Assignment 1

March 25, 2021

1 Computer Vision 2021 Assignment 1: Image filtering

In this prac you will research, implement and test some image filtering operations. Image filtering by convolution is a fundamental step in many computer vision tasks and you will find it useful to have a firm grasp of how it works. For example, later in the course we will come across Convolutional Neural Networks (CNNs) which are built from convolutional image filters.

The main aims of the prac are:

- to understand the basics of how images are stored and processed in memory;
- to gain exposure to several common image filters, and understand how they work;
- to get practical experience implementing convolutional image filters;
- to test your intuition about image filtering by running some experiments;
- to report your results in a clear and concise manner.

This assignment relates to the following ACS CBOK areas: abstraction, design, hardware and software, data and information, HCI and programming.

1.1 General instructions

Follow the instructions in this Python notebook and the accompanying file *a1code.py* to answer each question. It's your responsibility to make sure your answer to each question is clearly labelled and easy to understand. Note that most questions require some combination of Python code, graphical output, and text analysing or describing your results. Although we will check your code as needed, marks will be assigned based on the quality of your write up rather than for code correctness! This is not a programming test - we are more interested in your understanding of the topic.

Only a small amount of code is required to answer each question. We will make extensive use of the Python libraries

- numpy for mathematical functions
- skimage for image loading and processing
- matplotlib for displaying graphical results
- jupyter for Jupyter Notebooks

You should get familiar with the documentation for these libraries so that you can use them effectively.

2 The Questions

To get started, below is some setup code to import the libraries we need. You should not need to edit it.

[1]:
```
# Numpy is the main package for scientific computing with Python.
import numpy as np

#from skimage import io

# Imports all the methods we define in the file a1code.py
from a1code import *

# Matplotlib is a useful plotting library for python
import matplotlib.pyplot as plt
# This code is to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
%reload_ext autoreload
```

2.1 Question 0: Numpy warm up! (5%)

Before starting the assignment, make sure you have a working Python 3 installation, with up to date versions of the libraries mentioned above. If this is all new to you, I'd suggest downloading an all in one Python installation such as Anaconda. Alternatively you can use a Python package manager such as pip or conda, to get the libraries you need. If you're struggling with this please ask a question on the MyUni discussion forum.

For this assignment, you need some familiarity with numpy syntax. The numpy QuickStart should be enough to get you started:

https://numpy.org/doc/stable/user/quickstart.html

Here are a few warm up exercises to make sure you understand the basics. Answer them in the space below. Be sure to print the output of each question so we can see it!

1. Create a 2D numpy array A with 2 rows and 3 columns. Fill with values 1 to 6.
2. Create a 2D numpy array B with 3 rows and 2 columns. Fill with values 6 to 1.
3. Calculate A' + B', where A' is the top left 2x2 submatrix of A.
4. Calculate the *matrix* product of A and B.
5. Calculate the *element wise* product of A and B^T (B transpose).

[2]:
```
import math

import numpy as np

A = np.arange(1,7).reshape(2,3)
B = np.array([(6,5),(4,3),(2,1)])
A_prime = np.array([[1,2],[4,5]])
B_prime = np.array([[6,5],[4,3]])
print('1:')
print(A)
print('2:')
print(B)
print('3:')
result3 = A_prime +B_prime
print(result3)
print('4:')
result4= A@B
print(result4)
print('5:')
result5=A*B.transpose()
print(result5)
```

```
1:
[[1 2 3]
 [4 5 6]]
2:
[[6 5]
 [4 3]
 [2 1]]
3:
[[7 7]
 [8 8]]
4:
[[20 14]
 [56 41]]
5:
[[ 6  8  6]
 [20 15  6]]
```

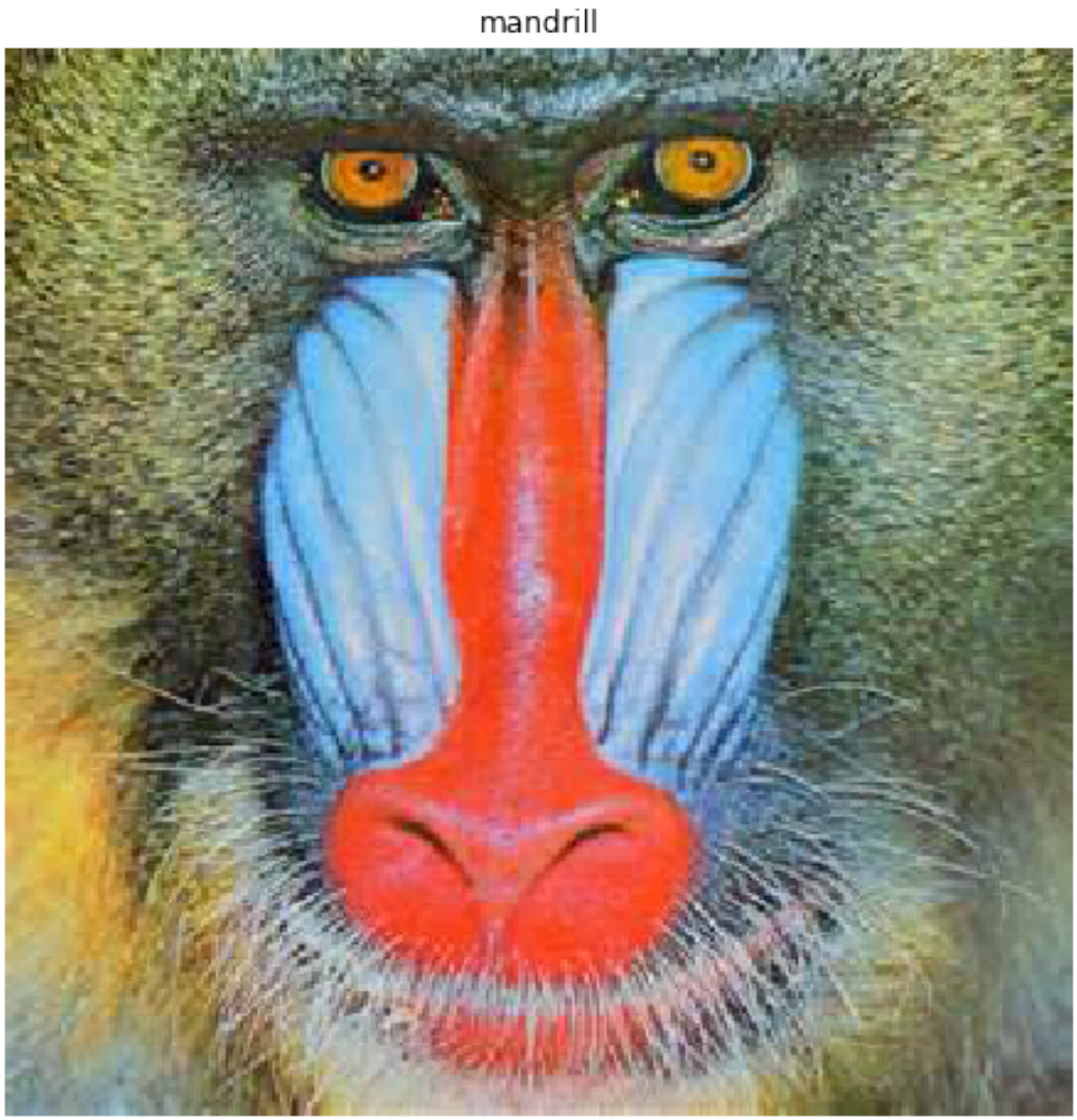
You need to be comfortable with numpy arrays because that is how we store images. Let's do that next!

