Imports

```
import cv2
import  pandas as pd
import numpy as np
```

Load Image

I will use OpenCV to load the image into a 3D array, containing the the pixels' rows, columns, and BGR colour channels. BGR is the default colour channel used by OpenCV, so I will convert the colours to RGB.

```
def load_image(path):
    # read image
    img = cv2.imread(path)
    # convert from BGR to RGB
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    return img
```

Find Colours

Firstly, I will create a pandas dataframe containing the rgb values of the colours used in the images. This dataframe defines the rgb values of the colours so that we can later look up what colour a certain pixel of an image should be.

```
# Firstly create a dictionary of the colours
dic = {'colour': ['red', 'green', 'yellow', 'blue'], 'rgb': [[255, 0,
0], [0, 255, 0], [255, 255, 0], [0, 0, 255]]}
# convert the dictionary to a dataframe.
df = pd.DataFrame(dic)
df
```

```
   colour            rgb
0     red    [255, 0, 0]
1   green    [0, 255, 0]
2  yellow  [255, 255, 0]
3    blue    [0, 0, 255]
```

Next, we define a helper function: getColourName, which takes the RGB value of a pixel, and returns the colour as a string. The way that this function works is by calculating the Euclidean distance of the RGB value of the input pixel from the RGB values of the colours defined in our previously created dataframe. The colour that minimises the distance from the pixel value is the colour that is returned.

We will use a variable called "minimum" to store the minimum distance so far. We initialise this variable to some large number. The corresponding colour index of the final minimum value is then used to fetch the colour from the dataframe.

```
def getColourName(rgb):
    # Arbirtrary high start value
    minimum = 10000
    # Iterate over all the colours in the dataframe
    for i in range(len(df)):
        d = abs(rgb[0] - int(df.loc[i, "rgb"][0])) + abs(rgb[1] -
int(df.loc[i,"rgb"][1])) + abs(rgb[2] - int(df.loc[i,"rgb"][2]))
        if(d<=minimum):
            minimum = d
            cname = df.loc[i,"colour"]
    return cname
```

Finally, we define the find_colours function. since the images being used have the coloured blocks in the same locations in each image, we can set the coordinates of the coloured blocks, without having to manually search for them in each image. The coordinates we specify will be the top left corners of each of the images. The block sizes are also constant throughout the images, so we can search the rest of the blocks using just one range.

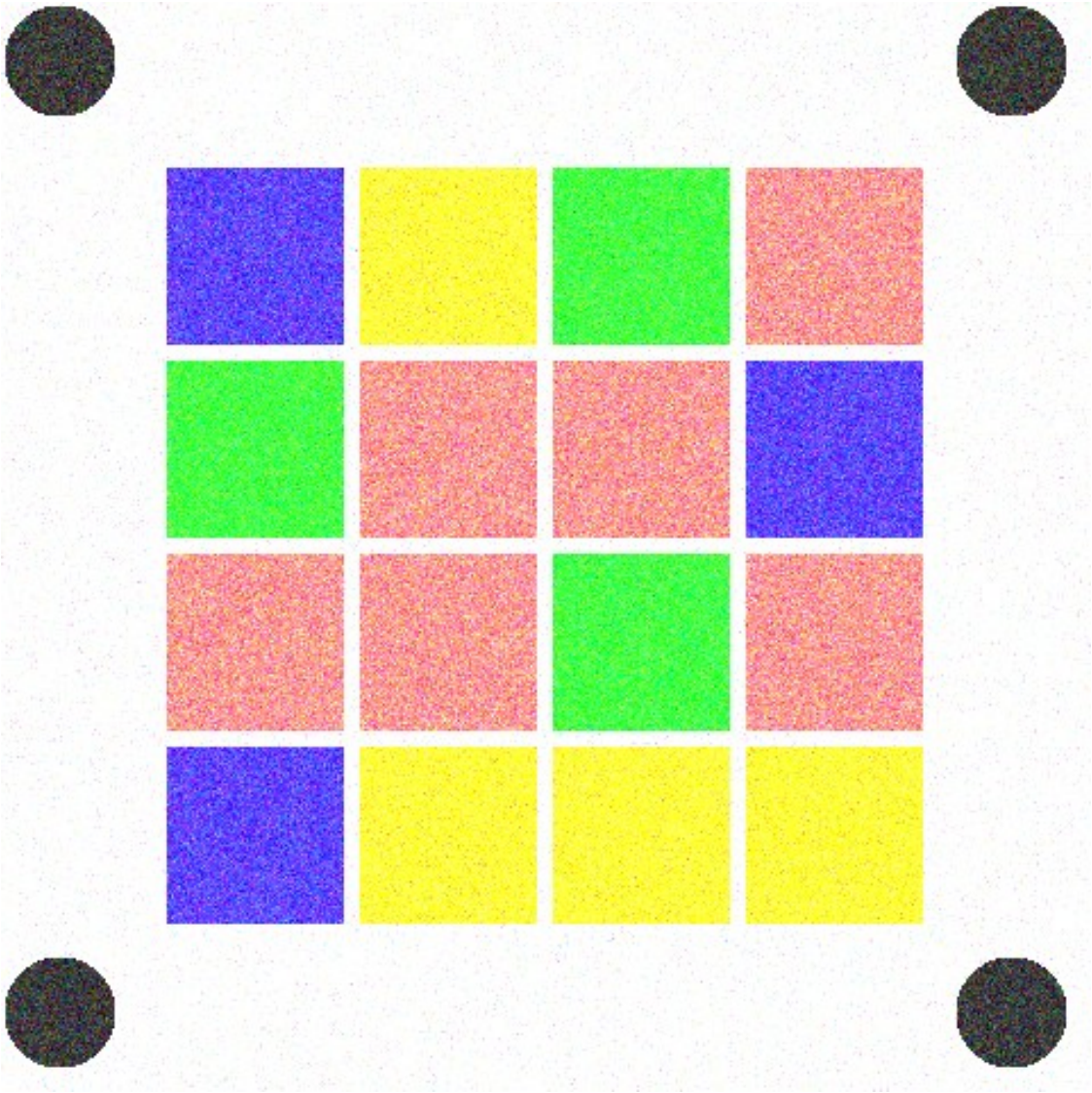
We iterate over the pixels of each of the blocks, assigning colours to each of these pixels. Finally, we get the most common colour in each block, and return that as the official block colour.

In preliminary tests of the function, I found that some blocks were being labelled as yellow when they were not. This is likely due to the noise of the images being returned as yellow (the closest colour to white), and the function then counts the noise colour as the block colour. To counter this, I added a condition that if the block colour is retunred as yellow, then check the second most common colour instead. This fixed the problem, and led to all images being correctly classified.

```
corners = (80, 170, 250, 330)

def find_colours(img):

    temp = []
    colours = []
    result = []

    for x in corners:
        for y in corners:
            for i in range(60):
                if i != 0:
                    temp.append(getColourName(img[x + i][y]))
                    temp.append(getColourName(img[x][y + i]))
                    temp.append(getColourName(img[x + i][y + i]))
                else:
                    temp.append(getColourName(img[x][y]))
```

```
            colour = max(set(temp), key=temp.count)

            if colour == 'yellow':
                no_yellow = [x for x in temp if x != 'yellow']
                if len(no_yellow) == 0:
                    colour = 'yellow'
                else:
                    colour = max(set(no_yellow), key=no_yellow.count)

            colours.append(colour)
            temp = []

    result.append(colours[:4])
    result.append(colours[4:8])
    result.append(colours[8:12])
    result.append(colours[12:16])

    return result
```

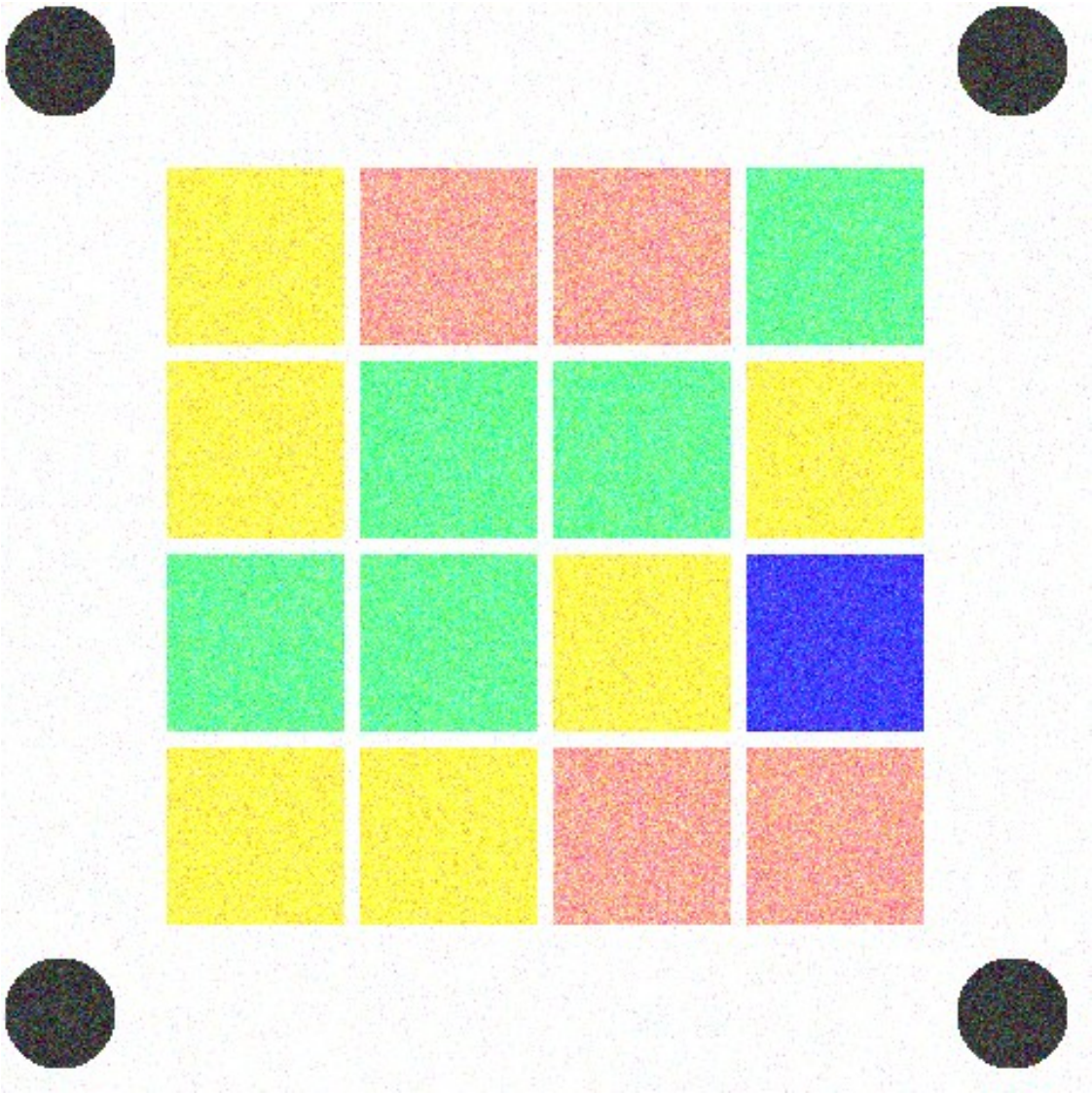
noise_1



```
img = load_image('images/noise_1.png')
print(find_colours(img))
```

```
[['blue', 'yellow', 'green', 'red'], ['green', 'red', 'red', 'blue'],
['red', 'red', 'green', 'red'], ['blue', 'yellow', 'yellow',
'yellow']]
```

