[ELE00142M] Machine Vision and Human Machine Interaction

UNIVERSITY of York

Lab 1: Image and Feature Processing with OpenCV

Part 2: Features and Identification

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*, *aruco.png*, and *Bismarck.jpg* provided on the module webpage. They are also provided at the end of this document.

Tasks

7. Gaussian Filtering

There are a huge number of filtering operations supported for image processing in OpenCV. Due to time constraints though we will test out only a few particularly useful operations to emphasize their usefulness.

One of the first operations that is typically used to process images is Frequency Domain Filtering. For example, the SIFT and SURF feature detectors apply a low pass filter (blur) to the image before detecting gradients within features to make detection more consistent. Blurring an image removes rapid changes and random pixel pattern noise (high frequencies) and retains slow, consistent variations in intensity (low frequencies) that are generally characteristic of features we want to recognize.

Two-dimensional images can be mathematically frequency filtered in the same way as one-dimensional data series (where instead of one dimension of time, you integrate/multiply along two dimensions of space). Since convolution in the spatial domain is equivalent to multiplication in the frequency domain, we don't have to convert our entire image into a frequency domain representation (very time-consuming!). We can instead convolve our spatial-domain image with filter patterns that have the desired frequencies. This is the OpenCV approach.

Start by reading the complex image that has been provided (as greyscale):

```
img = cv2.imread('Bismarck.jpg', cv2.IMREAD_GRAYSCALE)
```

Alternately you can use *cv2.VideoCapture()* to capture video and place the filter

operations within the image frame capture loop that you created previously (it is highly recommended that you try filtering live video!)

Create a blurred image of the static image or your camera frame with

```
blur_img = cv2.GaussianBlur(img,(5,5),0)
```

If you are not using the camera you can compare the difference between the original image and the filtered image with:

```
cv2.namedWindow("Image")
while True:
    cv2.imshow("Image", img)
    if cv2.waitKey(0) == ord('q'):
        break
    cv2.imshow("Image", blur_img)
    if cv2.waitKey(0) == ord('q'):
        break
cv2.destroyWindow("Image")
```

With this user-friendly loop, press any key to change images, then press the *'q'* key (with CapsLock off) to quit. Can you see the difference that blurring causes by cycling between them? What has happened to the fine details and lines?

What you have done is you have convolved your image intensity (pixel) data with a low-frequency Gaussian function. The second argument to *cv2.GaussianBlur()* is the kernel size *x=5* and *y=5*. The third argument is the kernel standard deviation *σx* (which is automatically set with *σy* for convenience if 0 is used). *Kernel* is just a term for a mathematical function that quantifies a difference between two data points, or in other words, transforms one piece of data into another. Plotting a Gaussian kernel produces the familiar "bell curve" shape in one dimension, or produces a cone shape in two dimensions such as in Figure 1.

Now, try changing the kernel size from *(5,5)* to other values such as *(9,9)* or *(3,11)* (note that they must be odd integers!). What is the difference that the change in kernel size causes?

Convolution "spreads out" each pixel in the image to distribute its intensity

among its neighbors with a distribution given by the kernel. If you want to customize the kernel that is used to create the Gaussian blur, you can use the *cv2.getGaussianKernel()* function, but usually it is not necessary. Note that this convolution is relatively time-consuming. The SURF feature detector uses a similar but faster box filter that is implemented in the *cv2.boxFilter()* function (but has slightly different arguments). Try it out with

```
cv2.boxFilter(img, -1, (5,5))
```

Can you see any difference between the box filter and Gaussian filter?

