[ELE00118M] MSc Practical Robotics (PRAR)
[ELE00103M] Year 4 Robotics module

Lab Session 6 Handout

Lab 6: Robot Vision with OpenCV

Lecturer: Mark A Post (mark.post@york.ac.uk)

Technician: Mike Angus

1. Aims and Objectives

In this lab session you will be experimenting with OpenCV and Python on a Raspberry Pi 4. The Raspberry Pi is a series of widely used, low-cost SBC (single-board computer), and you will be using the Model 4 B+ is the most recent and powerful model. Each Pi has an 8MP CMOS camera attached to its CSI camera interface.

2. Learning outcomes

- Using the Raspberry Pi 4 B and the Raspbian operating system
- Opening and using the built-in camera interface on the Raspberry Pi
- Writing and running OpenCV Python code using command line tools
- Communicating with the Raspberry Pi using a desktop PC with SSH, X11 forwarding and SCP
- Use of colour detection, blob detection and tag detection in OpenCV

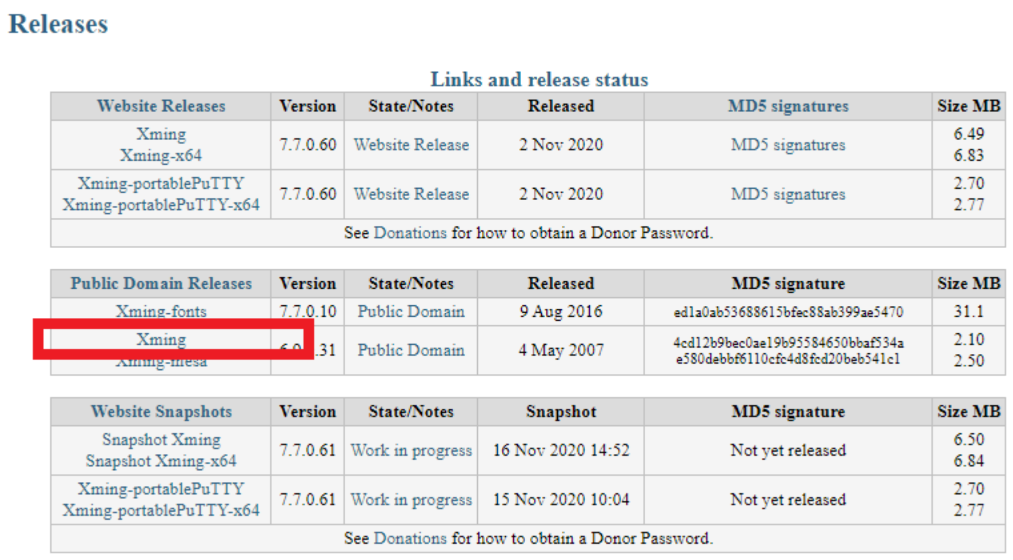
3. Software and hardware setup

Use the Raspberry Pi on your robot and make sure that the camera is connected. For viewing the camera images and processing results in real time you probably want to have a network connection working to your pi at this point.

1. Connect the USB-C power lead to the Raspberry Pi. It should begin to boot as soon as the power is connector.
2. Using the ‘Auto’ button on the bottom of the monitor, select the ‘HDMI’ input and wait for the display to change.
3. The Pi is configured to boot to command line and login automatically. The username is `pi` and the password is `raspberry.`
4. Move to the `vision_lab` folder using `cd ~/vision_lab`

You will want to see the images and camera feed that the Pi is capturing by starting graphical windows from the Pi. To do this, you will need your Raspberry Pi connected to a wireless or wired network and you have a choice of:

- Connect to the Raspberry Pi using “`ssh -X <address>`” from a native Linux PC or by running the Electronics VirtualBox VM image that is available, so that graphical programs opened on the Pi will show on your local X server.
- If you are using a Windows PC you can installed the “Public Domain Release” version of XMing from here: http://www.straightrunning.com/XmingNotes/ and as indicated below. Then have PuTTY installed, run Xming, connect the Pi to your network or to your PC with an ethernet cable, connect PuTTY with the '*enable X-11 forwarding*' box checked under Connections->SSH->X11
- Connect to the X server running on the Raspberry Pi using VNC (remote desktop). Instructions: https://magpi.raspberrypi.org/articles/vnc-raspberry-pi You will either have to enable booting to graphical desktop in the `raspi-config` utility, or start the graphical X server manually. you may need to start “vncserver” manually from the serial port or SSH console, and then in the VNC client use the address <IP address>:1 to connect - the “:1” indicates to add 1 to the port number that VNC uses to connect to display 1 (port 5900+1 = 5901).