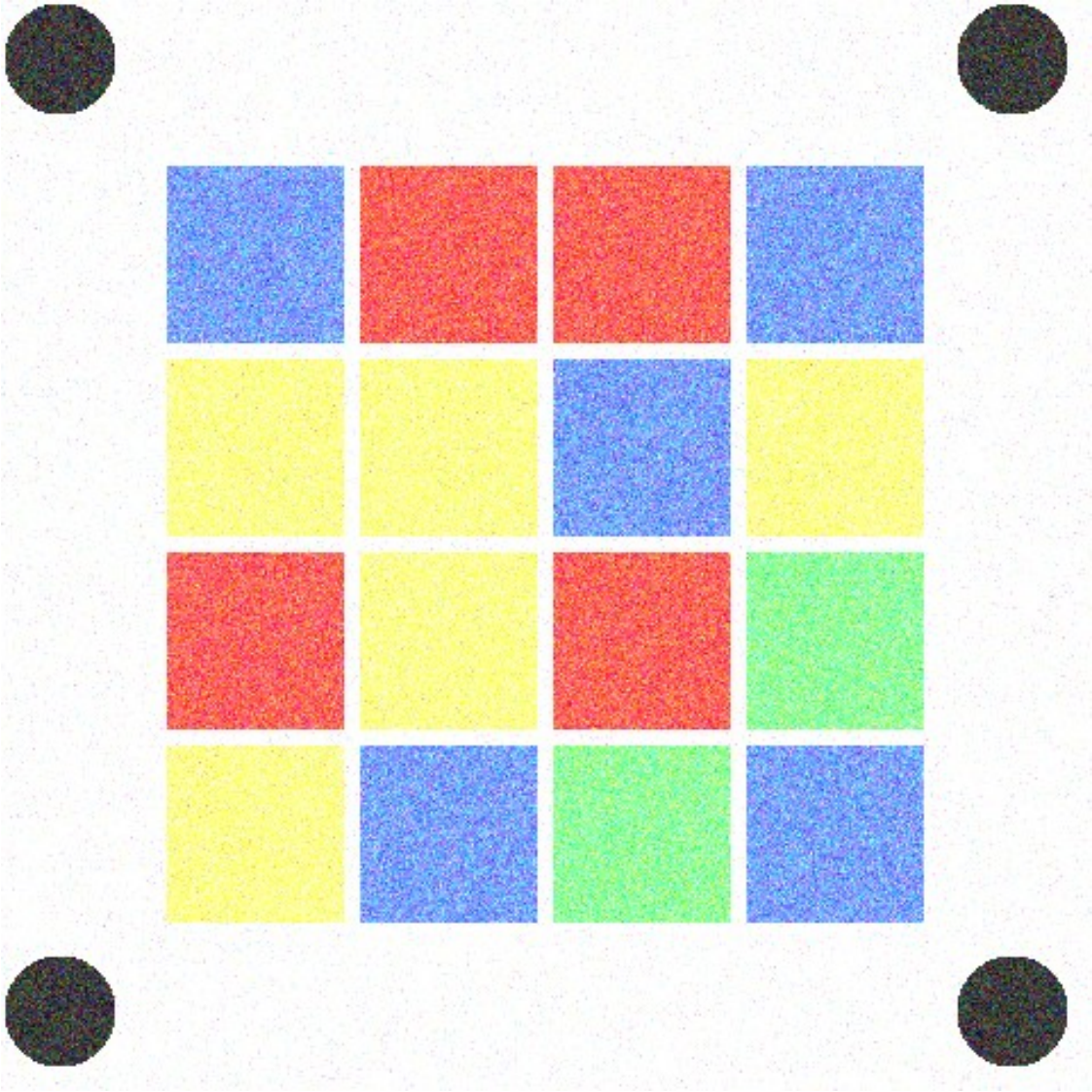
noise_2



```
img = load_image('images/noise_2.png')
print(find_colours(img))
```

```
[['yellow', 'red', 'red', 'green'], ['yellow', 'green', 'green',
'yellow'], ['green', 'green', 'yellow', 'blue'], ['yellow', 'yellow',
'red', 'red']]
```

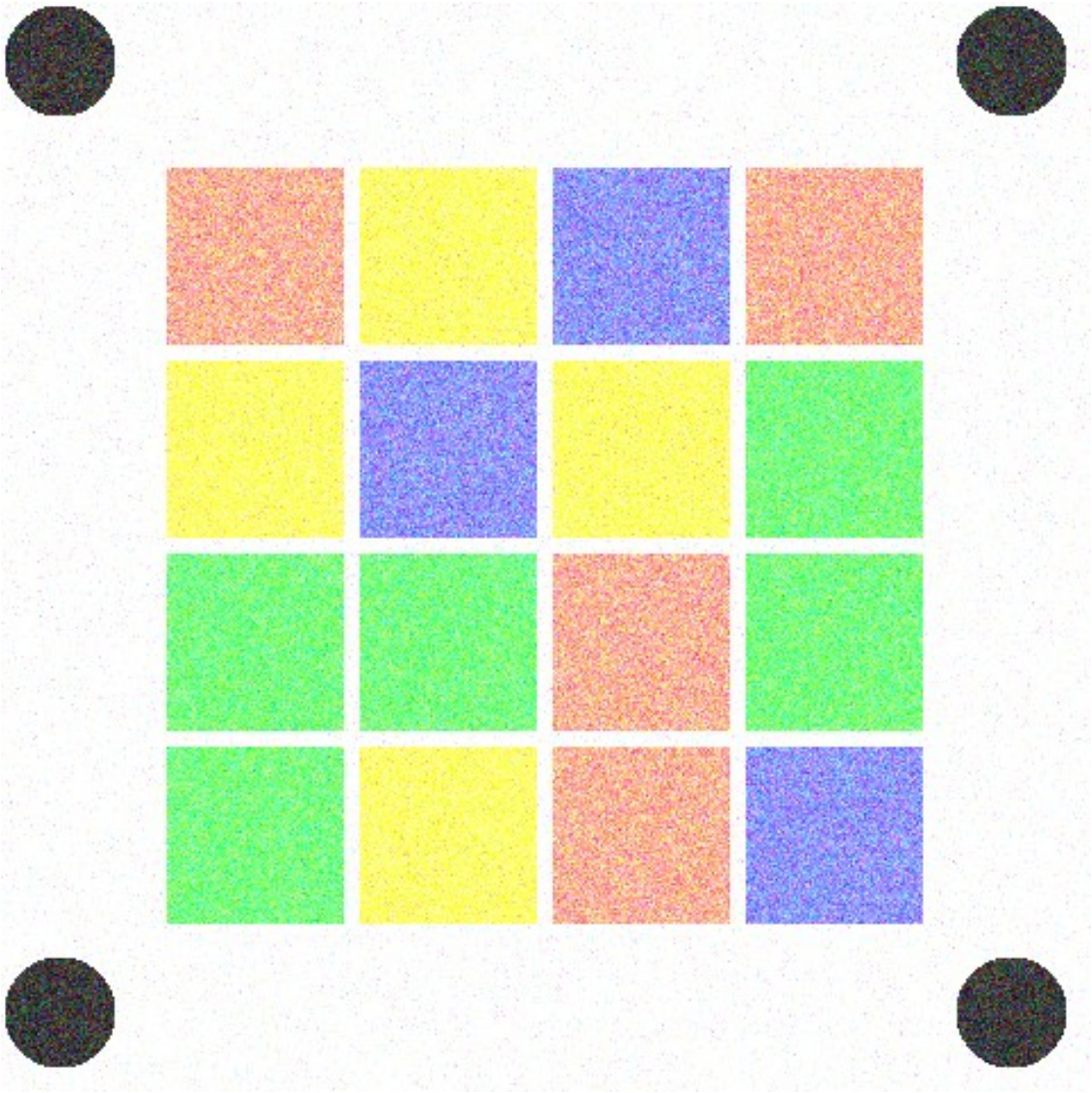
noise_3



```
img = load_image('images/noise_3.png')
print(find_colours(img))
```

```
[['blue', 'red', 'red', 'blue'], ['yellow', 'yellow', 'blue',
'yellow'], ['red', 'yellow', 'red', 'green'], ['yellow', 'blue',
'green', 'blue']]
```