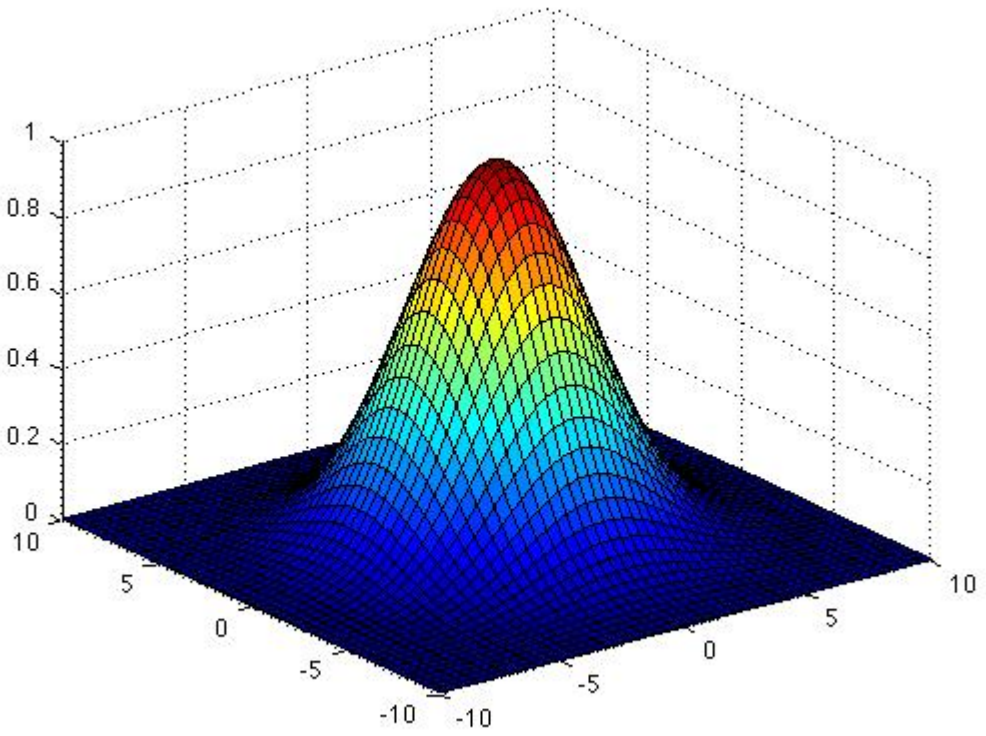

Figure 1: A Gaussian Kernel Function

8. Frequency Filtering

You may be wondering “why do we want to make otherwise good-looking images blurry?” The answer is that by changing the convolution kernel in the spatial domain, we can change their frequency content in the frequency domain!

A blurry image effectively has had its high frequencies removed by the

“spreading” effect of convolution. To get the high frequencies instead, all we have to do is subtract the low-frequency image from the original image (with all frequencies). Try this out with

```
high_img = img - blur_img
```

Now add `cv2.imshow("Image", high_img)` to your viewing loop (with another delaying call to *cv2.waitKey(0)*) or in your camera frame capture loop. Cycle through the images and try to explain the new image that you see. What has been removed from the image with this simple subtraction of intensities? What is left? Why is there so much noise in the otherwise-smooth areas of the original image?

Try to change the Gaussian kernel size again and observe what happens to *high_img*. Can you improve the “edges” of the image, or make them more precise?

9. Edge Detection

There are many applications in which we would like to detect the “outlines” of objects clearly. Most of them use frequency filtering in one form or another, but obviously a lot of extra work is necessary to make “clean” edges. This is where algorithms such as the Canny Edge Detector come in, implemented using *cv2.Canny()*. The algorithm is indeed quite “*canny*” at finding clean edges in images but is far from new, having been first developed by John F. Canny in 1986 (whose name was *uncannily* appropriate for the algorithm!). The steps are the algorithm are as follows:

- **Noise Reduction** with a 5x5 Gaussian filter, as above
- The **Intensity Gradient** of the Image is found by filtering with a Sobel kernel (a 2-D spatial gradient measurement) to get an estimate of the first derivative (slope) of the image at every point in *x* and *y* directions. This is another form of high-pass filtering and produces similar results.
- **Non-maximum Suppression** is applied by checking if every pixel is a local maximum (peak) in the direction of its gradient, if not it is set to 0.
- **Hysteresis Thresholding** amplifies edges by discarding any gradient values below *minVal* and retaining gradient values above *maxVal*.

Gradients between these thresholds are retained only if they are adjacent (connected) to edges above *maxVal*.

Try out this edge detection algorithm with

```
edges = cv2.Canny(img,100,200)
```

add `cv2.imshow("Image", edges)` to your viewing loop or your camera frame capture loop. How have the extra steps made it different from the raw high-pass filtered image?

The second and third arguments to cv2.Canny() are the thresholds *minVal=100* and *maxVal=200* respectively. Try changing these thresholds and note which edges disappear or reappear first - are they the weakest/strongest edges in the image?

10. FAST Feature Detection

Image “Features” are identifiable fragments of information that are unique within an image or to the object in the image, and that can be used to characterize the image or object. The simplest “features” are often simply groups of pixels that describe edges or corners with consistent inter-pixel intensity gradients. The goal of feature extraction and description is to create a data “footprint” that can be identified and tracked between multiple images of a common object. The features located on an image with a feature extractor are called “keypoints”.