2.2 Question 1: Loading and displaying an image (10%)

Below is a function to display an image using the pyplot module in matplotlib. Implement the `load()` and `print_stats()` functions in a1code.py so that the following code loads the mandrill image, displays it and prints its height, width and depth.

[7]:
```
def display(img, caption=''):
    # Show image using pyplot
    plt.figure()
    plt.imshow(img)
    plt.title(caption)
    plt.axis('off')
    plt.show()
```

[8]:
```
image1 = load('images/mandrill.jpg')

display(image1, 'mandrill')

print_stats(image1)
```



```
(300, 300, 3)
```

Return to this question after reading through the rest of the assignment. Find 2 more images to use as test cases in this assignment and display them below. Use your print_stats() function to display their height, width and number of channels. Explain *why* you have chosen each image.

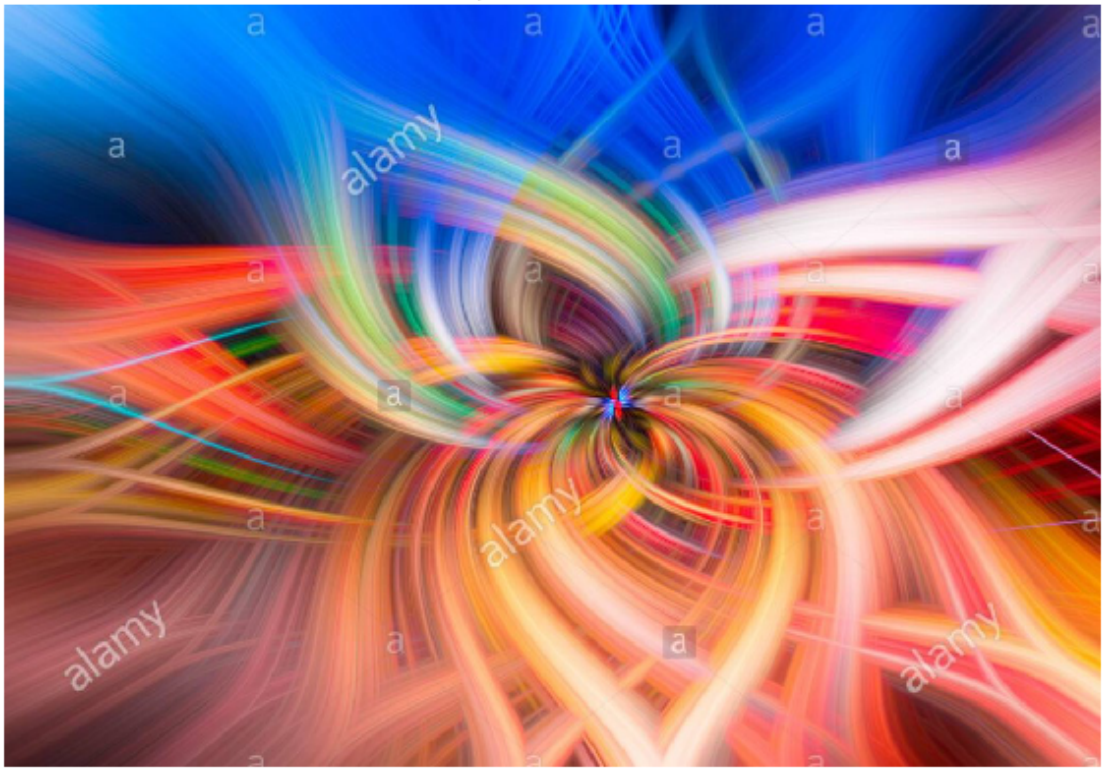
[4]:
```
### Your code to load and display your images here
pic1 = load('one.jpg')
display(pic1)
print_stats(pic1)

pic2 = load('two.jpg')
display(pic2, 'picture two')
print_stats(pic2)

pic3 = load('three.jpg')
display(pic3, 'picture three')
print_stats(pic3)

pic4 = load('four.jpg')
display(pic4, 'picture four')
print_stats(pic4)

pic5 = load('five.jpg')
display(pic5, 'Picture five')
print_stats(pic5)
```

```
(620, 828, 3)
```

picture two

```
(512, 768, 3)
```

picture three



(727, 1047, 3)

picture four

(756, 1168, 3)


```
(360, 427, 3)
```

Using the load function, the image in the file is loaded into this program from its path images/mandrill. The image is a RGB mandrill image i.e. is a 3 layers array representing red green and blue. The combination of these 3 colours can output a coloured image. When we overlap these 3 layers together we get a 3D array. The image height and width is 300x300, thus a coulourful image is in size of 300x300x3.

There are five images chosen for this assignment, four colourful images and one greyscale image. They all have their own height and width. The grey one also has 3 channels, this implies its RGB channel values are all avaerged to be the same, just like the greyscale function defined in python code used in next section.

2.3 Question 2: Image processing (20%)

Now that you have an image stored as a numpy array, let's try some operations on it.

1. Implement the `crop()` function in a1code.py. Use array slicing to crop the image.
2. Implement the `resize()` function in a1code.py.

3. Implement the `change_contrast()` function in a1code.py.
4. Implement the `greyscale()` function in a1code.py.

Is the contrast function always reversible? Explain why or why not.

[15]:
```
# This should crop the eye from the mandrill image
display(crop(image1, 25, 80, 25, 50),'Cropped Image')
display(resize(image1, 200, 400 ),'resized image:')
print('Resized image stats:')
print_stats(resize(image1, 200, 400 ))
display(change_contrast(image1, 0.5),'Change Contrast(0.5):')
contrast = change_contrast(image1, 1.5)
contrast = contrast/np.amax(contrast)# Dealing with the clipping warning
contrast = np.clip(contrast, 0, 1)
display(contrast,'Change Contrast(1.5):')
grey=greyscale(image1)
display(grey,'Grey Image:')

# Add your own tests here...
display(crop(pic4, 10, 10, 400, 700),'Crop test:')
display(resize(pic2, 200, 400 ),'Resize Test:')
display(change_contrast(pic3, 0.5), 'Contrast test 0.5:')
contrast1 = change_contrast(pic3, 1.5)
contrast1 = contrast1/np.amax(contrast1)
contrast1 = np.clip(contrast1, 0, 1)
display(contrast1,'Contast Test 1.5:')
grey=greyscale(pic1)
display(grey,'Grey Test:')
```