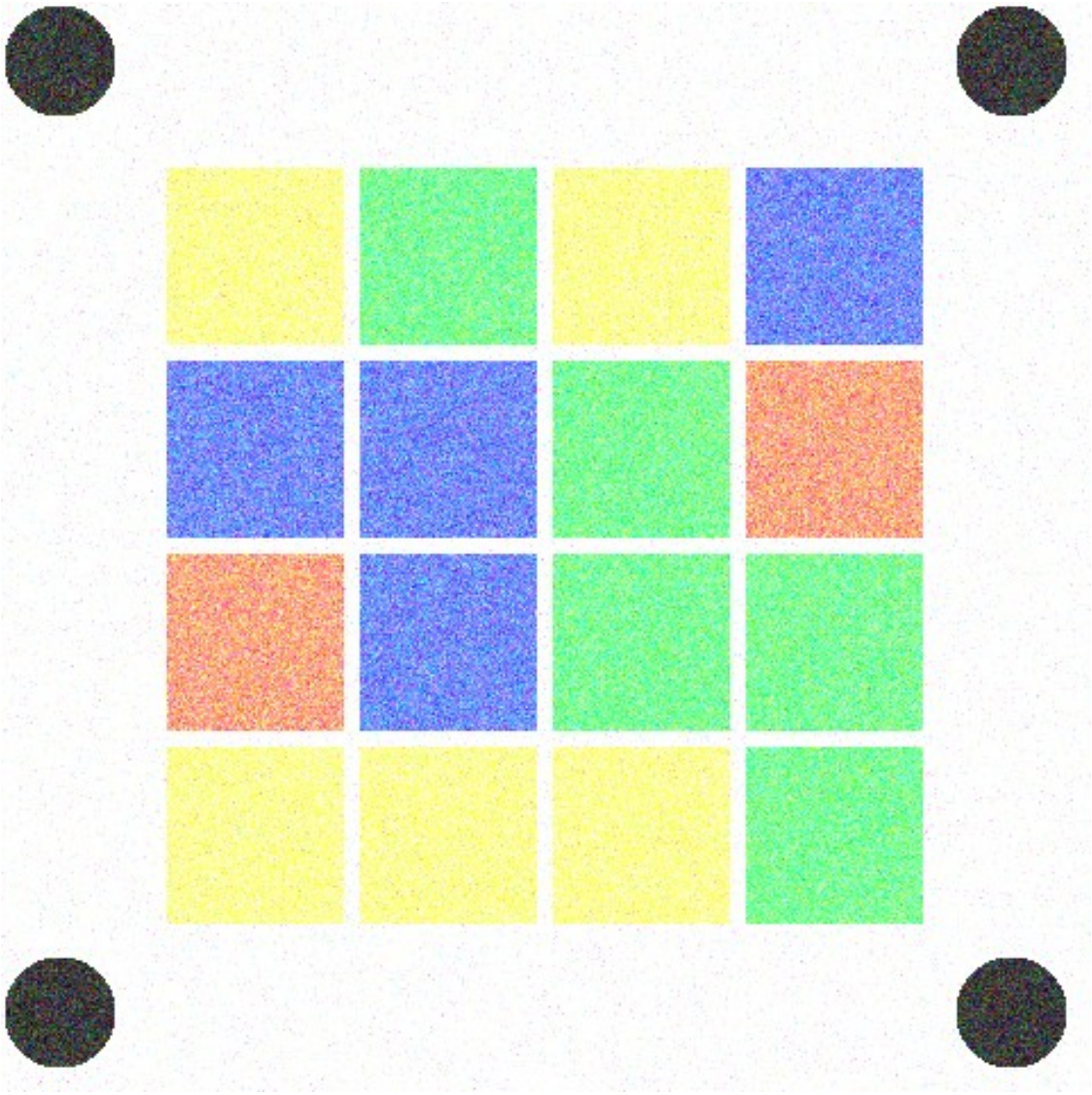
noise_4



```
img = load_image('images/noise_4.png')
print(find_colours(img))
```

```
[['red', 'yellow', 'blue', 'red'], ['yellow', 'blue', 'yellow',
'green'], ['green', 'green', 'red', 'green'], ['green', 'yellow',
'red', 'blue']]
```

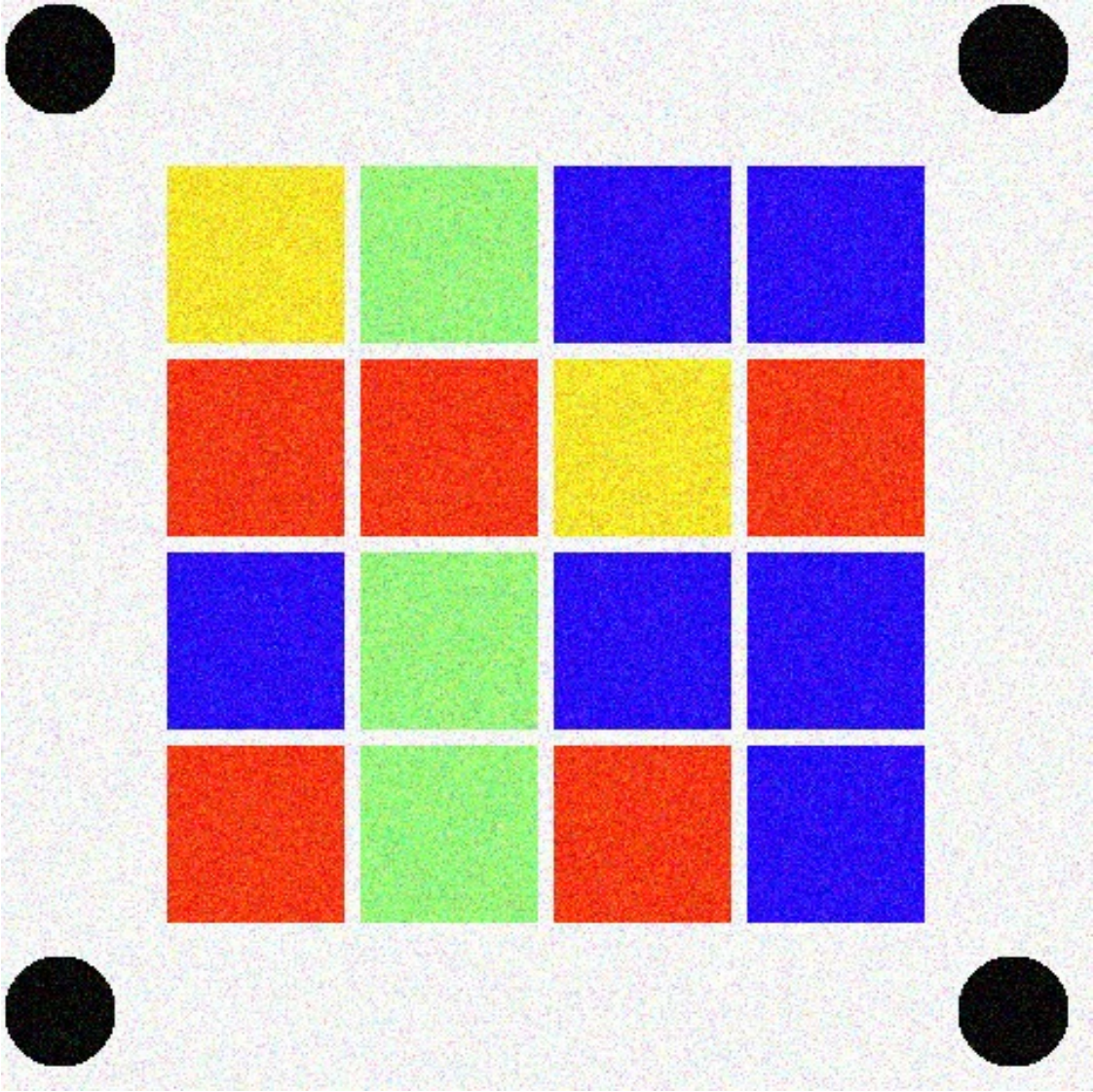
noise_5



```
img = load_image('images/noise_5.png')
print(find_colours(img))
```

```
[['yellow', 'green', 'yellow', 'blue'], ['blue', 'blue', 'green',
'red'], ['red', 'blue', 'green', 'green'], ['yellow', 'yellow',
'yellow', 'green']]
```