The Features from Accelerated Segment Test (FAST) feature detector was designed to be exactly that - *FAST!* It allows every pixel in an image to be checked to see if it is a corner (i.e. with adjacent pixels having an asymmetric intensity gradient distribution, indicating an edge with a bend) quickly enough for real-time applications such as stereo vision, visual SLAM, and machine learning. The process that is used by the FAST detection algorithm is:

- Select a pixel *p* in the image with intensity *I* and select an appropriate threshold value *t.*
- Consider a circle of 16 pixels around the pixel under test as in Figure 2.
- *p* is a corner if a set of n contiguous pixels in the circle which are all brighter than *I*+*t*, or all darker than *I*−*t*. Here, *n* was chosen to be 12.

Detecting multiple interest points in adjacent locations is another problem, but it

is solved in the algorithm by using Non-maximum Suppression as follows:

- Compute a score function, *V* for all the detected feature points. *V* is the sum of absolute difference between p and 16 surrounding pixels values.
- Consider two adjacent keypoints and compute their *V* values.
- Discard the one with lower *V* value.

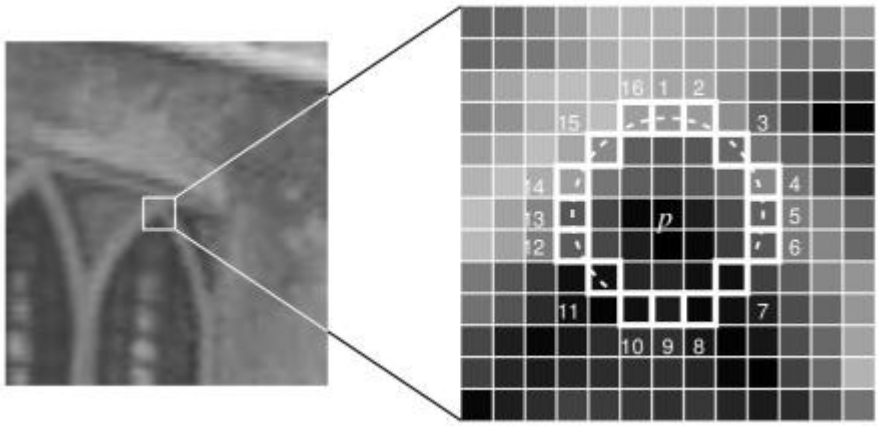

Figure 2: FAST Feature Detection

A FAST detector is created in OpenCV using the *cv2.FastFeatureDetector_create()* function

```
fast = cv2.FastFeatureDetector_create()
```

Feature detectors in OpenCV have a common base class and a common interface to make them programmably interchangeable. To run the detector on an image which outputs a python *tuple* of all the corner keypoints, use

```
kp = fast.detect(img, None)
```

You can see how many keypoints were detected with `len(kp)` and index a particular keypoint such as keypoint 10 with `kp[10]`. Each one is of type *cv2.KeyPoint* and contains methods and attributes such as `kp[10].size` which stores the size of keypoint 10 and `kp[10].pt` which stores its actual location on the image in pixels. Try entering these commands now - it is not unusual for a complex image to have thousands of keypoints so look at a few.

It is generally useful to see where the keypoints are located in an image, and

OpenCV provides a function *cv2.drawKeypoints()* that annotates an image with its list of keypoints. Use it on *img* and its list of keypoints *kp* with

```
kp_img = cv2.drawKeypoints(img, kp, None, color=(255,0,0))
```

Note that if you omit the *colour* argument (a tuple in B,G,R order) OpenCV uses random colors for each keypoint - pretty but not very useful. If plotting more than one set of keypoints you can draw them in different colours using successive calls to cv2.drawKeypoints().

Display the image with keypoints by adding cv2.imshow("Image", kp_img) to your viewing loop. Look at the keypoints detected. Do they define the shapes of the scene well?

Next, try disabling Non-maximal Suppression with

```
fast.setNonmaxSuppression(0)
```

Detect the keypoints and view the image again, do you think the keypoints are more useful in this case, or less useful?

11. ORB Feature Descriptor

While FAST is good at finding features in an image, what if we want to match them with similar features in another image? This is a problem, as a lot of keypoints are detected as you have seen, but there is no characteristic “footprint” by which to compare them to keypoints in another image.