Releases

Links and release status

Website Releases	Version	State/Notes	Released	MD5 signatures	Size MB
Xming Xming-x64	7.7.0.60	Website Release	2 Nov 2020	MD5 signatures	6.49 6.83
Xming-portablePuTTY Xming-portablePuTTY-x64	7.7.0.60	Website Release	2 Nov 2020	MD5 signatures	2.70 2.77
See Donations for how to obtain a Donor Password.					

Public Domain Releases	Version	State/Notes	Released	MD5 signature	Size MB
Xming-fonts	7.7.0.10	Public Domain	9 Aug 2016	ed1a0ab53688615bfec88ab399ae5470	31.1
Xming Xming-mesa	6.[illegible].31	Public Domain	4 May 2007	4cd12b9bec0ae19b95584650bbaf534a e580debbf6110cfc4d8fcd20beb541c1	2.10 2.50

Website Snapshots	Version	State/Notes	Snapshot	MD5 signature	Size MB
Snapshot Xming Snapshot Xming-x64	7.7.0.61	Work in progress	16 Nov 2020 14:52	Not yet released	6.50 6.84
Xming-portablePuTTY Xming-portablePuTTY-x64	7.7.0.61	Work in progress	15 Nov 2020 10:04	Not yet released	2.70 2.77
See Donations for how to obtain a Donor Password.					

4. Connect to the Pi from your PC

We can communicate with the Pi from a Linux PC using a protocol called Secure Shell, which includes the remote terminal tool `ssh` and the remote remote file copy tool `scp`. On Windows and Mac you can use the PuTTY program to connect an SSH terminal to the Raspberry Pi. On Windows you can also transfer files easily using the WinSCP program.

First we need to detect the inet (IP address) of the Pi, using the program `ifconfig`. The wifi interface wlan0 should have an inet address that corresponds to whatever network you have set it up on. If there is no inet address, you may need to set a static IP address for your raspberry pi using `raspi-config`.

Open a terminal on the workstation and connect to the Pi using the following command *(with the correct IP!)*:

```
ssh pi@xxx.xxx.xxx.xxx
```

Enter the password [`raspberry`] and should see the prompt change to reflect that you are now controlling a terminal **on the Pi**. Use `ls` and `cd` to confirm this and navigate to the folder you were working in. You should be able to edit your file in nano exactly as before, but what happens when you try to run `test_camera.py`?

If it fails with an error similar to "could not open display", you do not have X11 tunneling enabled in SSH. Most Linux programs with a GUI, including the cv2.imshow() method, use X11 windowing. We can make ssh forward the X11 data by adding a -X flag (*ssh -X pi@144.32..*) or by going to the PuTTY settings under Connection->SSH and ticking the “Enable X Forwarding” box before connecting. Enter `exit` to end the current ssh session and run again with X-forwarding enabled, then retry camera_test.py. Do you observe any lag and or other differences in performance when compared to use directly on the Pi? There is a file called `test_camera_fps.py` that calculates the approximate frame rate of the last 5 frames and displays it on the image.

5. Testing the Camera

The Raspberry Pi is equipped with a camera module. This is attached to a ribbon cable so can be moved about, but please be careful as the lens, ribbon cable and connectors are quite delicate and easily damaged. The Python code to initialise the camera and to extract and display video frames has been provided for you as test_camera.py. Type:

```
nano test_camera.py
```

to edit the file. This will open the file in the Nano text editor within the terminal window. Note that you cannot use the mouse to navigate the file. The arrow keys will take you up and down in the file, while the ‘PgUp’ and ‘PgDown’ keys will move you more quickly. The most common keyboard commands are given at the bottom of the window; note that ‘^’ refers to the ‘Ctrl’ key, so for example to leave the editor you type Ctrl-X.