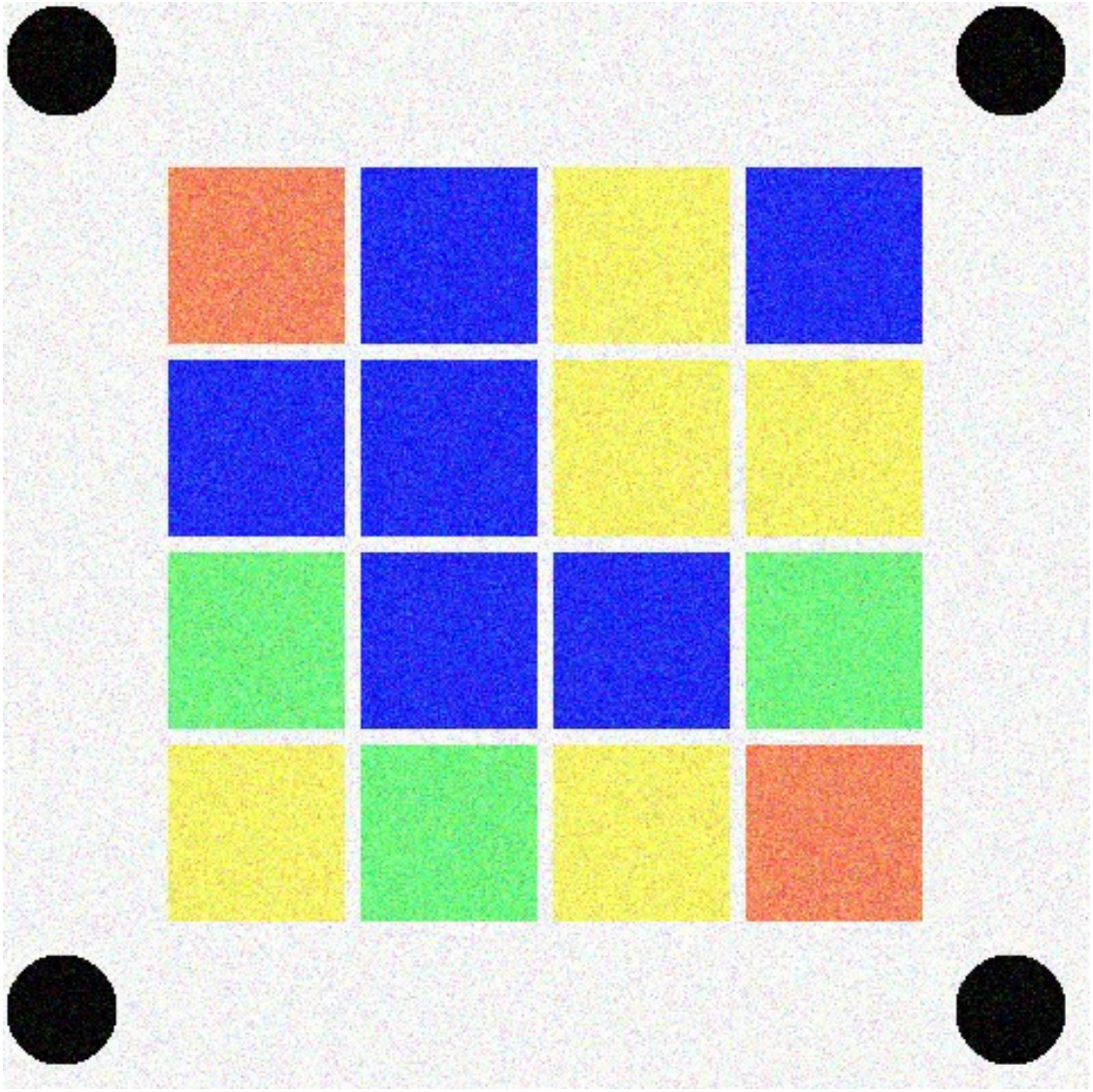
org_1



```
img = load_image('images/org_1.png')
print(find_colours(img))
```

```
[['yellow', 'green', 'blue', 'blue'], ['red', 'red', 'yellow', 'red'],
['blue', 'green', 'blue', 'blue'], ['red', 'green', 'red', 'blue']]
```

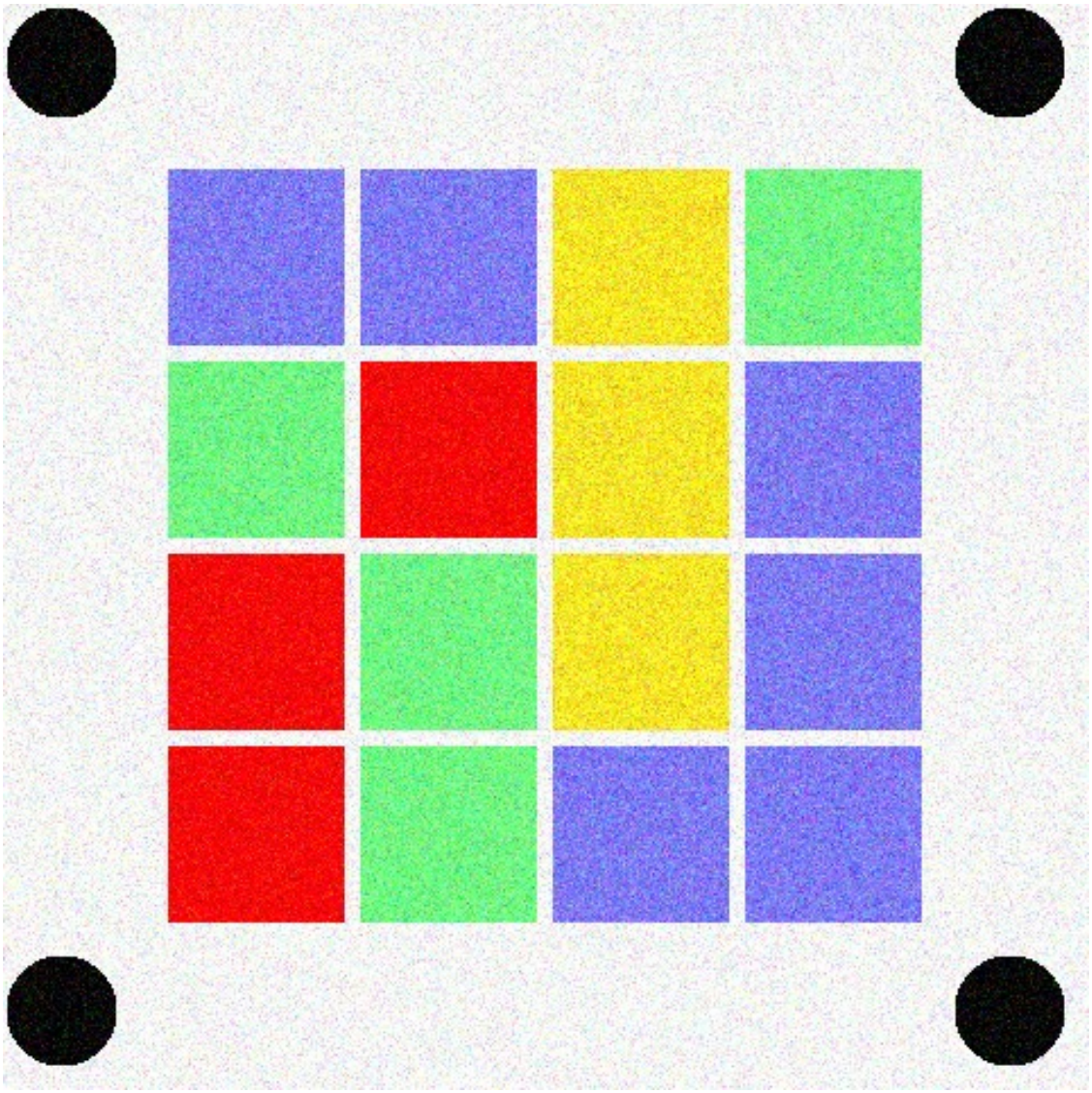
org_2



```
img = load_image('images/org_2.png')
print(find_colours(img))
```

```
[['red', 'blue', 'yellow', 'blue'], ['blue', 'blue', 'yellow',
'yellow'], ['green', 'blue', 'blue', 'green'], ['yellow', 'green',
'yellow', 'red']]
```

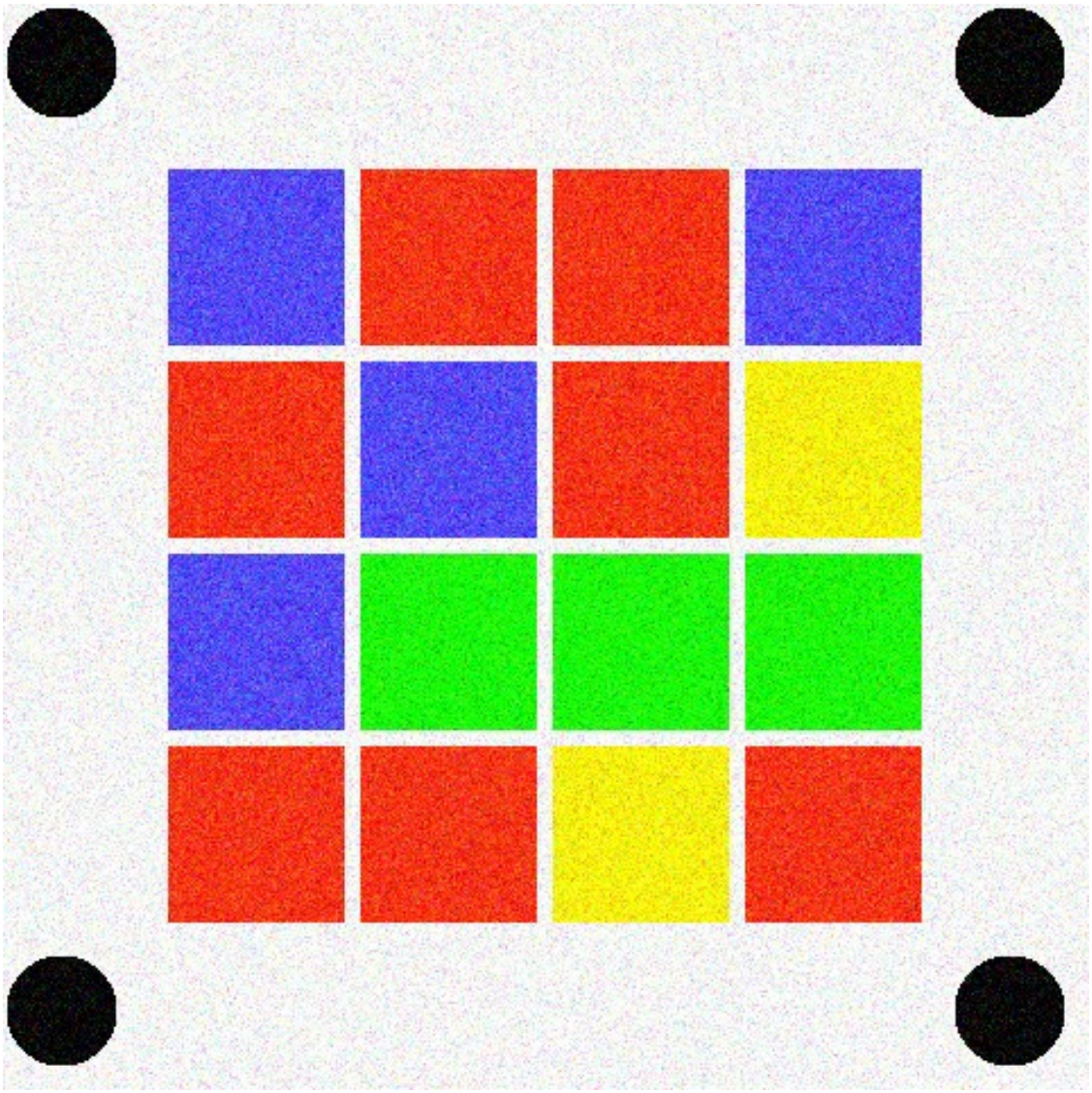
org_3



```
img = load_image('images/org_3.png')
print(find_colours(img))
```

```
[['blue', 'blue', 'yellow', 'green'], ['green', 'red', 'yellow',
'blue'], ['red', 'green', 'yellow', 'blue'], ['red', 'green', 'blue',
'blue']]
```

