etc.
- Spyder is the recommended IDE for scientific use, designed to look like a MATLAB IDE.
- Online IDE lets you use basic Python in a browser: https://www.online-ide.com/

Spyder is recommended for ease of use and similarity to MATLAB/Octave, but IDLE is ubiquitous and easy to use. Both should be available on lab computers.

If you are not using an IDE, you can just open a python (or ipython, for extra functionality) console to run individual commands to test if they work. To make a Python script, just open a new text file, write your program and save it to have extension '.py'. Then run it with "python3 <filename>.py".

2. Reading an Image

OpenCV is an image processing library that operates on image data that is organized within a matrix data type. Have a quick look at the OpenCV API documentation at https://docs.opencv.org/4.x/ before you start. This is the main reference for all the functions you will be using for machine vision.

The first thing you will need to do is read an image into a matrix of numbers, each of which typically represents one "picture element" or "pixel".

Create a new Python file (just a text file ending in '.py' e.g. "lab1.py"). In all cases when you use OpenCV in Python you will first need to import the OpenCV library with

```
import cv2
```

It is also recommended to import the numpy numerical processing library on which opencv is based, so that you can manipulate OpenCV abstractions using the functions in numpy. Usually for convenience it is imported as "np".

```
import numpy as np
```

Functions in OpenCV are then called with `cv2.<function name>()` and functions in numpy are then called using the “np” namespace with `np.<function name>()`

Reading an image, such as the example image *gradient.png* provided can be accomplished with

```
img = cv2.imread('gradient.png')
```

Run these lines separately in a Python console.

Note: you can scroll back and forward to previously entered lines with the up and down arrows, so you do not have to re-type lines that you have already entered.

Note: ipython and most IDEs (like most modern text consoles) will auto-complete variable names and class members for you with the TAB key, for example you can type *’cv2.’* and press TAB to see a list of all the OpenCV namespace functions, classes and sub-classes.

Now enter `type(img)` to check that the image read in is of type *numpy.ndarray*. This means it is an n-dimensional array of numbers. If you enter just `img` by itself into the console, Python will print out the contents of the array. Try this now. You will see that *img* is a three-dimensional array (indicated by nested square brackets ‘[]’) that includes one dimension for RGB colour. Python tries to compress the contents of the array so that it is easier to view but still shows a few of the start and end of each matrix column/row.

Think about how the image that you read in looks. Do the numbers look like they correspond with some of the pixels of the image? You can view the image that you have read in with

```
cv2.namedWindow("Image")
cv2.imshow("Image", img)
cv2.waitKey(0)
cv2.destroyWindow("Image")
```

In this example we open a “named window” which allows us to reference it when we show an image, and then the call to cv2.waitKey(0) keeps it open until a

keypress is detected when the window is is selected/highlighted, so you can press any keyboard key after selecting the window with the mouse to end the program, at which time the window is closed with cv2.destroyWindow().

3. Capturing from the Camera

Working with still images is useful, but in most cases vision systems need to operate on constantly changing video images streamed from cameras and visual sensors. It is much more interesting to work with an image stream from the camera as you can see changes in the image immediately.

You can open a camera stream in OpenCV and then check if it is opened successfully with (note the whitespace indentation on lines under “if:”)

```
cap = cv2.VideoCapture(0)
if not cap.isOpened():
    print("Cannot open camera")
    exit()
```

Note that if you have more than one camera connected to your computer, you may need to pass a different number to *cv2.VideoCapture()*. For example, in Linux and most UNIX-like operating systems *cv2.VideoCapture(0)* will open the camera device at */dev/video0*, *cv2.VideoCapture(1)* will open the camera device at */dev/video1*, and so on. Make sure you know which enumerated device is the camera that you want.

Also, most modern cameras can capture streams of resolution 1920x1080 pixels or more. Displaying this will likely be larger than your screen. You can set the resolution to a more reasonable level such as 1280x720 by setting it in the *cap* object with

```
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)
```

You can then display a camera stream in an OpenCV window by reading the first frame with `success, img = cap.read()` and then entering a *while* loop to read successive frames, such as in

```
cv2.namedWindow("Video Stream")
success, img = cap.read()
```

```
while success and cv2.waitKey(1) == -1:
    cv2.imshow("Video Stream", img)
    success, img = cap.read()

cv2.destroyWindow("Video Stream")
cap.release()
```

The call to cv2.waitKey(1) causes a delay and then returns a non-negative value if a key is pressed while the window is selected/highlighted, so you can press any keyboard key after selecting the window with the mouse to end the program, at which time the window is closed with *cv2.destroyWindow()* and the *cv2.VideoCapture()* object is released.