Read through the code and see if you can see how it works. The important features of the code are:

- **Imports**. These lines load libraries that can then be accessed from the Python code.
- **Camera initialisation**. The next few lines enable and configure the camera, and wait for it to start up.
- **Frame grabbing**. The `camera.capture_continuous()` method returns frames from the camera so that they can be processed. Effectively it returns a list of frames, but that list grows every time a new frame is captured. The ‘for’ loop iterates over the captured frames, and will run forever unless interrupted.
- **Display**. Displaying the frame is one of the few parts of this code to use the OpenCV library. There are other ways to display an image from Python, but using OpenCV will make the remaining tasks simpler.
- **List truncation**. It is very likely that it will take longer to process and display a frame of video than it does to capture it. If you are processing frames more slowly than the camera is delivering them, the list of frames awaiting processing will just

keep growing forever, and the processing will lag further and further behind the capture. Truncating the list discards the unused frames and ensures that the next frame that is captured is used straightaway.

- **Key capture**. It's useful to be able to exit from the program, so key capture is used to detect if the '**q**' key has been pressed, and to exit the 'for' loop if so. The OpenCV library is used for this.

Leave the editor by pressing Ctrl-X.

Type

```
python test_camera.py
```

to run the program.

What do you see? Use the test sheet provided to check that the colours in the image are approximately correct. Remember you can press **Ctrl-C** to exit the program at any time when you've finished testing it or if a problem occurs.

You will likely need to rotate the image 180 degrees if the camera on your robot is upside-down. You can specify the camera rotation when being initialized, in Python this is done after instantiating in the code:

```
camera = picamera.PiCamera()
```

right after this put:

```
camera.rotation = 180
```

The image from the camera will now be rotated 180 degrees. If you want to rotate the image later in your code you can also use the OpenCV function:

```
cv2.rotate(img, cv2.ROTATE_180)
```

6. Transfer files to and from the Pi with SCP

It is up to you whether you do the remainder of this lab directly on the Pi, or using ssh with X11 forwarding on the workstation.

We can use the program `scp` in a similar way we would copy a file locally using `cp`. As you are currently have all your files on the Raspberry Pi, you may want to copy them to your workstation using:

```
scp pi@144.32.70.xxx:/home/pi/filename filename
```

As it is sometimes hard to find the file on the Pi's filesystem, you can also use '*' wildcards to copy all the files that include certain text in their filenames, e.g.:

```
scp pi@144.32.70.xxx:/home/pi/*part-of-filename* filename
```

You can then copy files back easily after editing. To copy from the workstation to the Pi just reverse the order of arguments to `scp`:

```
scp filename pi@144.32.70.xxx:/home/pi/filename
```

Try making a copy of camera_test.py onto your network filestore.

7. Create an OpenCV mask

For this task, start with the code you already have, but create a copy by typing

```
  cp test_camera.py mask.py
```

Now edit this new file (eg. `nano mask.py`)

The aim of this mask will be to identify the blue areas on the test sheet.

To do this, you'll need to:

- Convert the BGR image from the camera into a HSV(*hue-saturation-value*) image. This line will perform the conversion and create a HSV copy of the image:

  ```
  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 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 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.

Exit the editor (*Ctrl-X*) and test your code with `python mask.py`. Does it behave as you expect? Does it reliably mask out only the blue regions? If not, go back and modify your HSV thresholds. Ask for help if you need to.

Bonus task:

Add the line

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

and change the call to `cv2.imshow()` to display the new masked image. What happens?

8. 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:

https://www.learnopencv.com/blob-detection-using-opencv-python-c/

First, copy your existing code to create a new file using