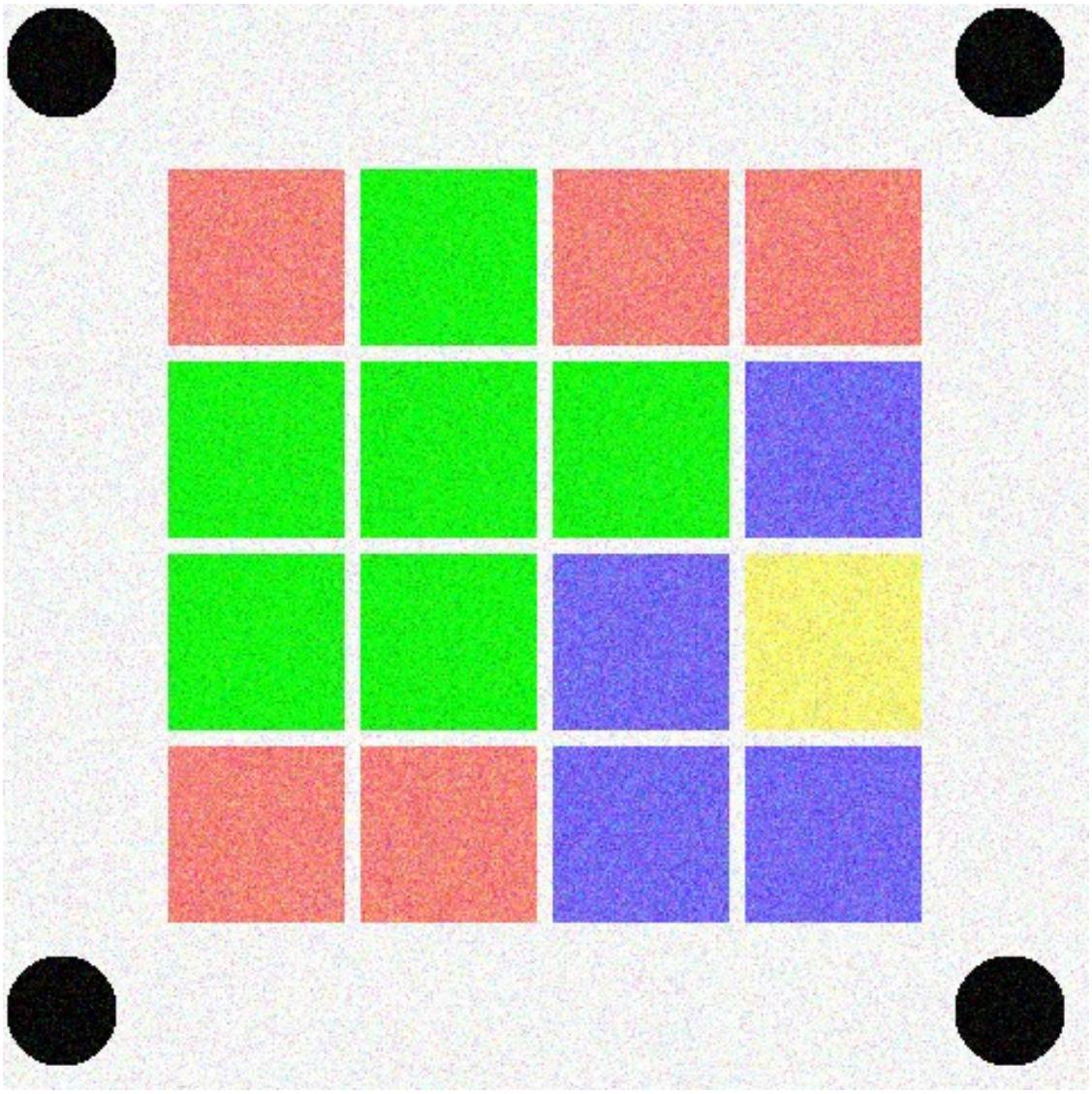
org_4



```
img = load_image('images/org_4.png')
print(find_colours(img))
```

```
[['blue', 'red', 'red', 'blue'], ['red', 'blue', 'red', 'yellow'],
['blue', 'green', 'green', 'green'], ['red', 'red', 'yellow', 'red']]
```

org_5



```
img = load_image('images/org_5.png')
print(find_colours(img))
```

```
[['red', 'green', 'red', 'red'], ['green', 'green', 'green', 'blue'],
['green', 'green', 'blue', 'yellow'], ['red', 'red', 'blue', 'blue']]
```

Accuracy

All images have been solved with 100% accuracy.

find_circles

For the undistorted images, the circle centre coordinates are: (25, 25), (445, 25), (25, 445), and (445, 445). I will define a function find_circles(image) which returns these centre

coordiantes for each image. The coordinates are fixed for the images, so the returned results should be very familiar (plus or minus a couple of pixels) for all images.

We load the image, convert it to grayscale (since colour doesn't matter here), then blur the image so that we can apply a Hough transform on the image. We can use OpenCV's HoughCircles method to detect the circles. We define a minimum and maximum radius (since we know the radius of the circles is roughly 25) as parameters of the HoughCircles function. We also input a minimum distance between the circles as 300 pixels (since we know the circles are a fixed distance from each other). OpenCV then detects the circles, and returns three values for each circle, the centre x coordinate, the centre y coordinate, and the radius.

For testing purposes, I viewed the images with the detected circles overlaid onto the images. I will comment this code out, but leave it in to demonstrate how I tested the function.

The order of circle discovery seems to be random. i.e., it does not seem to go from left-to-right, top-to-bottom, or any other permutation. All that matters is that all four circles are eventually discovered.

```
def find_circles(img):

    # Read image.
    img = cv2.imread(img, cv2.IMREAD_COLOR)

    # Convert to grayscale.
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Blur using 3 * 3 kernel.
    gray_blurred = cv2.blur(gray, (3, 3))

    # Apply Hough transform on the blurred image.
    detected_circles = cv2.HoughCircles(gray_blurred,
                       cv2.HOUGH_GRADIENT, 1, 300, param1 = 100,
                       param2 = 30, minRadius = 1, maxRadius = 40)

    # Draw circles that are detected.
    if detected_circles is not None:

        # Convert the circle parameters a, b and r to integers.
        detected_circles = np.uint16(np.around(detected_circles))

        for pt in detected_circles[0, :]:
            a, b, r = pt[0], pt[1], pt[2]
            print(a, b, r)

            # Draw the circumference of the circle.
            #cv2.circle(img, (a, b), r, (0, 255, 0), 2)

            # Draw a small circle (of radius 1) to show the center.
```

```
            #cv2.circle(img, (a, b), 1, (0, 0, 255), 3)
            #cv2.imshow("Detected Circle", img)
            #cv2.waitKey(0)

find_circles('images/noise_1.png')
```

```
26 26 25
444 26 25
26 444 25
444 444 25
```

```
find_circles('images/noise_2.png')
```

```
444 26 25
24 444 25
24 26 25
446 446 25
```

```
find_circles('images/noise_3.png')
```

```
446 444 25
26 24 25
446 26 24
24 444 24
```

```
find_circles('images/noise_4.png')
```

```
444 26 25
444 444 25
26 26 25
24 444 25
```

```
find_circles('images/noise_5.png')
```

```
26 26 24
446 26 24
26 444 24
446 446 25
```

```
find_circles('images/org_1.png')
```

```
444 26 25
26 444 25
444 444 25
24 26 25
```

```
find_circles('images/org_2.png')
```

```
26 26 25
26 444 25
444 444 25
444 24 25
```

```
find_circles('images/org_3.png')
```

```
26 26 25
26 444 25
444 446 25
446 24 24
```

```
find_circles('images/org_4.png')
```

```
26 26 25
26 444 25
444 26 25
444 444 24
```

```
find_circles('images/org_5.png')
```

```
444 444 25
444 24 25
26 444 24
24 24 24
```