Add these lines to your python program and run it with your camera attached. Change the argument to *cv2.VideoCapture()* if it does not work or is not the camera that you want. Make sure you have this working as the rest of the labs in this module will depend on it!

NOTE: Depending on your PC platform, you may experience a long start-up delay before the camera starts and you receive an image stream (up to a couple minutes!) This is due in most cases to the initialization of hardware acceleration by OpenCV. If you are using Windows, you *might* be able to shorten this delay considerably by disabling MSMF hardware acceleration with

```
import os
os.environ["OPENCV_VIDEOIO_MSMF_ENABLE_HW_TRANSFORMS"] = "0"
```

The drawback is that the frame rate may be lower and image processing operations may take longer, but if you are re-starting your program frequently a short start-up is usually preferred. (Thanks to Benjamin Allen for finding this!)

4. Thresholding an Image

The simplest form of image filtering is called Thresholding. For every pixel in an image at *(x,y)*, a threshold is applied. The most common operation is a binary selection: if the pixel value *src(x,y)* (typically for a greyscale image) is lower than the threshold it is set to *0*, otherwise it is set to a defined maximum value *maxval*.

Using your previous code, you can try thresholding either by reading in the

provided *gradient.png* image file, or by using the camera capturing live video to view a printout of the image file (and watching what thresholding does to live video scenes is also fun to try!) Since images in OpenCV are generally read in as RGB colour (in byte order, BGR, as it was historically the most popular among camera developers), you need to explicitly convert each image frame to greyscale using

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Alternatively if you are reading an image file you can read it in as greyscale using

```
img = cv2.imread('gradient.png', cv2.IMREAD_GRAYSCALE)
```

You can then threshold the image using *cv2.threshold()* with arguments of the source image, the destination image, the threshold value, and the maximum value to set thresholded pixels to. If you set the threshold at 127, which is half of the maximum of an 8 bit per pixel image, and the maximum value at 255 which is the highest possible value, you could use

```
thresh_value,thresh1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
```

In addition to the binary selection, the *cv2.threshold()* function takes several *type* values as its last argument to determining the kind of thresholding to use. These are:

- cv2.THRESH_BINARY (*maxval if src(x,y) > thresh, 0 otherwise*)
- cv2.THRESH_BINARY_INV (*maxval if src(x,y) > thresh, otherwise*)
- cv2.THRESH_TRUNC (*threshold if src(x,y) > thresh, src(x,y) otherwise*)
- cv2.THRESH_TOZERO (*src(x,y) if src(x,y) > thresh, 0 otherwise*)
- cv2.THRESH_TOZERO_INV (*0 if src(x,y) > thresh, src(x,y) otherwise*)

To see what each of the thresholds do you can open a new python source file and copy in the following code.

```
import cv2
img = cv2.imread('gradient.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
thresh_value,thresh1 = cv2.threshold(gray,127,255,cv2.THRESH_BINARY)
thresh_value,thresh2 = cv2.threshold(gray,127,255,cv2.THRESH_BINARY_INV)
```

```
thresh_value,thresh3 = cv2.threshold(gray,127,255,cv2.THRESH_TRUNC)
thresh_value,thresh4 = cv2.threshold(gray,127,255,cv2.THRESH_TOZERO)
thresh_value,thresh5 = cv2.threshold(gray,127,255,cv2.THRESH_TOZERO_INV)
titles = ['Original Image','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]
for i in range(6):
    cv2.namedWindow(titles[i])
    cv2.imshow(titles[i], images[i])
    cv2.waitKey(0)
    cv2.destroyWindow(titles[i])
```

Remember to select each window when it pops up and press a keyboard key to cycle to the next one. Try changing the threshold values and maximum values. Are the circles in *gradient.png* actually different brightnesses as they appear? And why does the thresholding sometimes mask out one or the other?