This is where the Binary Robust Independent Elementary Features (BRIEF) descriptor comes in. BRIEF focuses on compact and efficient representation of features, by taking a smoothed image patch and selecting a set of n (x,y) location pairs. Then, for each location pair p and q it stores a binary 1 if intensities $I(p)<I(q)$, or else a 0. This is applied for all the n location pairs to get a n-dimensional bitstring, which can be compared with other descriptors by Hamming distance (the number of bit positions in which bits are different).

However, BRIEF does not find keypoints, it just describes them (ORB uses FAST for that). Also, BRIEF is not good for matching keypoints that have been rotated (e.g. if the camera rotates), so ORB "steers" BRIEF according to the orientation of n binary tests with a $2\times n$ matrix S which contains the coordinates

of the pixels. Then the rotation matrix for orientation angle θ is used to rotate S to a rotated version Sθ, discretizing the angle to increments of $2\pi/30$ (12 degrees). Hence the name of ORB: “Oriented fast and Rotated Brief”.

There are a few other details as well: ORB prunes FAST features to the most recognizable set using a response measure similar to the Harris corner detector, and creates a scale pyramid by downsampling the image, and then re-detecting and storing features at lower resolution so that features can be matched even if the image is re-scaled. This comprehensive set of capabilities have made ORB the most popular feature detector in most applications, with performance comparable to the (patented) SIFT algorithm and generally superior to the (patented) SURF algorithm. An additional advantage is that ORB uses bitstrings that can be implemented in fast fixed-point arithmetic while most others require floating-point calculations.

As ORB was developed by members of the OpenCV community specifically to be a free OpenCV feature descriptor extractor, it has excellent support. You can create an ORB detector with

```
orb = cv2.ORB_create()
```

Feature detection is done the same way as for FAST, with the detector method

```
kp = orb.detect(img, None)
```

Since we are now computing a descriptor from the keypoints for later feature matching, an extra compute step is needed using the keypoints detected

```
kp, des = orb.compute(img, kp)
```

This also highlights the useful ability of Python functions to return more than one variable (technically they are returned as an implicit tuple). *des* is the descriptor itself, you can print its contents as we have with other data types. The keypoint locations in *kp* we can annotate to our image as we did with FAST. Try taking the annotated *kp_img* containing the annotated FAST keypoints and adding the ORB keypoints to it in green using

```
orb_img = cv2.drawKeypoints(kp_img, kp, None, color=(0,255,0), flags=0)
```

Now add `cv2.imshow("Image", orb_img)` to your display loop. What do

you notice about the ORB features compared to FAST. Is this consistent with what you know about the ORB detector? Is it more useful for recognizing specific objects in images?

12. Improving Feature Detection

You have already seen that filtering and image processing can alter an image to make it easier (or harder) to recognize features and process it using algorithms. One remaining challenge is to use everything you have learned in this lab to improve…or not…the detection of ORB keypoints (and other features, if you want to try). For example:

- Try applying a Gaussian filter and providing a low-pass filtered image to ORB feature detection.
    - Are the feature points more or less in number?
    - More or less useful for recognizing shapes?
- Try a high-pass filtered image.
    - Are there applications where the resulting keypoints might be particularly useful?
- Try thresholding the image at an appropriate level with one or more thresholding algorithms
    - How do the features differ from the Gaussian-filtered images?
- Try other images or camera scenes to see which ones work best with feature detection…

Once you are finished with the activities of this lab, please ask a demonstrator to check your work to receive credit that will count towards your final grade.

OPTIONAL ACTIVITY 3: Spectrum Viewing

Are you wondering what all those filtered images look like in the frequency domain? Using OpenCV's *cv2.dft()* function and the fast Fourier transform functions in the *np.fft* numpy library, you can decompose a greyscale image into a two-dimensional frequency distribution. To view the spectrum you will need to

use the matplotlib library.

Note: if you do not have matplotlib installed you can install it by opening a shell within your Python environment e.g. Anaconda if you are using it, and installing it with `pip3 install matplotlib` or similar pip command.

Import matplotlib with

```
from matplotlib import pyplot as plt
```

Make sure you start with a greyscale image by reading it with *cv2.IMREAD_GRAYSCALE* as an argument or converting a camera image with *cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)*. You can get the complex Discrete Fourier Transform of the image with

```
dft = cv2.dft(np.float32(img),flags = cv2.DFT_COMPLEX_OUTPUT)
```

The first channel *dft[:,:,0]* will have the real part of the result and the second channel *dft[:,:,1]* will have the imaginary part of the result. (look at the size with *dft.shape* and note it is the same number of pixels as your image *img.shape* but with two colour channels).