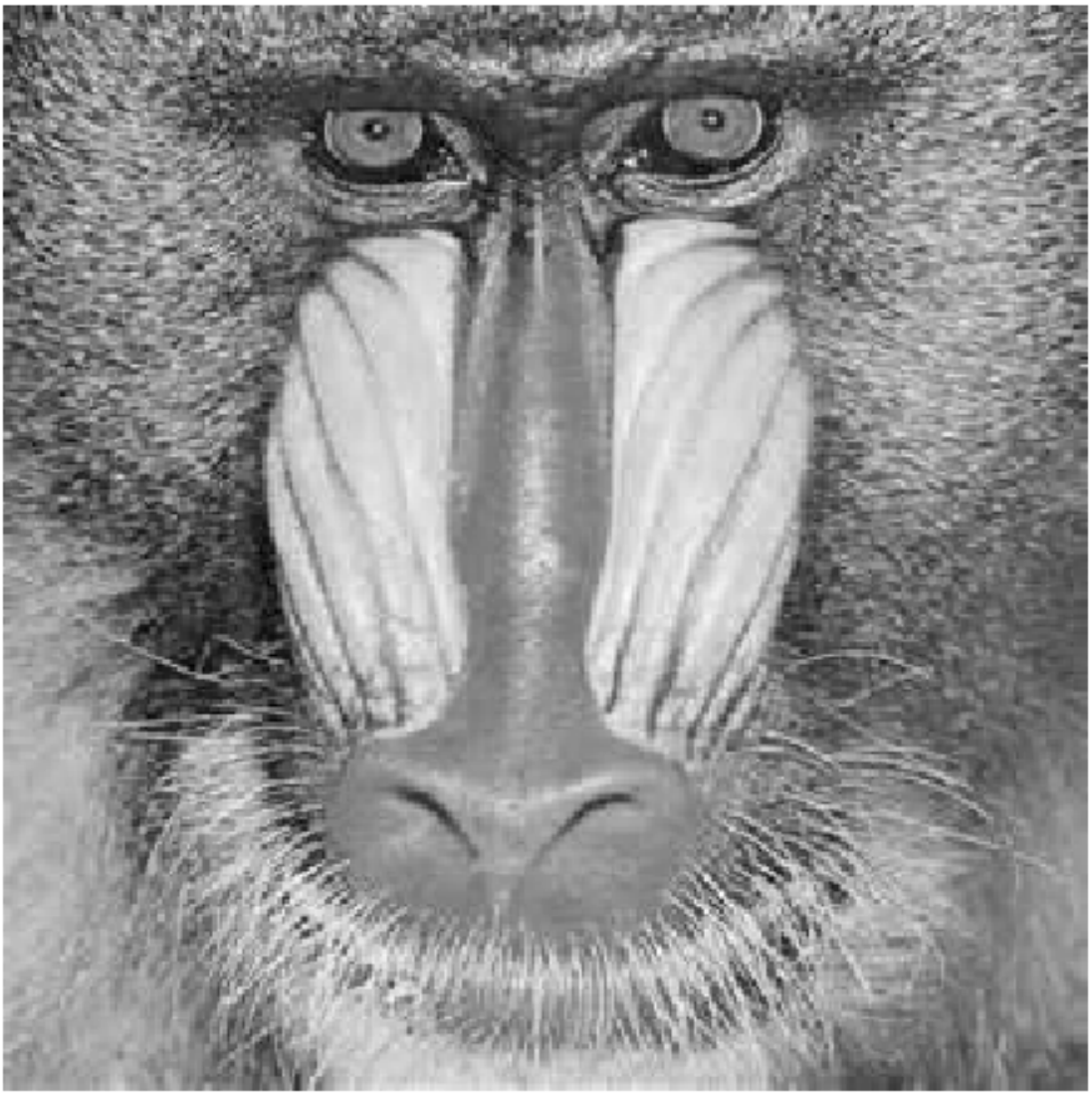
resized image:

```
Resized image stats:
(200, 400, 3)
```

Change Contrast(0.5):

Change Contrast(1.5):

Grey Image:

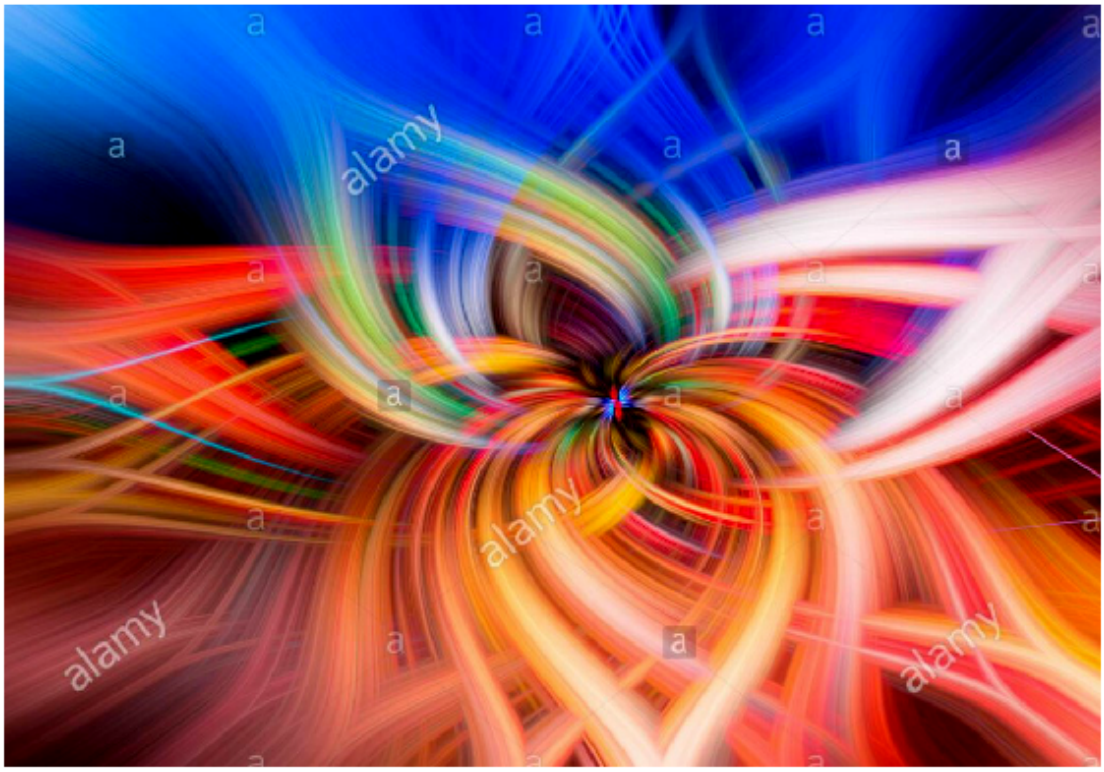


Crop test:

Resize Test:

Contrast test 0.5:

Contast Test 1.5:



As shown above all the functions passed the tests. The eye is successfully cropped out of the image. The size is successfully changed from 300*300*3 to 200*400*3. The contrast changing function successfully changed the contrast with low contrast of 0.5 and high contrast of 1.5. Anad the greyscale scale function successfully changed the colored image into a grey image shown in the pic of 'Grey Image'. Not all contrast functions are reversible, but this change contrast function is reversible due to it's a linear operation, we can simply reverse it to its original by multiplying or dividing the ratio. In other cases, there are non-linear contrast changing functions, such as th eexample in the lecture 1 slide, a non-linear lower contrast can be ((x/255)^(1/3))*255

2.4 Question 3: Convolution (20%)

2.4.1 3.1 2D convolution

Using the definition of 2D convolution from week 1, implement the convolution operation in the function `conv2D()` in a1code.py.