As a bonus test, what happens if you run *cv2.threshold()* on a colour image?

5. Create an OpenCV mask

For each new task, you can start with the code you already have, but you may want to create a copy of the python script first with a new name so that you have the original code that you know works for reference.

Masking is a method by which regions of an image can be “cut out”, or *segmented* on the basis of what pixel values (intensities or colours) they contain. The aim of this mask will be to identify the blue areas in an image. You can either read in the static image *“shapes.png”* or capture frames from the camera. If you are using the camera you can use a paper print-out of the shapes as a test.

To do this, you’ll need to:

- Display the image in a window using *cv2.imshow()*. Note that if you move your mouse cursor over the image it shows Red, Green, and Blue (RGB) colour component values at the bottom of the window. Move the cursor over the different colours and make a note of the RGB values for each shape or object in the scene.

- **NOTE:** Some OpenCV implementations do not show these component values using *cv2.imshow()*. A workaround for Windows is:
    - Click the 'prtscn' button on your keyboard to take a screenshot.
    - Paste the clipboard image into the MS Paint application.
    - Click on the color selector icon (the eyedropper), and then click on the color of interest to select in.
    - Then click on 'edit color' to the right of the color selection palette
- Convert the BGR image from the camera into a HSV(*hue-saturation-value*) image. You can perform the conversion and create a HSV copy of the image with

```
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
```

- Decide on the range of hues, saturations, and values that you're going to count as 'blue'. You may find the color picker, calculator and generator tool at http://colorizer.org/ helpful for this process. Note that you'll need to scale the values you find from this website: OpenCV uses integers from 0-179 for hue, and 0-255 for saturation and value.
- Create variables (suggested names `blueMin` and `blueMax`) to hold the minimum and maximum values you have chosen in the form of *(hue, saturation, value)* tuples. For example, if you've chosen a minimum hue of 20, a minimum saturation of 180 and a minimum value of 200 (*these are not good values!*) you could write:

```
blueMin = (20, 180, 200)
```

- Use OpenCV's `inRange()` function to create the mask:

```
mask = cv2.inRange(hsv, blueMin, blueMax)
```

- Modify the call to `cv2.imshow()` to display the mask instead of the source image.

Test your code. Does it behave as you expect? Does it reliably mask out only the blue regions? If not, go back and modify your HSV thresholds. It is sometimes helpful to set the *blueMin* values very low and *blueMax* values very high and then make them closer together in steps to see what gets masked out. Ask for help if you need to.

Once this is working, Add the line

```
masked_image = cv2.bitwise_and(img, img, mask=mask)
```

and change the call to `cv2.imshow()` to display the new masked image. What happens? Can you explain what the mask is doing?

What if you want the opposite masking? The mask is just a binary image, and it can be inverted quickly with

```
inverted_mask = cv2.bitwise_not(mask)
```

6. Blob Detection

Now for the fun part! You're going to use OpenCV to identify 'blobs' from the mask, and then filter them based on criteria you will choose.

You might want to have a look at this webpage to familiarize yourself with the blob detection method:

Blob Detection Using OpenCV (Python, C++)

Copy your code so far to a new file, and remove or comment out the code that generated *masked_image* from the end of the previous task (leave the part that generates the mask itself). Setting up the blob detection is very easy but requires a few more steps.

First, just after the line that creates the mask, create the parameters object for the blob detection:

```
params = cv2.SimpleBlobDetector_Params()
```

The object you have just created contains a number of parameters, which all have default values. The full set of filter parameters is given in Table 1.