Initially the zero frequency component (DC component) will be at the top left corner. If you want to center it, you need to shift the result by N/2 in both axes. This is simply done by the function *np.fft.fftshift()* as

```
dft_shift = np.fft.fftshift(dft)
```

To more clearly see the dynamic range of the frequency spectrum, you generally need to take the amplitude of the complex values and scale it logarithmically, such as

```
magnitude_spectrum =
20*np.log(cv2.magnitude(dft_shift[:,:,0],dft_shift[:,:,1]))
```

You can then plot this spectrum conveniently alongside the image using the *matplotlib.pyplot* library as follows

```
plt.subplot(321),plt.imshow(img, cmap = 'gray')
plt.title('Spatial Image'), plt.xticks([]), plt.yticks([])
plt.subplot(322),plt.imshow(magnitude_spectrum, cmap = 'gray')
```

```
plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
```

For details of the arguments to pyplot functions see the matplotlib documentation. For example you can plot more figures together using *matplotlib.pyplot.subplot()* with a three-digit numeric argument; the first digit is the number of plots along the *x* axis, the second number is the number of plots along the *y* axis, and the third is the index of the currently selected plot (in row-major order).

Try plotting the spectra of your original image, the low-pass filtered image, and the high-pass filtered image. Can you see the differences in the spectra?

If you are looking for an additional challenge, try modifying the frequency-domain spectrum and then converting it back into an image - there is a lot of help to do this in the OpenCV documentation.