[17]:
```
test_conv2D()
```

```
correct
output:
[[0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

```
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 1. 2. 1. 1. 0. 0.]
 [0. 0. 1. 1. 2. 1. 1. 0. 0.]
 [0. 0. 2. 2. 3. 1. 1. 0. 0.]
 [0. 0. 1. 1. 1. 0. 0. 0. 0.]
 [0. 0. 1. 1. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]]
expect:
[[0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 1. 2. 1. 1. 0. 0.]
 [0. 0. 1. 1. 2. 1. 1. 0. 0.]
 [0. 0. 2. 2. 3. 1. 1. 0. 0.]
 [0. 0. 1. 1. 1. 0. 0. 0. 0.]
 [0. 0. 1. 1. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

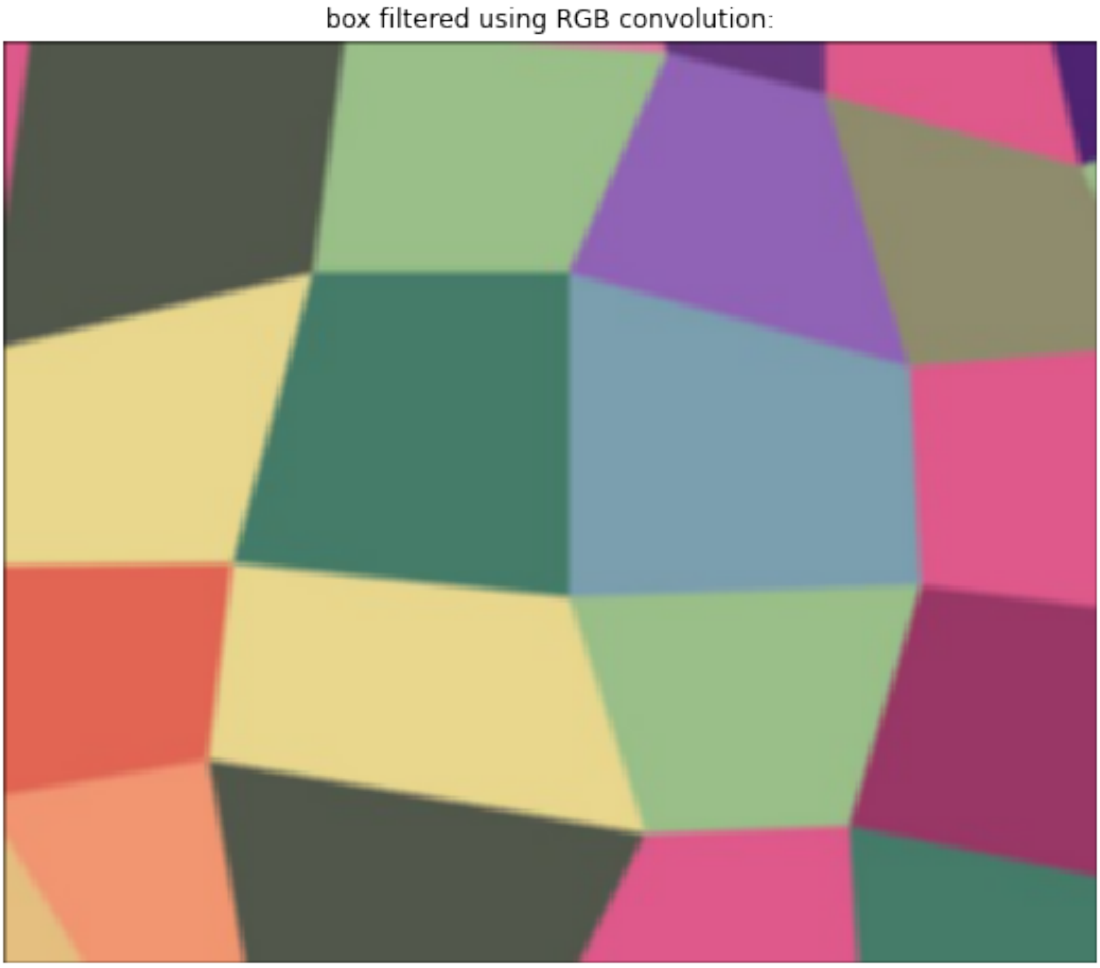
As shown above the 2D convolution function passed the test.

2.4.2 3.2 RGB convolution

In the function `conv` in a1code.py, extend your function `conv2D` to work on RGB images, by applying the 2D convolution to each channel independently.

[17]:
```
box_kernel = B = np.array([(1/16,1/16,1/16,1/16),(1/16,1/16,1/16,1/16),(1/16,1/
↪16,1/16,1/16),(1/16,1/16,1/16,1/16)])
boxFiltered = conv(pic5,box_kernel)
display(pic5,'Original picture 5:')
display(boxFiltered,'box filtered using RGB convolution:')
```

Original picture 5:



Using the RGB convolution, a 4x4 box filter is applied to the picture 5. As shown is the processed image, the picture is blurred, which is obvious around the colour block edges. Because the box filter is replacing each pixel with its local average. The blurred effect we got indicates the success of the RGB conv() function.

2.4.3 3.3 Gaussian filter convolution

Use the `gauss2D` function provided in a1code.py to create a Gaussian kernel, and apply it to your images with convolution. You will obtain marks for trying different tests and analysing the results, for example:

- try varying the image size, and the size and variance of the filter
- subtract the filtered image from the original - this gives you an idea of what information is lost when filtering

What do you observe and why?

2.4.4 3.4 Sobel filters

Define a horizontal and vertical Sobel edge filter kernel and test them on your images. You will obtain marks for testing them and displaying results in interesting ways, for example:

- apply them to an image at different scales
- considering how to display positive and negative gradients
- apply different combinations of horizontal and vertical filters

[19]:
```
# Your code to answer 3.3, 3.4 and displaay results here.
kernel_1 = gauss2D(3, 0.3)
display(conv(image1, kernel_1),'image1 (3,0.3)')
display(conv(resize(image1,200,400),kernel_1),'Gaussian Resized Image:')

kernel_2 = gauss2D(3, 0.9)
display(conv(image1, kernel_2),'Image 2 (3,0.9):')

difference = image1 - conv(image1, kernel_2)
difference = np.clip(difference,0,1)
display(difference,'Difference:')

kernel_3 = gauss2D(9, 0.3)
image3 = conv(image1, kernel_3)
image3 = image3/np.amax(image3)#dealing with the clipping warning
image3 = np.clip(image3, 0, 1)
display(image3,'image 3 (9,0.3)')
kernel_4 = gauss2D(9, 0.9)
display(conv(image1, kernel_4),'Image 4 (9,0.9)')

sobel_kernel_H = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])
sobel_kernel_V = np.array([[1,2,1],[0,0,0],[-1,-2,-1]])
H_sobel=conv(image1, sobel_kernel_H)
H_sobel=H_sobel/np.amax(H_sobel)#clip
H_sobel=np.clip(H_sobel,0,1)
display(H_sobel,'Horizontal Sobel:')
V_sobel=conv(image1, sobel_kernel_V)
V_sobel=V_sobel/np.amax(V_sobel)#clip
V_sobel=np.clip(V_sobel,0,1)
display(V_sobel,'Vertical Sobel:')
```

image1 (3,0.3)