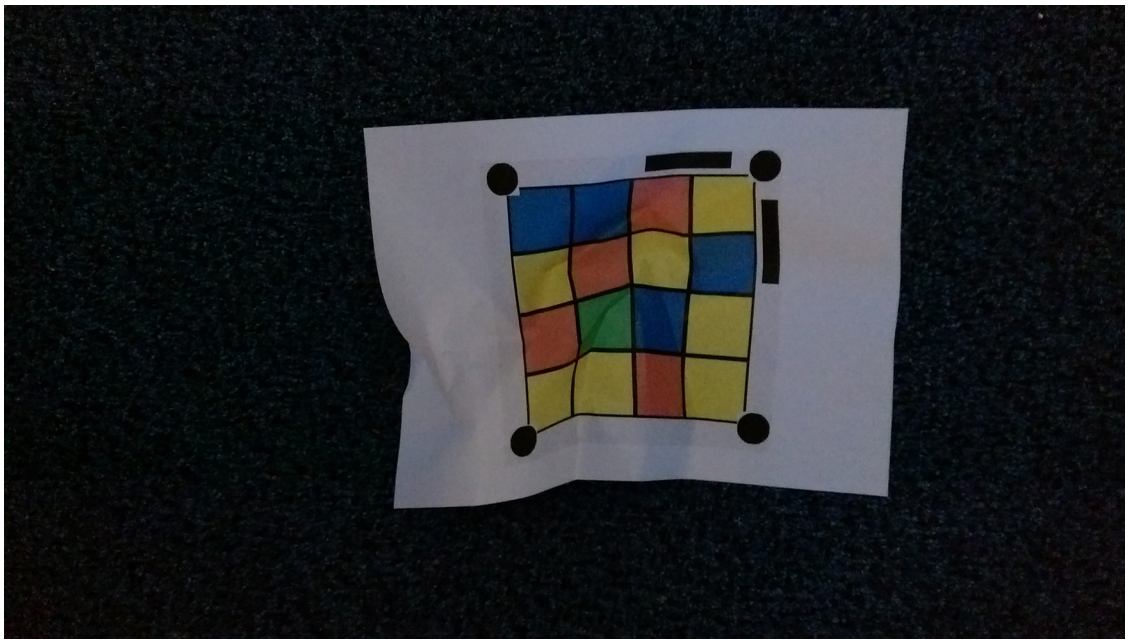
Accuracy

The coordinates and radii of the circles in all images are 100% accurate within 1 pixel of error.

Real Images

In this section, we discuss the problems of using the real photo images.

IMAG0041

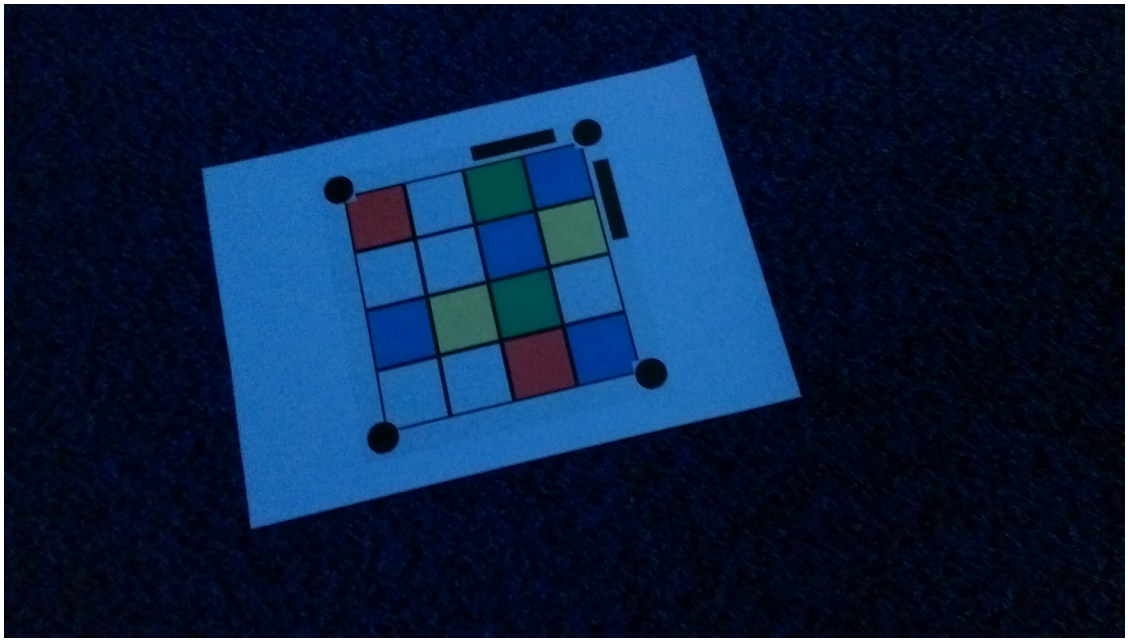


We can see that this image closely resembles the undistorted images. The centre colours are slightly off due to the paper having been bent. This bending causes the colours to be darker or lighter in some spots, depending on the altitude of the block. Overall, despite the slight distortion, we would expect our functions to perform well on this image.

IMAG0044

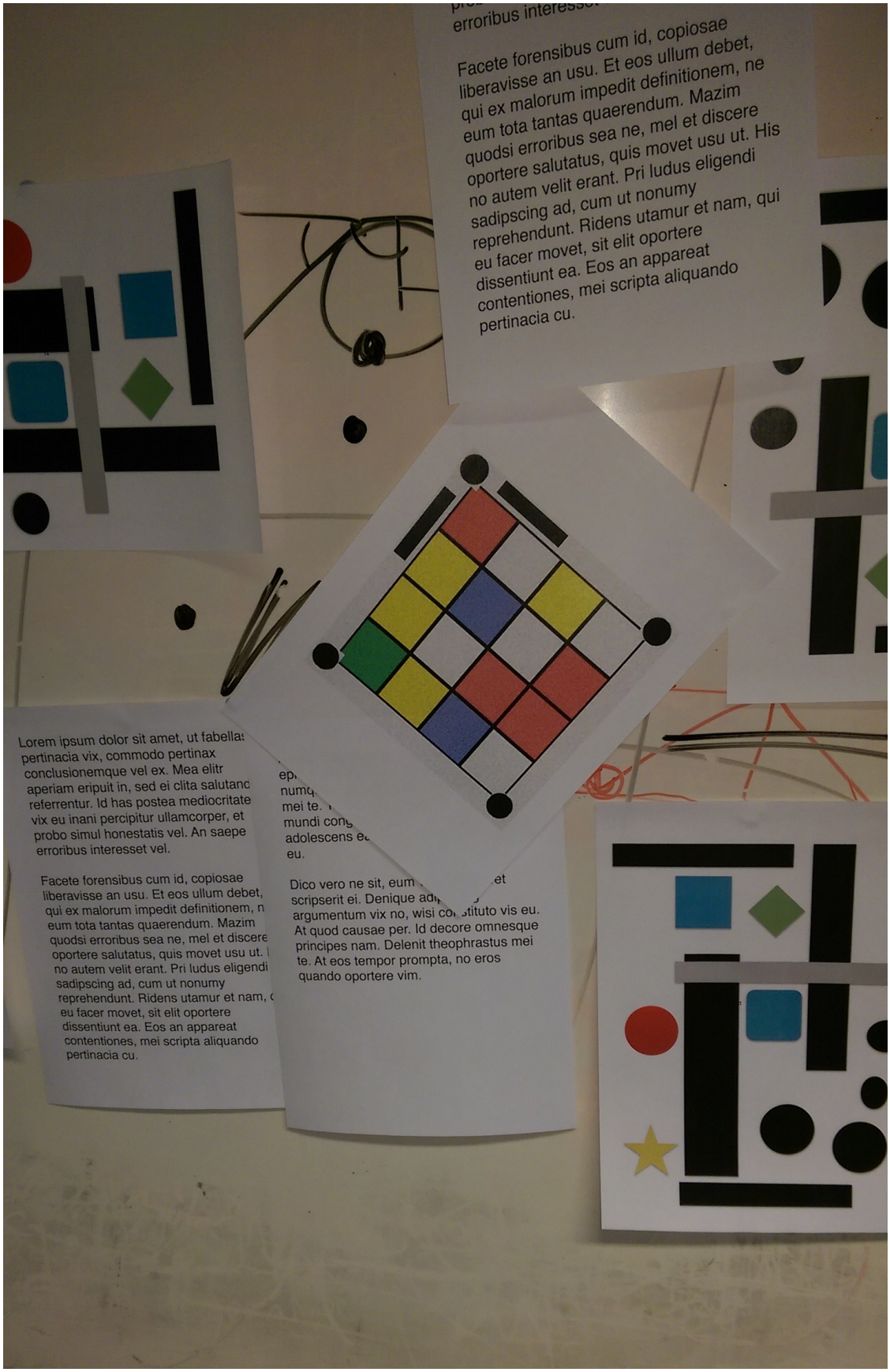
This photo looks to be challegning for our functions to solve. The noise surrounding the coloured blocks is high, which might lead to our functions being unable to locate where each block starts and ends. Also, the noise around the edges of the blocks has caused the colours to be mixed, which could cause our functions to return more colours than are actually present.