Table 1: Blob detector parameters

Parameter Name	Default	Description
thresholdStep	10	These parameters control the *thresholding* process. OpenCV takes a greyscale image and slices it into several black-and-white images by looking to see whether each pixel is above or below a threshold value, and then uses the other parameters to control blob detection. Because our mask image is already black-and-white, we will be disabling this feature.
minThreshold	50	
maxThreshold	220	
minRepeatability	2	
minDistBetweenBlobs	10	
filterByColor	true	If filterByColor is true, blobs will only be detected if their (greyscale) colour matches the value of blobColor. Valid values are 0 or 255 (black or white).
blobColor	0	
filterByArea	true	If filterByArea is true, blobs will be rejected if their area (measured in number of pixels) is outside the range minArea to maxArea.
minArea	25	
maxArea	5000	
filterByCircularity	false	If filterByCircularity is true, blobs will be rejected if their circularity does not lie within the range minCircularity to maxCircularity. The circularity of a blob is given by $\frac{4\pi A}{P}$ where A is the area of the blob and P is its perimeter. A circle has a circularity of 1, and everything else is less than this.
minCircularity	0.8	
maxCircularity	infinity	
filterByInertia	true	Inertia ratio is a measure of how the diameter of a blob changes with angle. An straight line has an inertia ratio of zero, and a circle has an inertia ratio of 1. If filterByInertia is true, blobs will be rejected if their inertia ratio lies outside of the range minInertiaRatio to maxInertiaRatio.
minInertiaRatio	0.1	
maxInertiaRatio	infinity	
filterByConvexity	true	As you saw in the lecture, the convexity of a shape is the proportion of its convex hull that is occupied by the shape itself. Any convex shape has a convexity of 1. If filterByConvexity is true, blobs will be rejected if their convexity is not in the range minConvexity to maxConvexity.
minConvexity	0.95	
maxConvexity	infinity	

To get an idea of how the various parameters affect detection of objects, look at the provided image *BlobTest.png*. It shows how typical shapes can be detected based on total area, thresholds, circularity, inertia, and convexity.

For the purposes of this task:

- Disable the thresholding by setting

  ```
  params.thresholdStep = 255
  params.minRepeatability = 1
  ```

- Our mask has white areas where the colours of interest are, so add this line to make the blob detector look for white blobs:

  ```
  params.blobColor = 255
  ```

- For now, disable all the other filters by setting the other '`filterBy...`' parameters to '`False`'.
- To use the blob detector, add the lines

  ```
  detector = cv2.SimpleBlobDetector_create(params)
  keypoints = detector.detect(mask)
  ```

This will generate a set of keypoints which contain information about the location and size of the detected blobs. When you use machine vision for automation you could use the information in the keypoints directly, but for now we're going to plot the keypoints onto the image so you can see them displayed.

OpenCV has a function called `drawKeypoints()` which does exactly this. We will use drawKeypoints in several activities that require identifying keypoints in images. For this example you can use

```
imagekp = cv2.drawKeypoints(mask, keypoints, None)
```

- Add code to display this image, then save and test your code. What does it do?
- We can add a colour and extra flags to the *drawKeypoints()*function. Try adding them additional function arguments, following *"None, "* as in

```
imagekp = cv2.drawKeypoints(mask, keypoints, None,
color=(0,0,255),
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

- You'll notice that it's not just the five blue shapes that can be detected. Any region of blue in the image, no matter how small or what shape it is, is detected.

OPTIONAL ACTIVITY 1: Colour Filtering

To achieve reliable detection of specific shapes in the example image *shapes.png*, more filtering is needed.

Modify your filter by changing the other parameters to reliably find

1. The blue circle
2. The blue rectangle
3. The red star

Note that these are three separate tasks and will need three different bits of code. You can copy your code file three times and edit the copies if you'd like to keep your code for each.

You will need to work out appropriate values for the filters for yourself. You may or may not need every filter every time. If you are having trouble finding the correct colour values to use, you can have a look at this forum post link which has a colour map and some guidance (remember, forums can be very useful!)

OPTIONAL ACTIVITY 2: ArUco Tags

OpenCV has a large library of built-in packages that can do many other image processing tasks beyond blob detection. One particularly useful library for robotics research is ArUco, which efficiently detects the locations of square marker tags within an image. Several different 'dictionaries' can be used, and are easily generated using this part of the OpenCV library.

To use ArUco tags, you first need to generate the dictionary. Different sizes of ArUco tags are available, and the larger they are, the more different tags can be generated but the slower the detection look-up is (there are up to 50 4x4 tags, and up to 250 6x6 tags). A 6x6 tag dictionary and (optionally) parameters that can be modified can be created with

```
aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_6X6_250)
```

```
aruco_parameters = cv2.aruco.DetectorParameters()
```

You can then generate a tag with (for example) ID number 3 and image size 64x64 and save it to an image file with