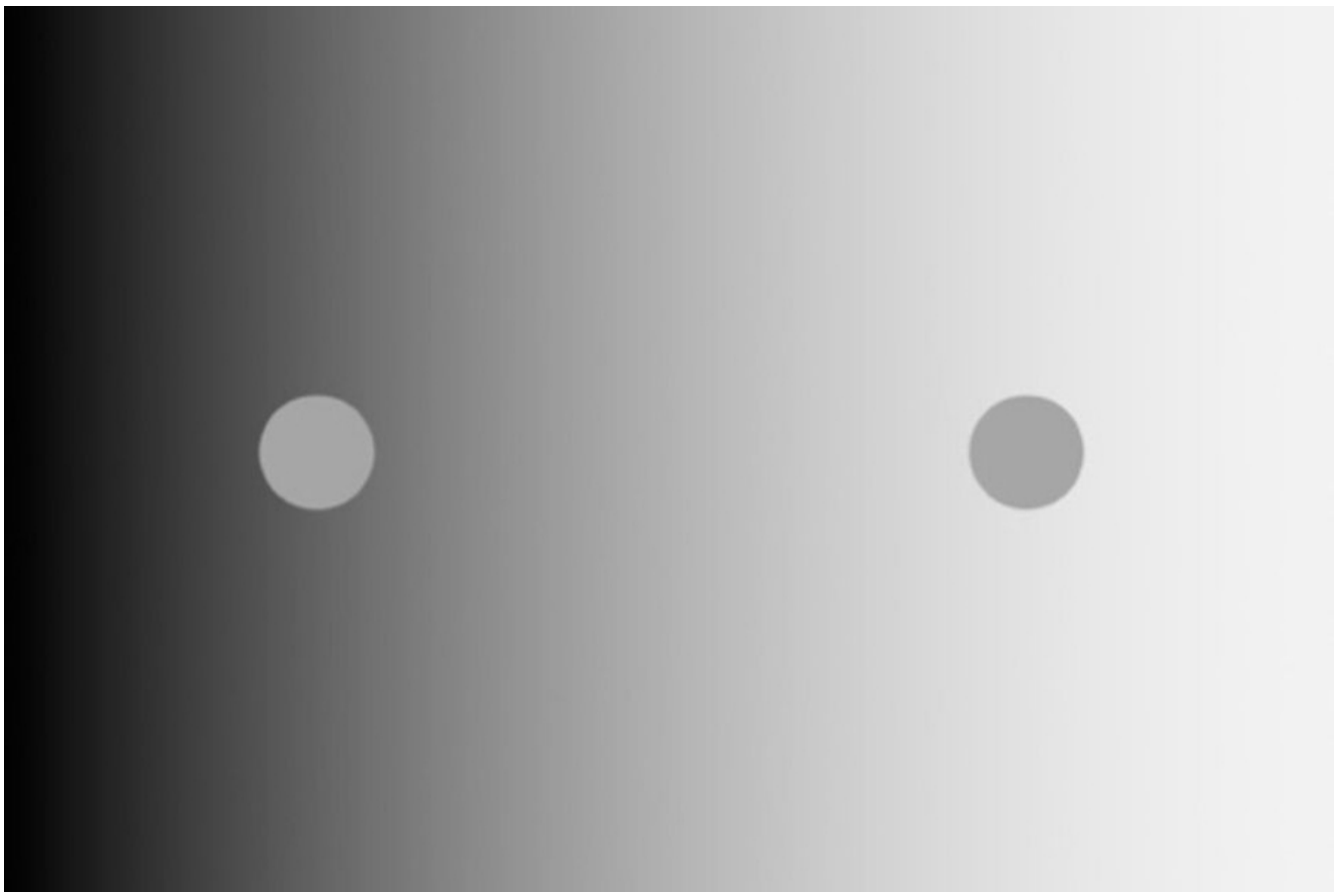

Figure 2: Test Sheet for Thresholding (gradient.png)