IMAG0042



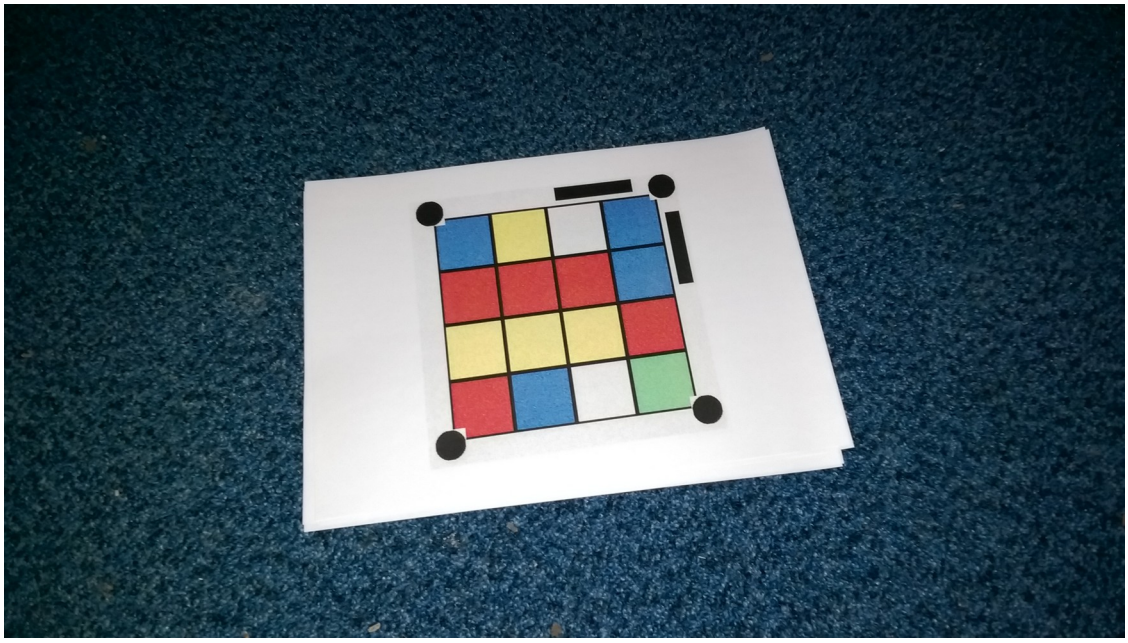
There is not much distortion to this image. We can see that there is a blueish filter over the entire image, but the colours are still readable, and our functions should not have a problem detecting them. However, the photo is slanted, which could cause our functions to return the colours in the wrong order. In the find_colours function, I have set the coordinates of the coloured blocks. I would have to reset them for this photo.

IMAG0038


erroribus interesset vel.
Facete forensibus cum id, copiosae liberavisse an usu. Et eos ullum debet, qui ex malorum impedit definitionem, ne eum tota tantas quaerendum. Mazim quodsi erroribus sea ne, mel et discere oportere salutatus, quis movet usu ut. His no autem velit erant. Pri ludus eligendi sadipscing ad, cum ut nonumy reprehendunt. Ridens utamur et nam, qui eu facer movet, sit elit oportere dissentiunt ea. Eos an appareat contentiones, mei scripta aliquando pertinacia cu.
Lorem ipsum dolor sit amet, ut fabella
pertinacia vix, commodo pertinax
conclusionemque vel ex. Mea elitr
aperiam eripuit in, sed ei clita salutanc
referrentur. Id has postea mediocritate
vix eu inani percipitur ullamcorper, et
probo simul honestatis vel. An saepe
erroribus interesset vel.
Facete forensibus cum id, copiosae
liberavisse an usu. Et eos ullum debet,
qui ex malorum impedit definitionem, n
eum tota tantas quaerendum. Mazim
quodsi erroribus sea ne, mel et discere
oportere salutatus, quis movet usu ut.
no autem velit erant. Pri ludus eligendi
sadipscing ad, cum ut nonumy
reprehendunt. Ridens utamur et nam,
eu facer movet, sit elit oportere
dissentiunt ea. Eos an appareat
contentiones, mei scripta aliquando
pertinacia cu.
mei te.
mundi con
adolescens e
eu.
Dico vero ne sit, eum
scripserit ei. Denique ad
argumentum vix no, wisi co
At quod causae per. Id decore omnesque
principes nam. Delenit theophrastus mei
te. At eos tempor prompta, no eros
quando oportere vim.

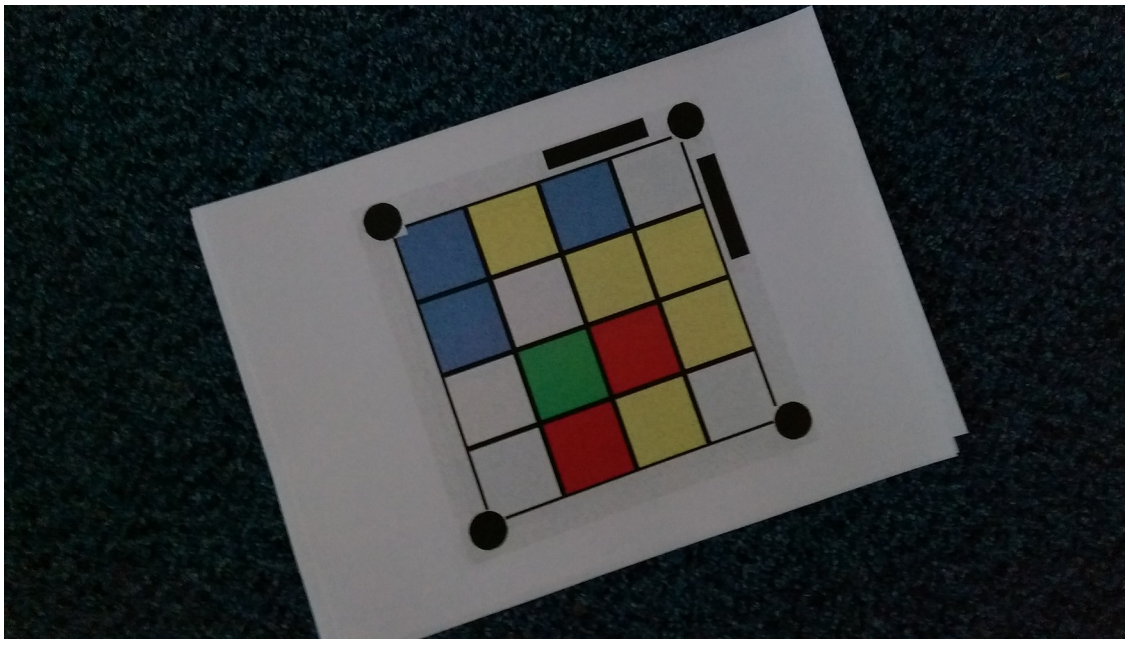
There are many colours in this image, which could cause problems for our functions if the coordinates of the coloured blocks are not set before parsing the photo. We can also see that the image has been rotated, making it unclear where the first row or column is.

IMAG0037



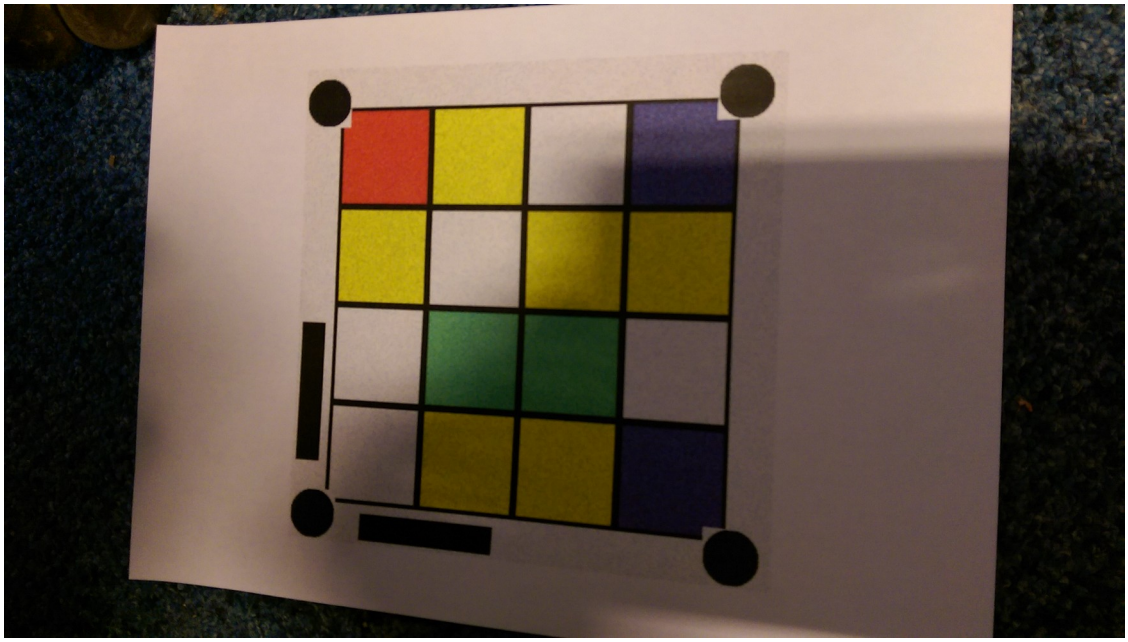
This photo is very clear. the colours are easily identifiable, and the image is only slightly slanted. Our functions should have no problem dealing with this image.