```
cp mask.py blob_detection.py
```

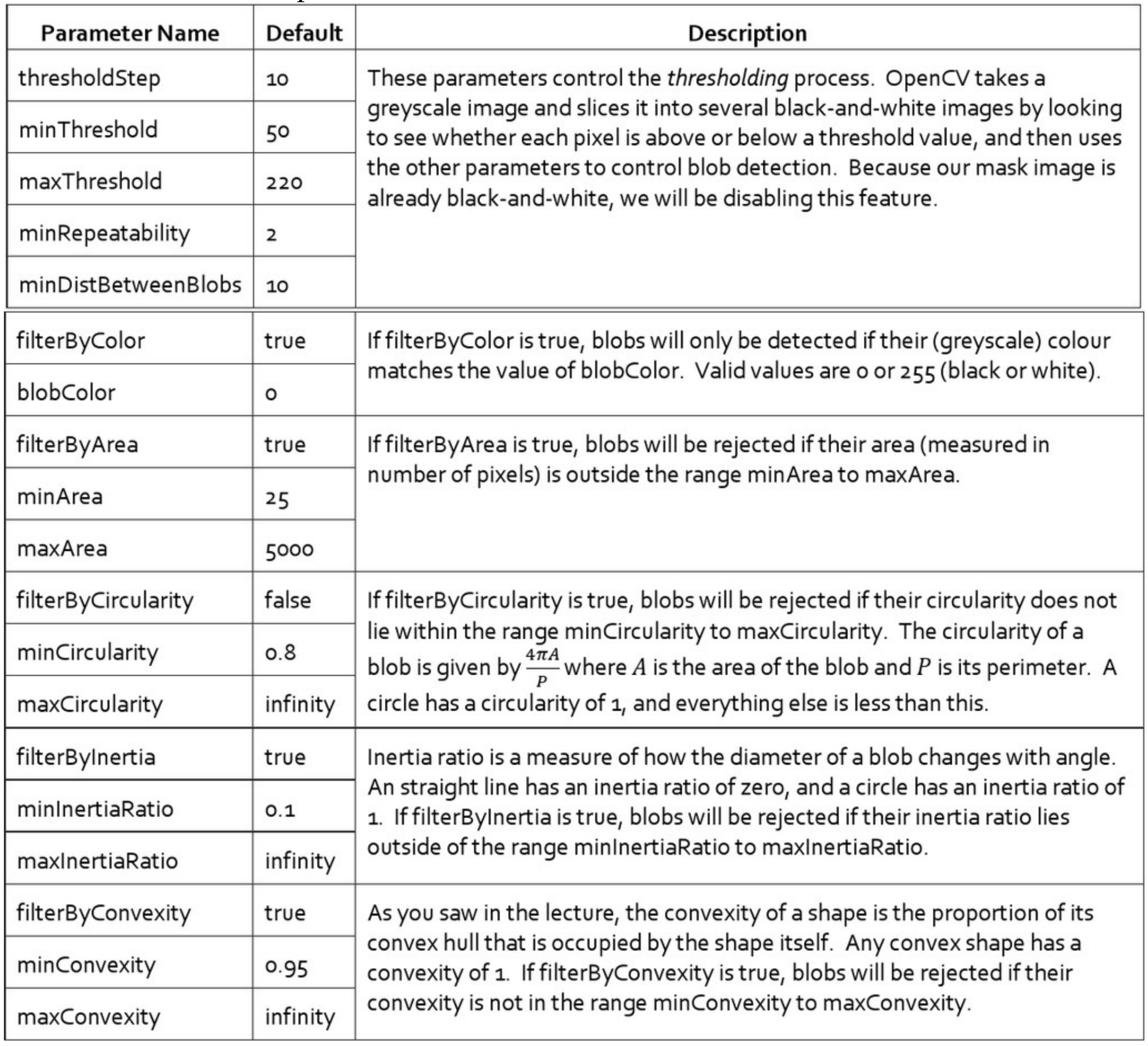
and edit blob_detection.py. Remove the code that generated the masked image from the end of the previous task (leave the part that generates the mask itself). Setting up the blob detection is very easy. You'll have to do a little bit of typing, though!

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	

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. In a robotics context 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.

  ```
  kp_image = 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 the following:

  ```
   color=(0,0,255),
   flags= cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
  ```

- You’ll notice that it’s not just the five blue symbols on the sheet that are detected. Any region of blue in the image, no matter how small or what shape it is, is detected.

These final tasks are optional to complete in the lab but demonstrate some other useful things that can be done with the OpenCV libraries.

9. Filtering

To achieve reliable detection of specific shapes, more filtering is needed.

Use the other filters 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.

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 link which has a colour map and some guidance: https://stackoverflow.com/questions/51229126/how-to-find-the-red-color-regions-using-opencv/51230291#51230291. At the end of the lab, if you want to keep a copy of your completed code to finish off later or as a reference for your final project, you can use `scp` to copy the file to the workstation.