Gaussian Resized Image:

Image 2 (3,0.9):

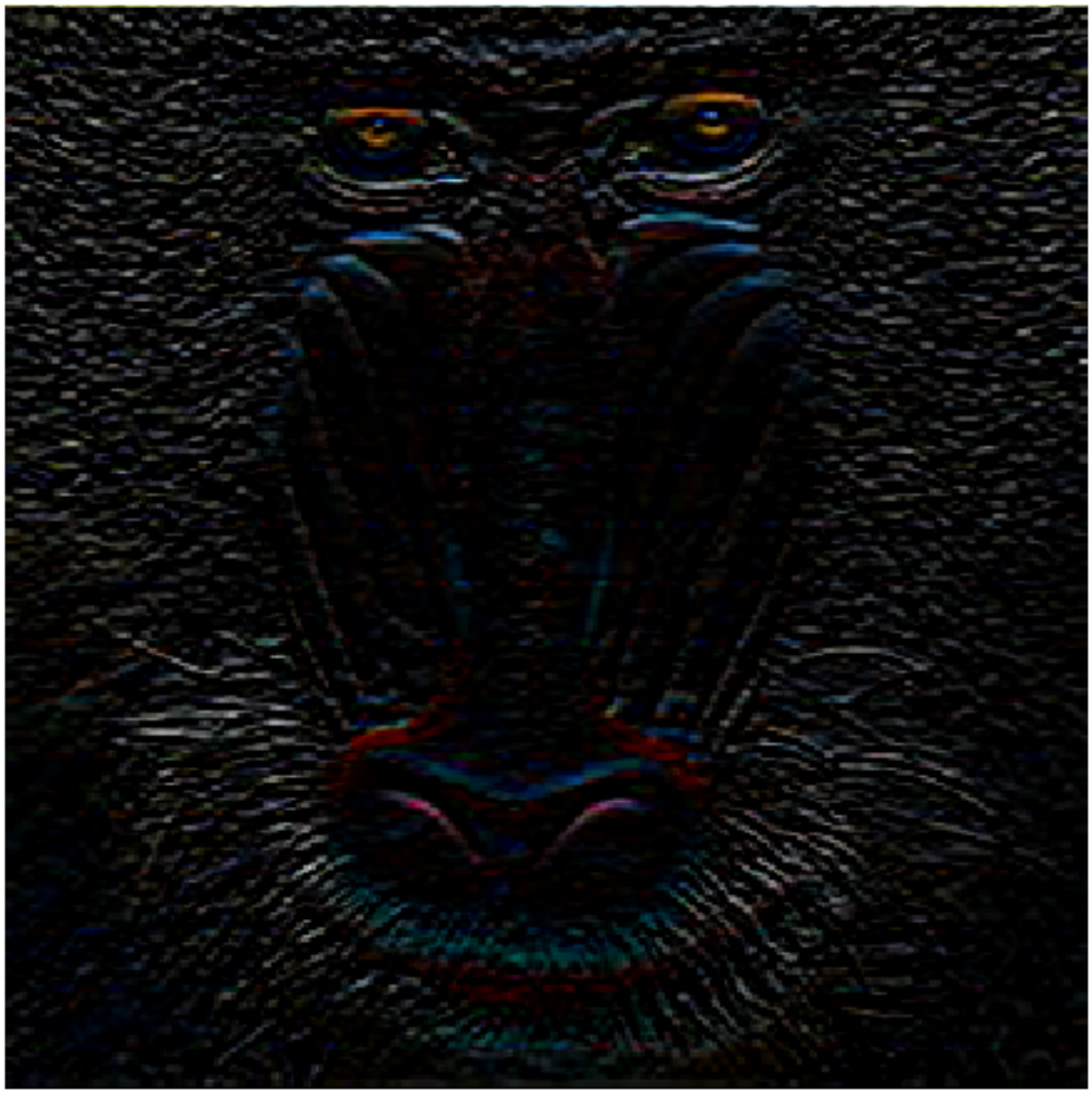
Difference:

image 3 (9,0.3)

Image 4 (9,0.9)

Horizontal Sobel:

Vertical Sobel:



As shown in the picture, from image1(3,0.3) vs image2 (3, 0.9), and image3(9,0.3) vs image4 (9, 0.5), we found that with the increase of the standard deviation, the weight falls of with distance from centre pixel slowly, and we can get a more blurred image. From image1 vs 3, and image2 vs 4, we found the larger the kernel sizes are the more blurred the images are. This is because larger kernel have more values factored into the average, and this implies that a larger kernel will blur the image more than a smaller kernel. The Gaussian filtered resized image has no big difference compared with the fintered version of original size. The lost information due to the downsampling using resize will be talked more detailed in the section 4. The gaussian difference contains the the edges and the regions that have shap changes, because the gaussian filter can smooth the image. When we take the difference from the original image, it only gives the region it smoothed. Also, the edges are the place got smoothed the most. Therefore there are edges on the difference image.

In the last two images, we found that the image used horiontal sobel filter has smoothing effect in

horizontal direction and edgeds are more clearer in vertical direction. If we look at the horizontal filter itself, it takes the difference for vertical direction, and have gaussian smoothing effect in horizontal direction. The image used vertical sobel filter has smoothing effect in vertical direction and edgeds are more clearer in horizontal direction. This makes sense, because the vertical sobel filter works the same way as the horizontal sobel filter, but with the direction changed.

2.5 Question 4: Image sampling and pyramids (25%)

2.5.1 4.1 Image Sampling

Apply your `resize()` function to reduce an image to 0.125 height and width, and then to enlarge the image back to its original size. Display the result and compare to the original image. Apply a Sobel filter to the resulting image and compare to the original. What do you observe and why?