IMAG0036



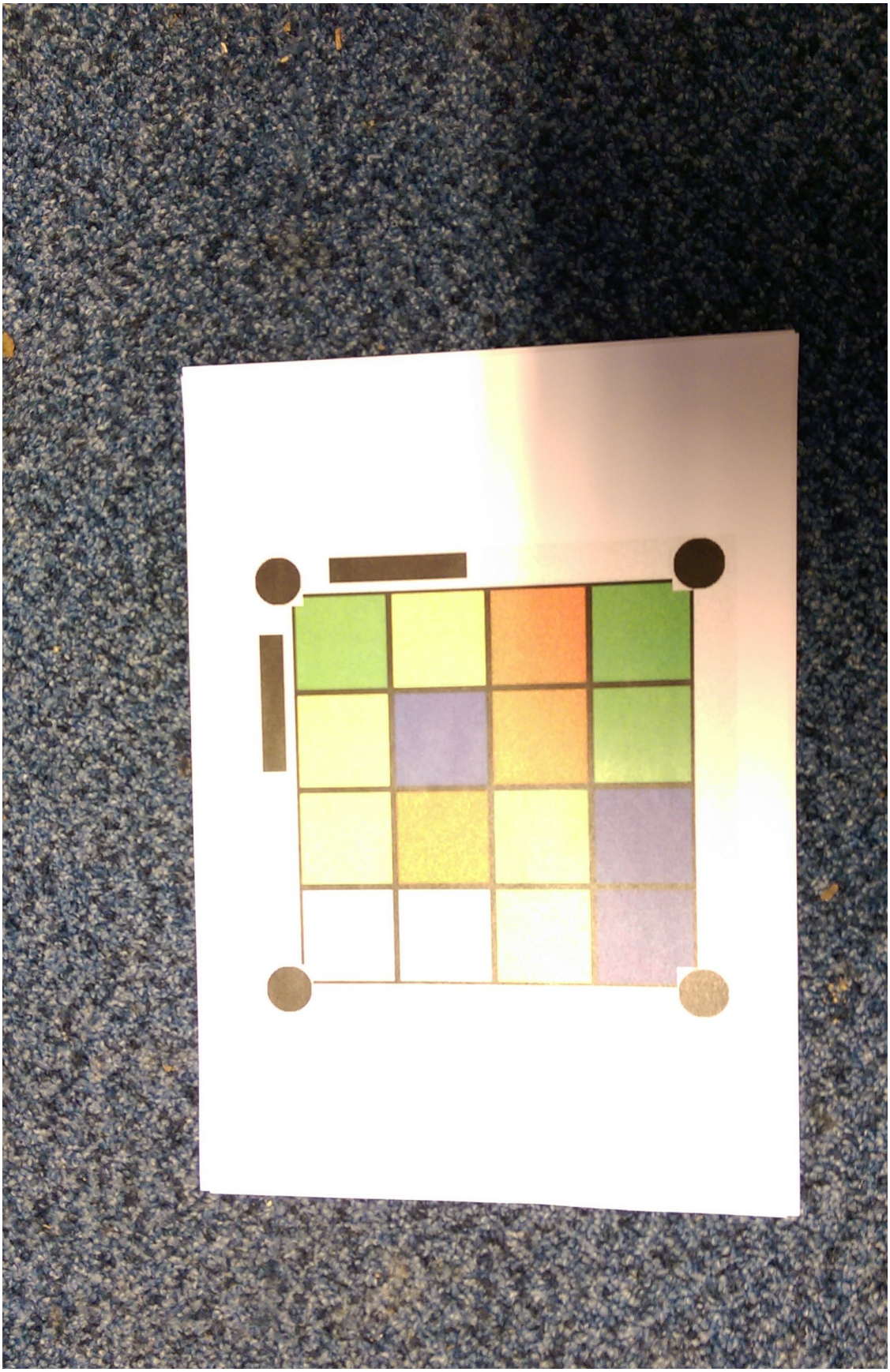
Similar to the previous photo. Instead of being bright, this photo is darker. The colours are easily identifiable though.

IMAG0035



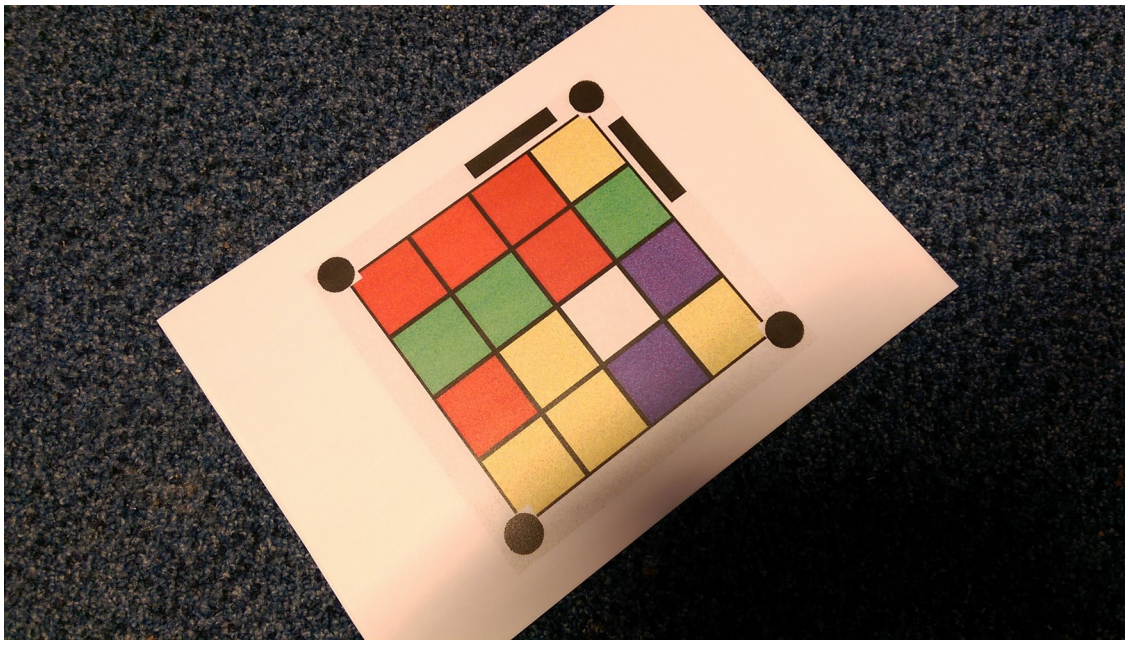
The shade in this photo should not be a problem for the green, yellow, and white colours. Looking at the bottom right blue block however, we can see that this could easily be confused as being black. It is only because we know that it shouldn't be black that we would call it blue. Our functions however, would probably classify this as black.

IMAG0034



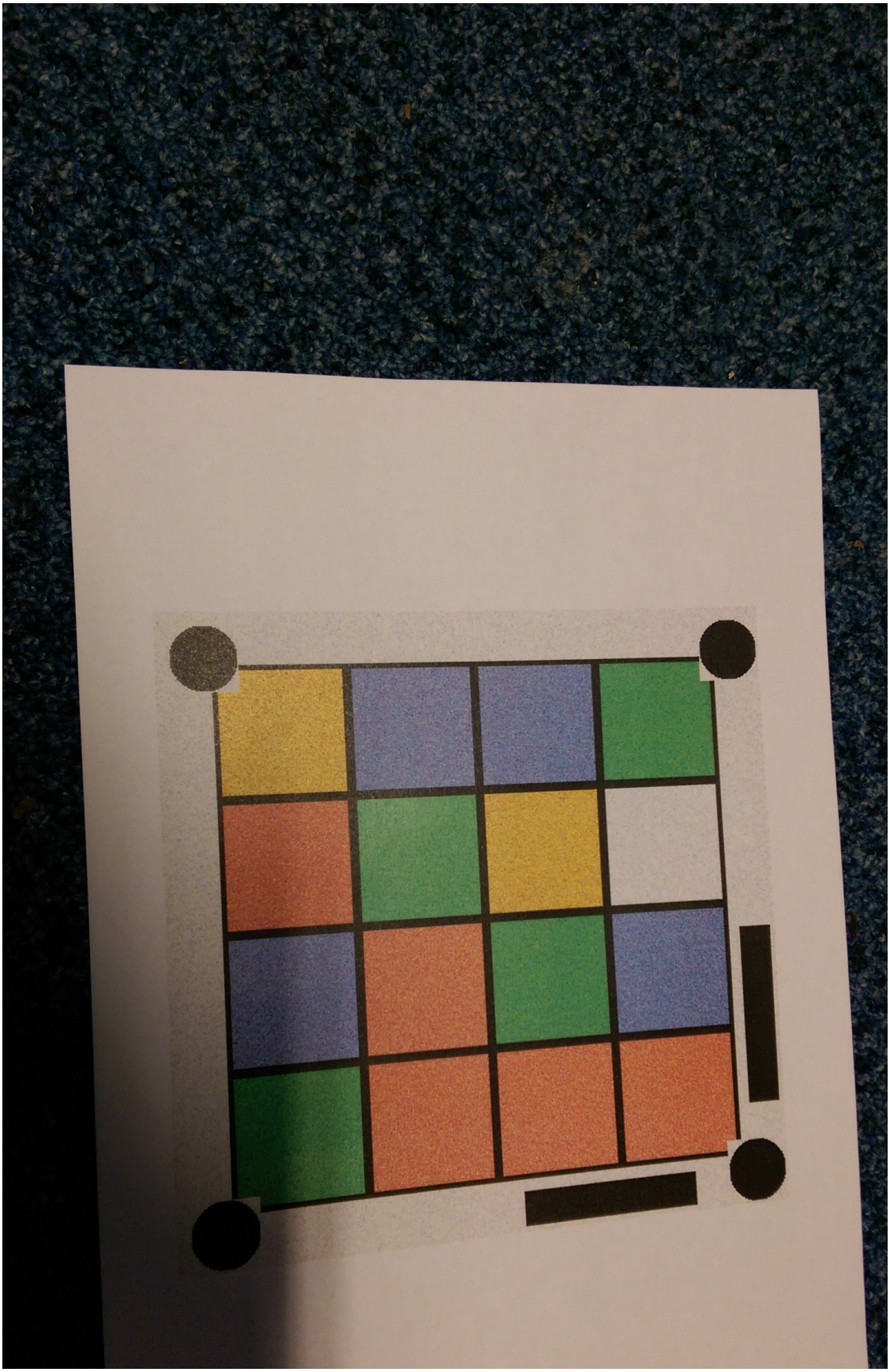
We have the inverse problem of the previous photo. We can make out most colours still, but the bottom row is hard to label. The second from right block in this row looks to be green, but our functions might classify it as white due to the centre concentration of bright light.

IMAG0033



Very slight shading in this photo. Not enough to cause errors. The rotation makes it hard to tell where the first row/column is. We would want to rotate this photo to be square for accurate colour matrix values.

IMAG0032



The shade or the slight rotation of the photo should not be a problem. We can see black noise has been added to the blocks, which could cause problems, especially considering how my colour detetction function works. My function takes the most common colour of the pixels inside the blocks as the colour of the block. In this case, we can see many black pixels, which could cause any of these blocks to be misjudged to be black. I could either take black out of my colour dataframe, or take the colour of the block to be the second most common colour, if the most common is black.