```
tag_img = cv2.aruco.generateImageMarker(aruco_dict, 3, 64)
cv2.imwrite("tag3.png", tag_img)
```

This function was used to generate the first 6 entries of the dictionary `DICT_6X6_250` contained in *aruco.png* and at the end of this document. You can modify this code to generate other sizes of tags also.

To detect ArUco tags in an image such as *aruco.png* you must first convert your image to greyscale (tags are defined as black and white images) with

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

You can then detect the corners and IDs of the tags in the image (parameters argument is optional) with

```
corners, ids, rejectedImgPoints = aruco.detectMarkers(gray, aruco_dict,
parameters=parameters)
```

OpenCV also provides a convenient way to draw the detected tags into an image with

```
frame_markers = cv2.aruco.drawDetectedMarkers(img.copy(), corners, ids)
```

Use these functions to detect the tags in *aruco.png*. Can you create a program to detect and locate them all?

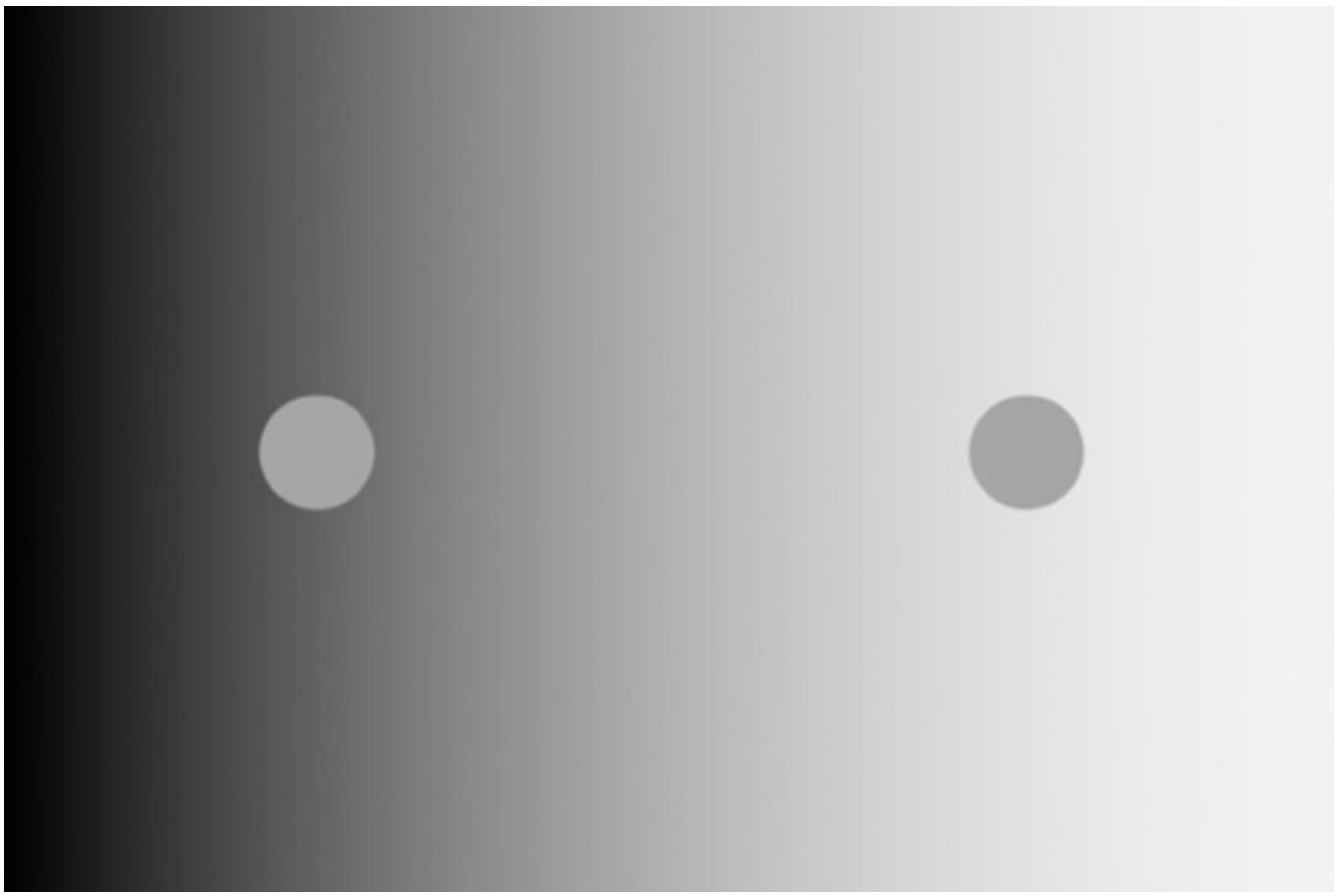