2.5.2 4.2 Image Pyramids

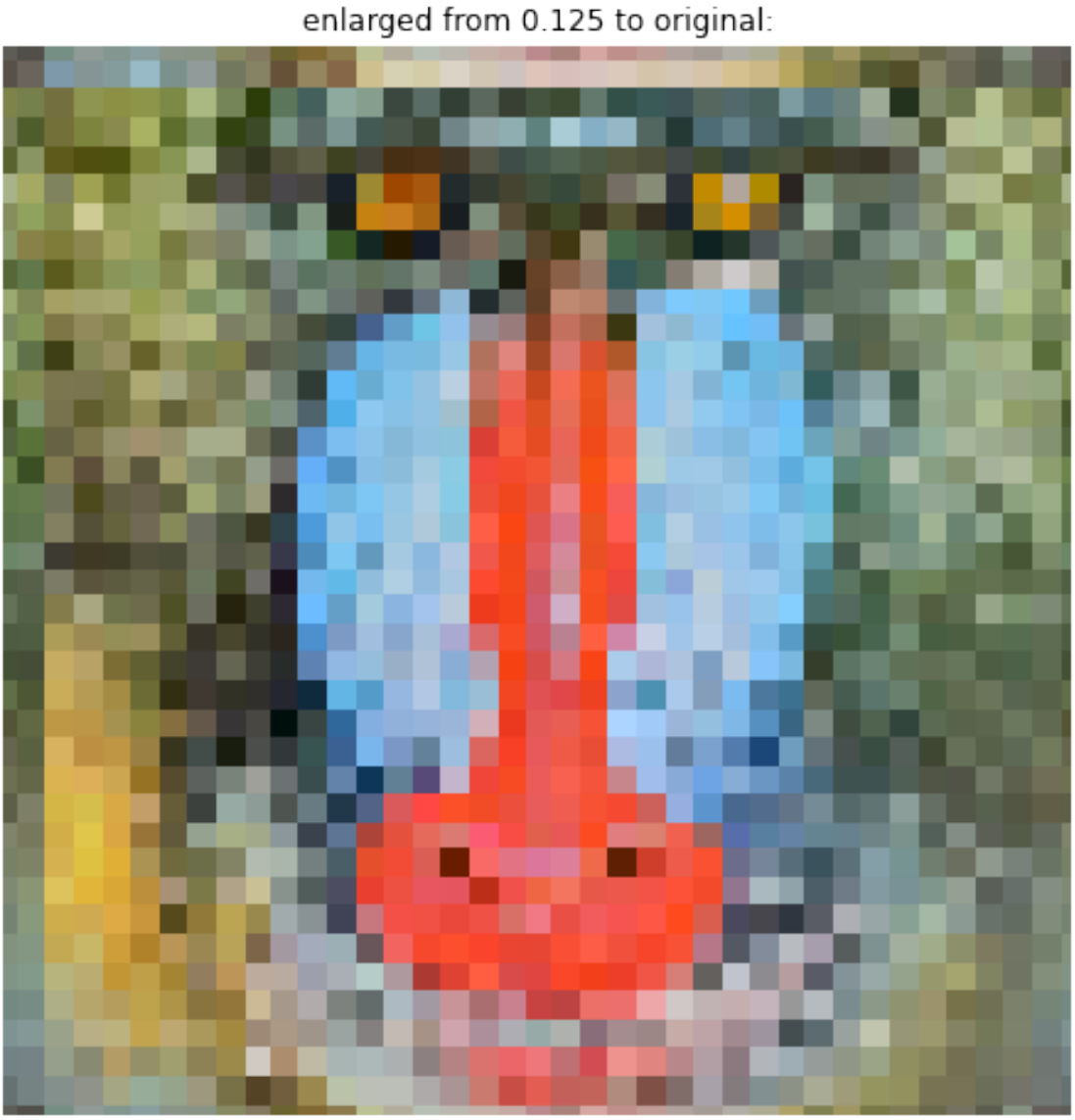
Repeat this procedure, but this time, apply a Gaussian filter to the image before resizing. What do observe, and why?

Repeat again, but this time apply the scaling in 3 steps of 0.5, creating a pyramid of images. What do you observe, and why?

[25]:
```
# Your answers to question 4 here
#4.1
display(resize(image1, int(round(37.5)), int(round(37.5))),'reduced to 0.125')
small = resize(image1, int(round(37.5)), int(round(37.5)))
print_stats(small)
large = resize(small, 300, 300)
display(large,'enlarged from 0.125 to original:')
print_stats(large)

sobel_kernel_H = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])
sobel=conv(large, sobel_kernel_H)
sobel = sobel/np.amax(sobel)
sobel = np.clip(sobel, 0, 1)
display(sobel,'Horizontal Sobel filter on resize processed Image:')

#4.2
kernel_1 = gauss2D(3, 0.7)
gaussed = conv(image1, kernel_1)
Gauss_small = resize(gaussed,38, 38)
display(resize(gaussed,38, 38),'Gaussian Small:')
print_stats(Gauss_small)
display(resize(Gauss_small,300,300),'Gaussian Large:')

#pyramid
print('pyramid:')
#step1
print('Getting small:')
```

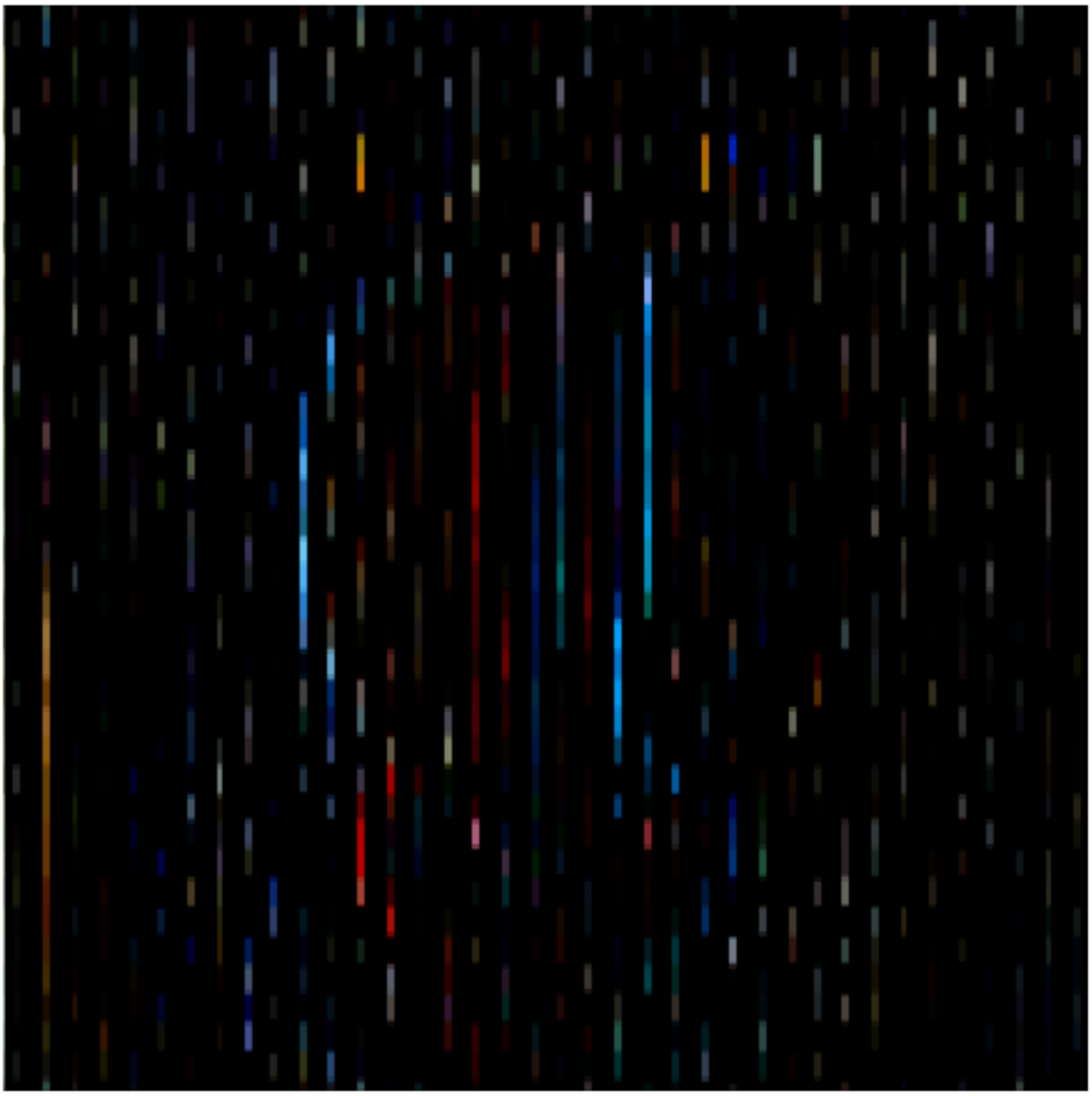
```
gaussed = conv(image1,kernel_1)
image_small1 = resize(gaussed, 150,150)
display(image_small1)

#step2
gaussed = conv(image_small1, kernel_1)
image_small2 = resize(gaussed, 75,75)
display(image_small2)

#step3
gaussed = conv(image_small2, kernel_1)
image_small3 = resize(gaussed,37,37)
display(image_small3)

#enlarge
print('Getting large:')
#step1
image_large1 = resize(image_small3, 75, 75)
display(image_large1)

#step2
image_large2 = resize(image_large1, 150,150)
display(image_large2)

#step3
image_large3 = resize(image_large2, 300,300)
display(image_large3,'Pyramid Final Image')
```


```
(38, 38, 3)
```




```
(300, 300, 3)
```