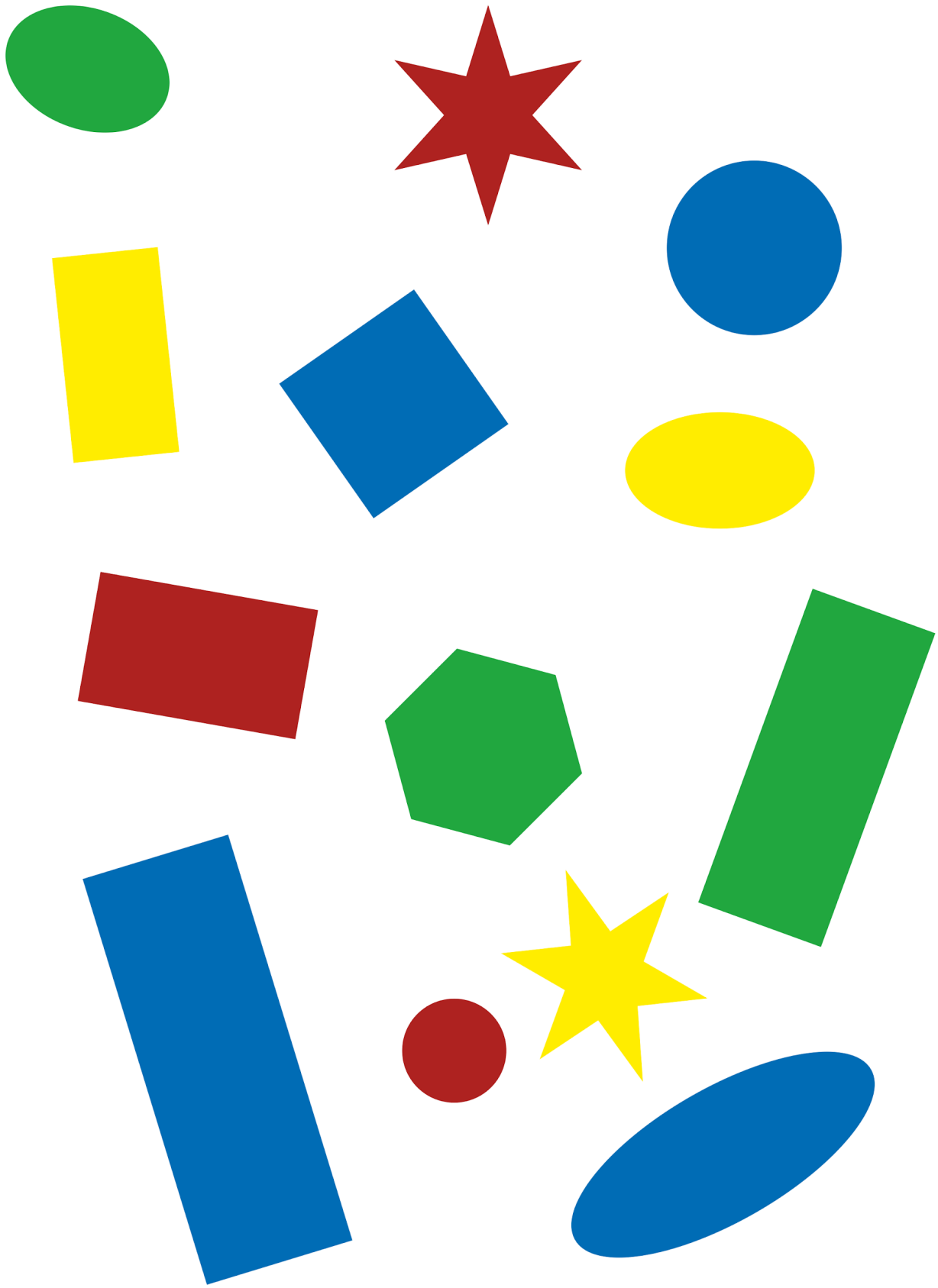

Figure 3: Test Sheet for Blob Detection (shapes.png)

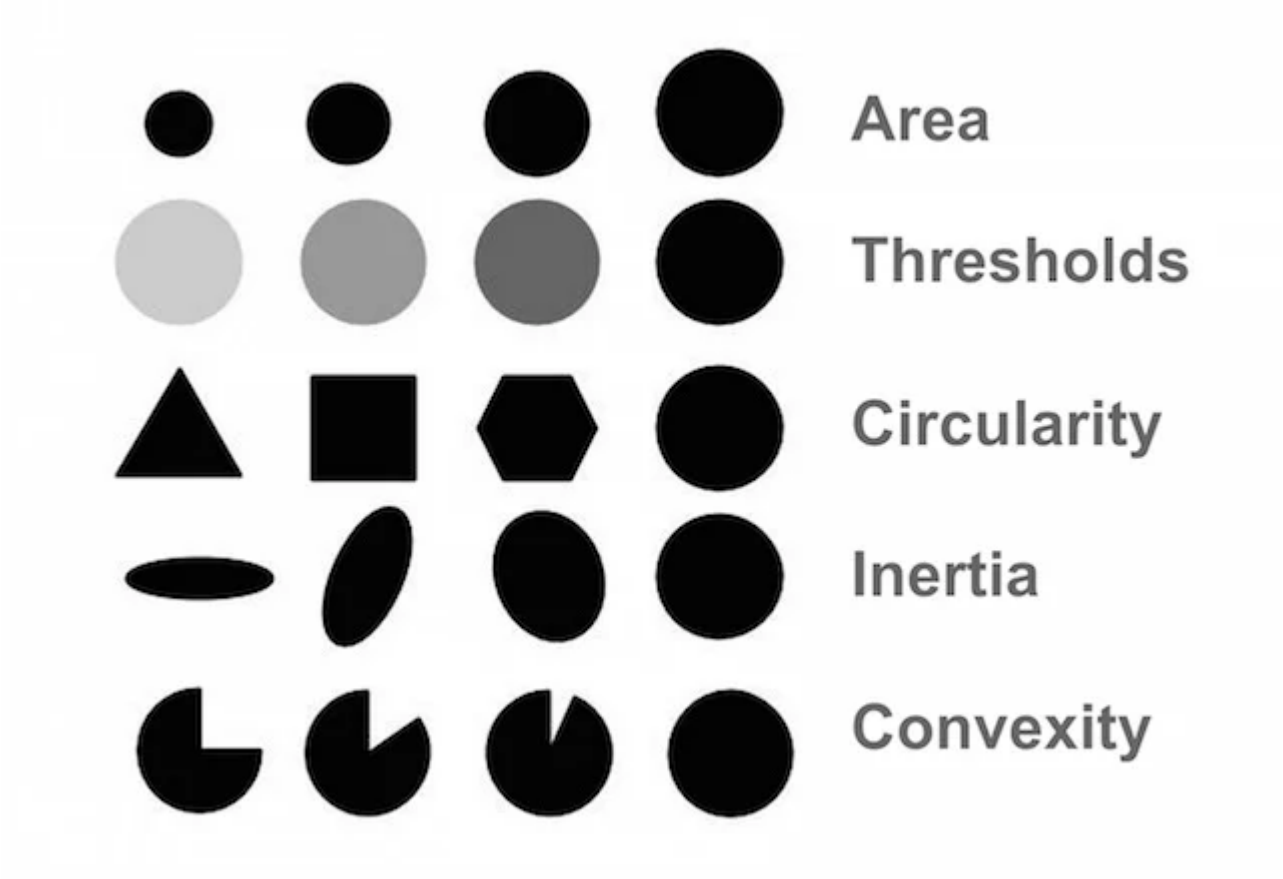

Figure 4: Example Sheet for Blob Detection Parameters (BlobTest.png)