10. 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 the library. The file `gen_aruco_pdf.py` creates a PDF containing the first 6 entries of the dictionary `DICT_6X6_250` as seen at the end of this document.

You may find it useful to use ArUco tags as markers to guide your robot. Read and then run the program `aruco_test.py` and try and detect the tags from the print out.

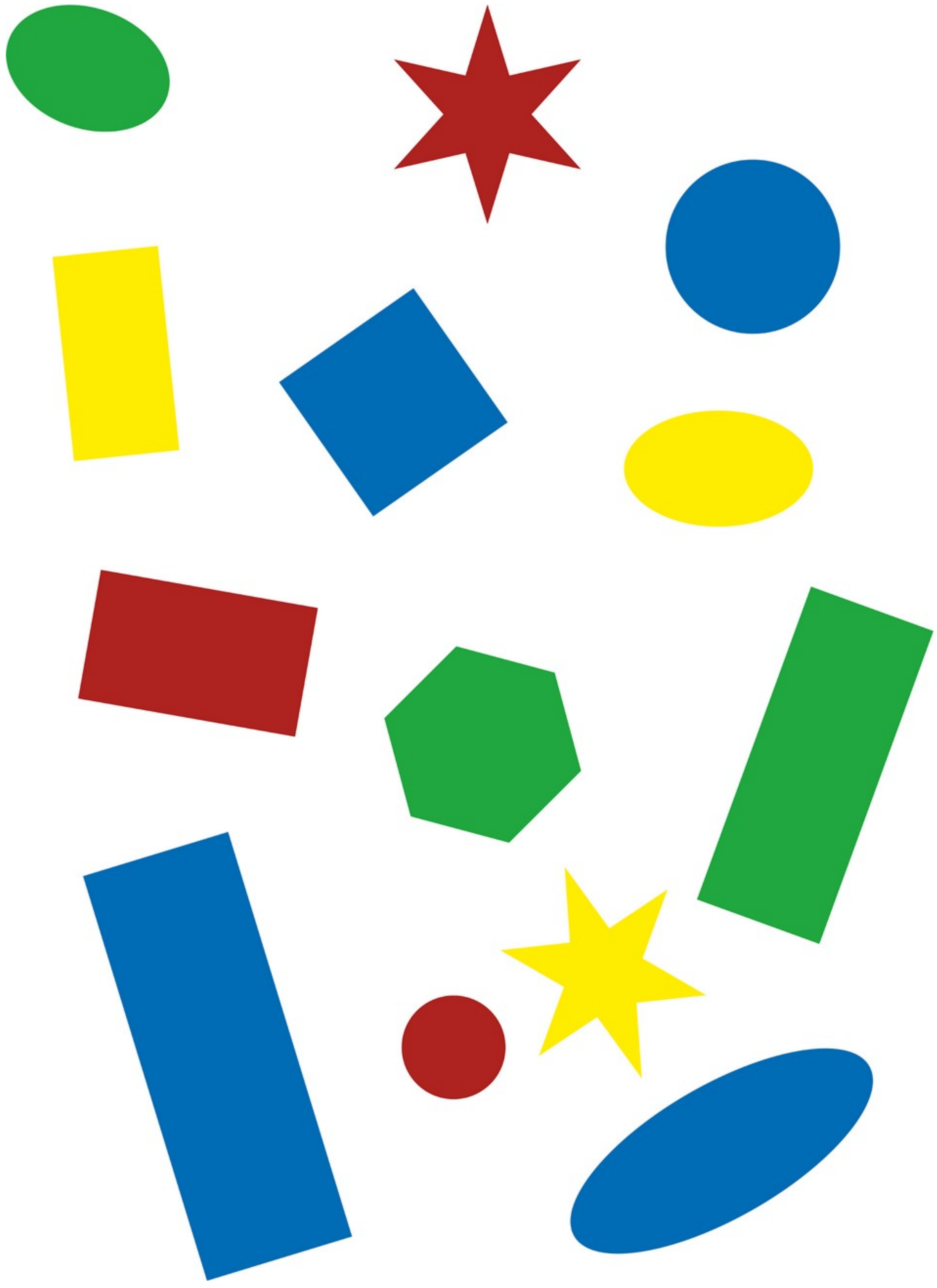


Fig 2: Test Sheet for Blob Detection

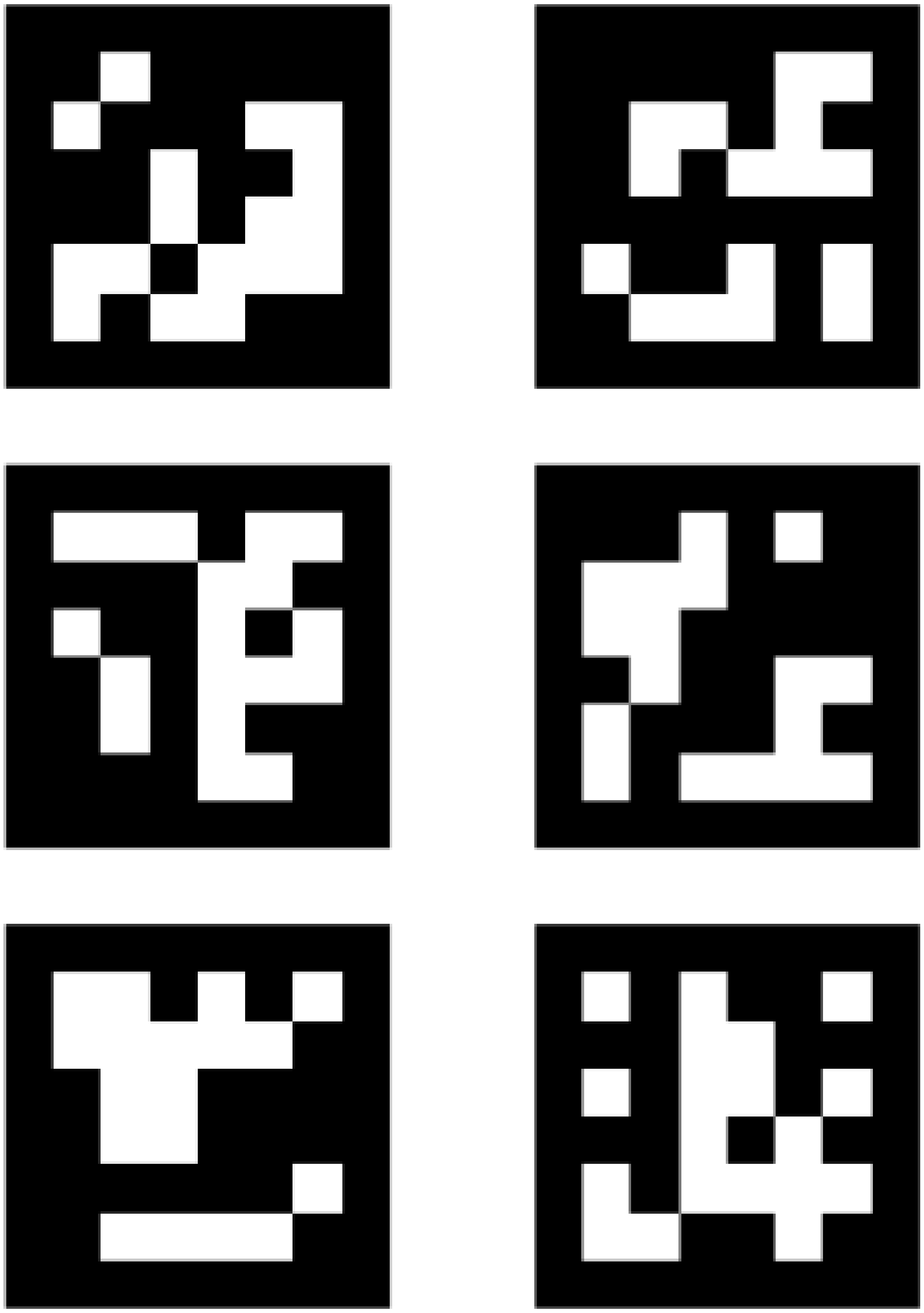


Fig 3: Markers 0 to 5 from `aruco.DICT_6X6_250`