Horizontal Sobel filter on resize processed Image:



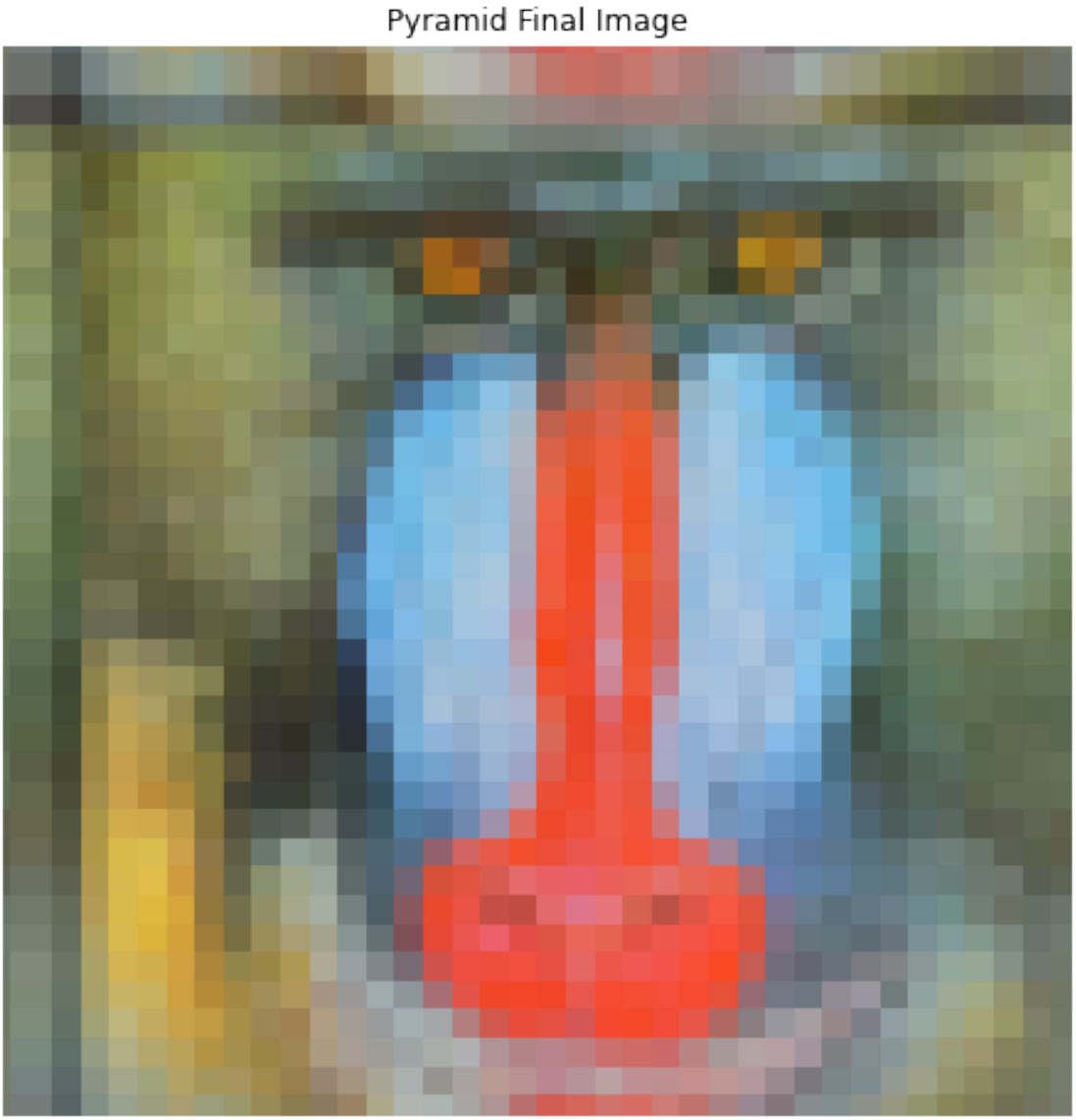
Gaussian Small:

```
(38, 38, 3)
```

Gaussian Large:

```
pyramid:
Getting small:
```

Getting large:



Using the resize(), we lost a lot information during downsampling, the unsampled pixels' information is completely lost. The horizontal sobel filter can only detect the edge of the sampled pixel. Due to this downsampling, only the selected pixel on the edge line is conserved, so there is only doted line edge detected. However, using the gaussian filter before resizing, we can keep some sampled pixel's neighouring infomation, but the details get smoothed out as we move to higher level. Therefore, when we resize the image into smaller one, the smaller ones looks more intact that the ones without Gaussian. At higher levels, mostly large uniform regions in the original image are preserved. As shown above, the pyramid resize process has better result than the previous process. Compared with the first procedure, the gaussian pyramid procedure conserved a lot of image information if we compare the pyramid final image and the .

2.6 Question 5: Auto correlation (20%)

Recall from week 2 that cross-correlation is a measure of similarity between a template and an image. It is defined similarly to convolution.

2.6.1 5.1 Correlation function

Implement the function `corr()` in a1code.py, based on your convolution function. Hint: numpy's `flip()` function may be useful here.

2.6.2 5.2 Auto-correlation

We will experiment with *auto-correlation*, where the template patch is taken from the image it is being compared to. Use the cropped eye from Question 2 as your template. Calculate the correlation of that template with every location in the image, and display the output as an image. Where is the maximum similarity? (Hint: numpy functions `argmax()` and `unravel()` will be useful here). For simplicity, you can use a greyscale version of the image and template.

Is it what you expect? Why or why not?