Figure 1: Test Sheet for Thresholding (gradient.png)

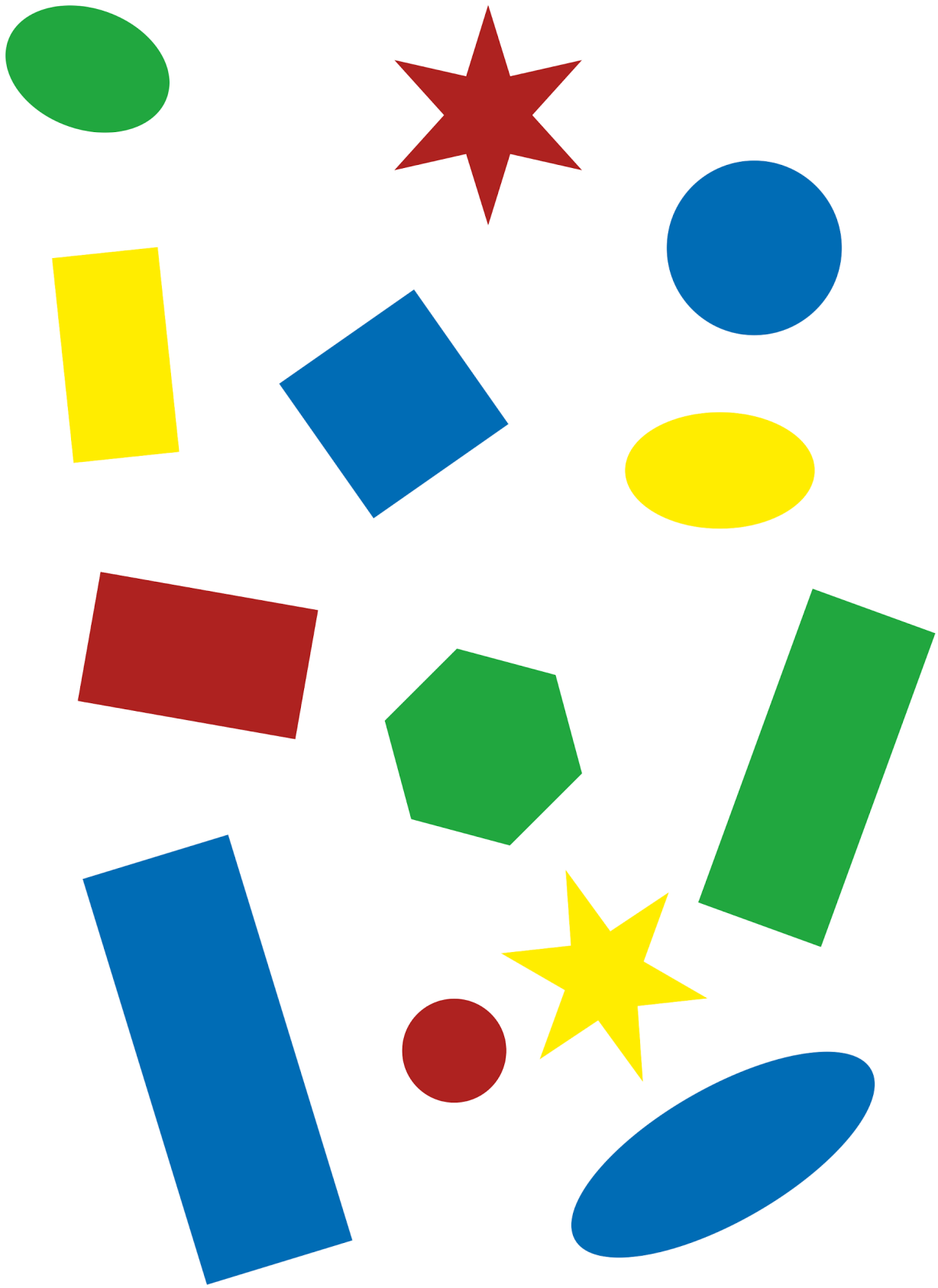

Figure 2: Test Sheet for Blob Detection (shapes.png)

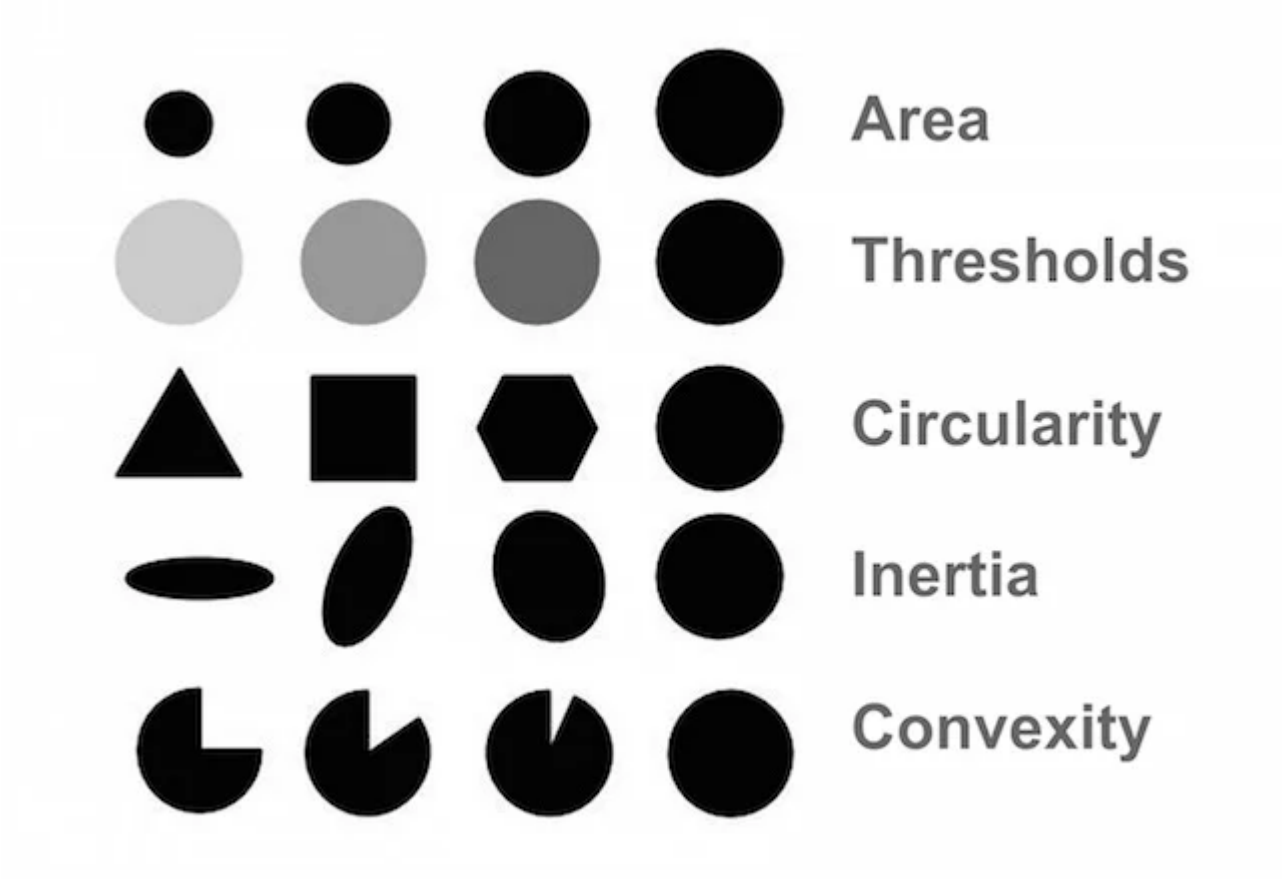

Figure 3: Example Sheet for Blob Detection Parameters (BlobTest.png)

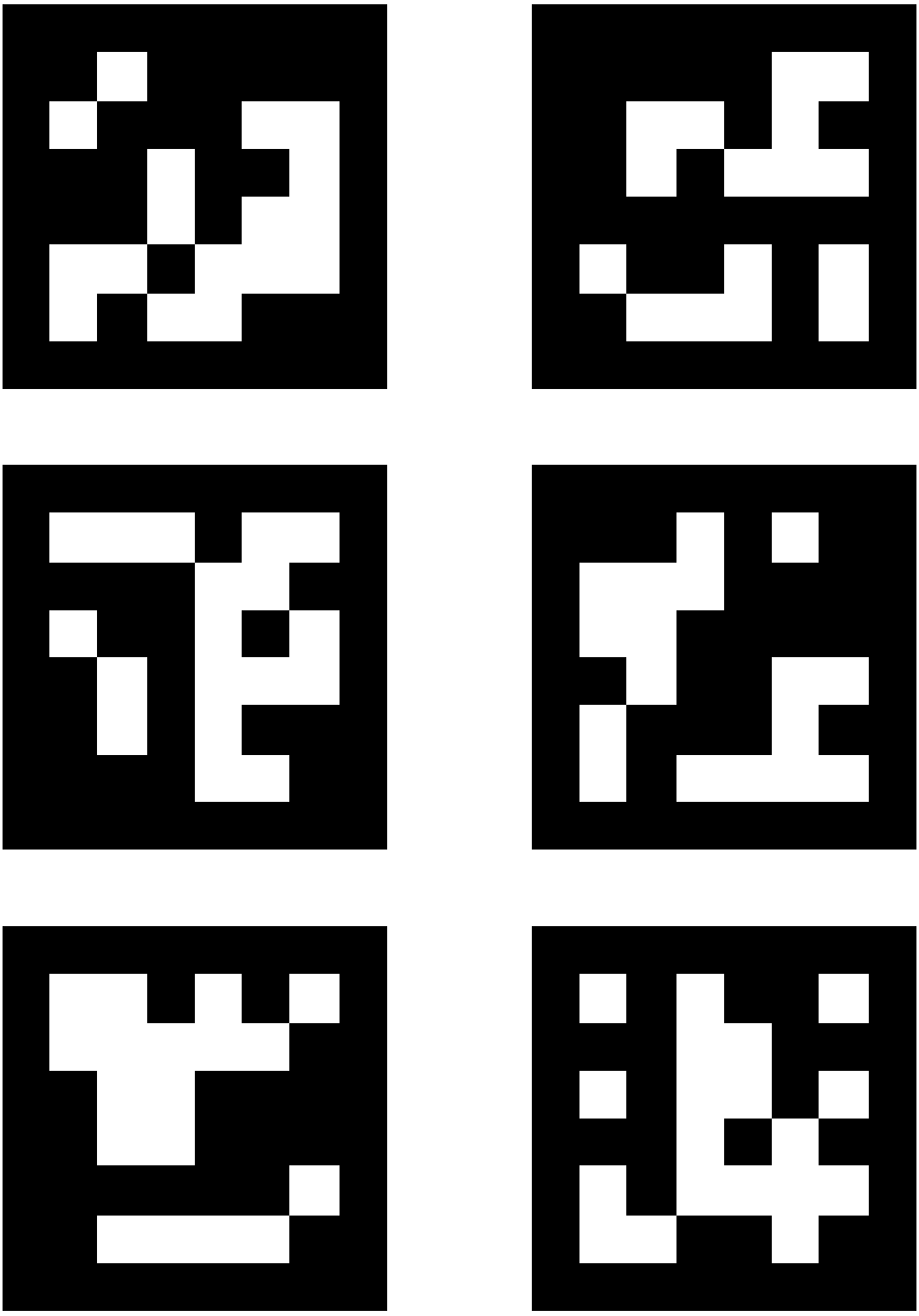

Figure 4: Markers 0 to 5 from `aruco.DICT_6X6_250` (aruco.png)