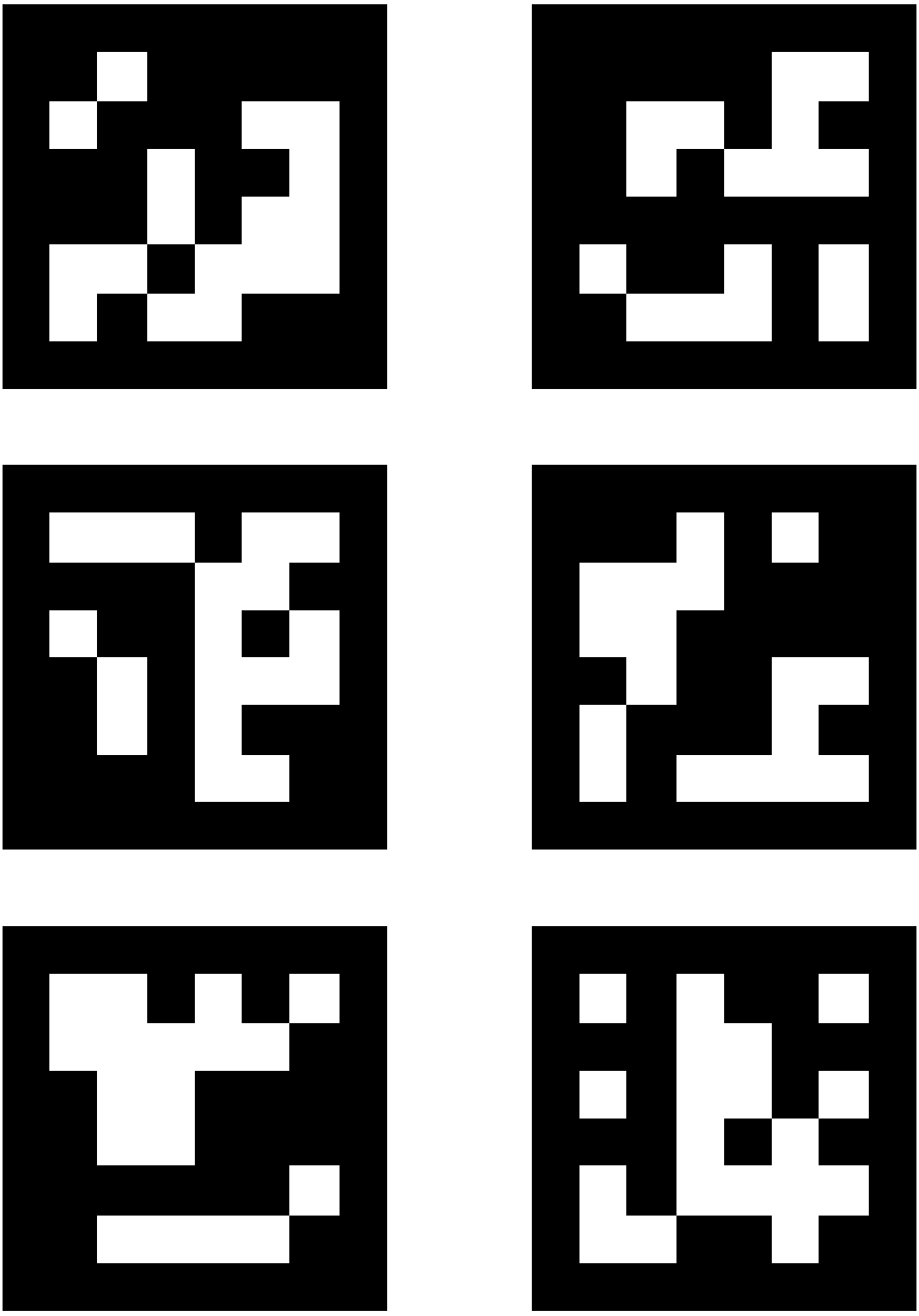

Figure 5: Markers 0 to 5 from `aruco.DICT_6X6_250` (aruco.png)

Figure 6: Complex image for filtering and feature extraction (Bismarck.jpg)