2.6.3 5.3 Modified auto-correlation

Try modifying your correlation template or the base image in different ways, and analyse the effect on correlation results. For example:

- if you did not find the correct location in 5.2, try centering the template about its mean (i.e. subtracting the mean brightness from each pixel)
- if you did find the correct location in 5.2, try using `resize()` and `change_contrast()` on the image. Where does it fail?

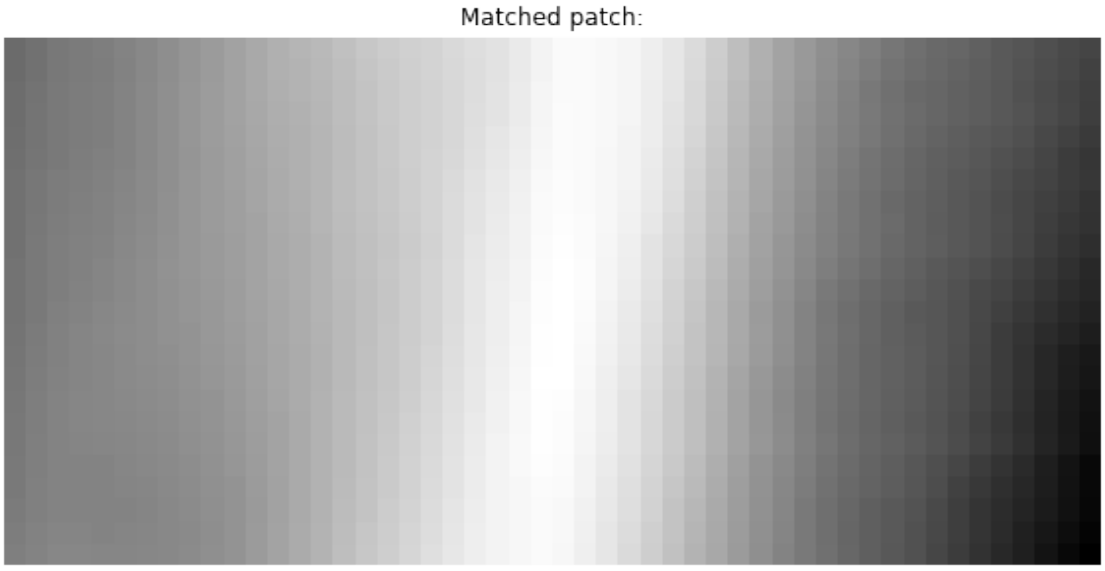
As before you will obtain marks for coming up with interesting tests and analysis, and displaying your results clearly.

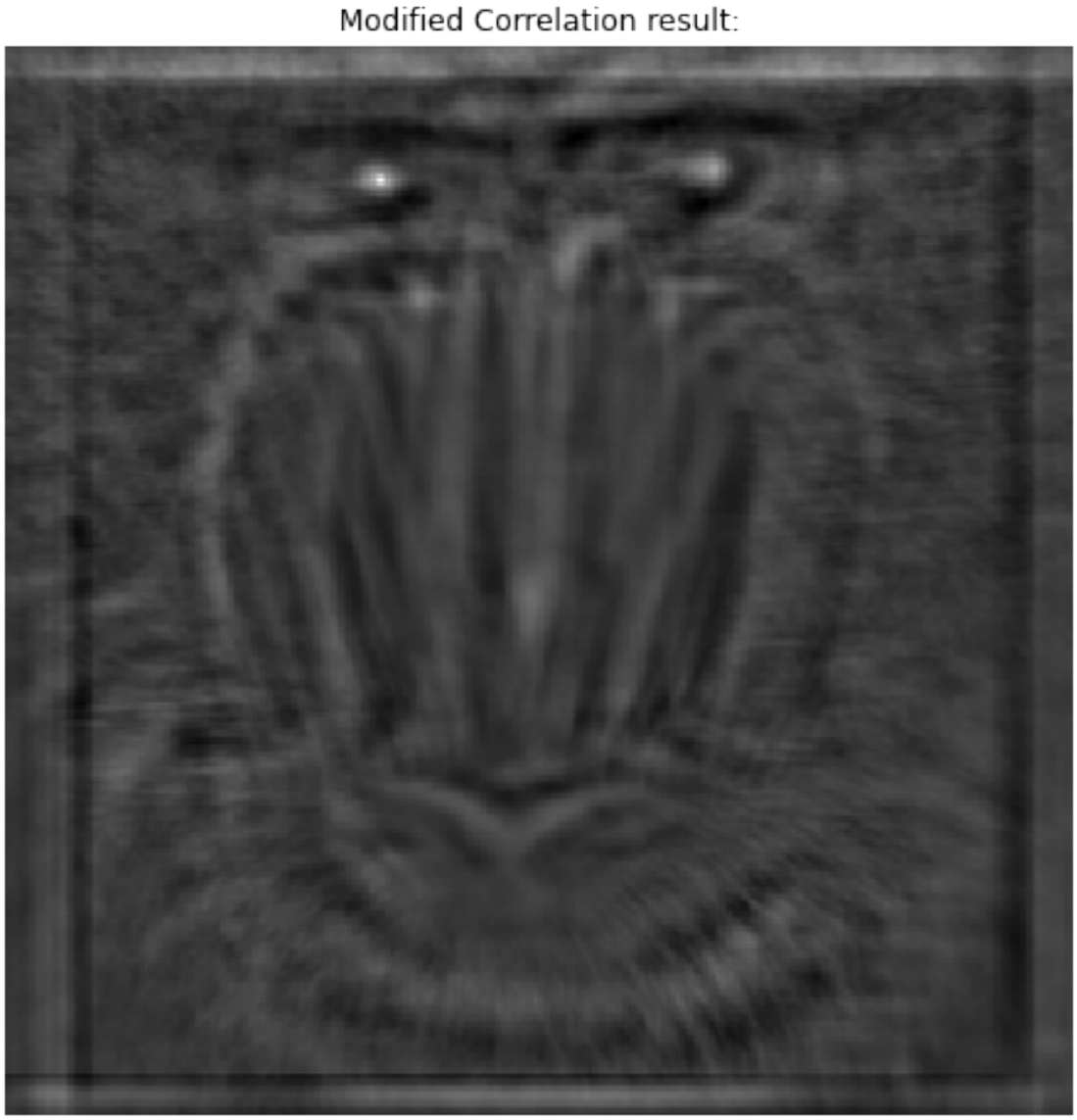
[29]:
```
# Your code to answer question 5 and display results here

Hi=image1.shape[0]
Wi=image1.shape[1]
template =crop(image1, 25, 80, 25, 50)
Hk=template.shape[0]
Wk=template.shape[1]
template_grey = greyscale(template)
template_grey = template_grey/np.amax(template_grey)

image1_grey = greyscale(image1)
result = corr(image1_grey, template_grey)
result = corr(image1_grey, template_grey)/np.amax(result)
display(result,'Correlation result image:')
ind = np.unravel_index(result.argmax(), result.shape)
print('matched centre:')
print(np.unravel_index(result.argmax(), result.shape))
patch = result[ind[0]-(Hk//2):ind[0]+(Hk//2),ind[1]-(Wk//2):ind[1]+(Wk//2)]
```

```
display(patch,'Matched patch:')

#5.3
template1 = template - np.mean(template)
template1_grey = greyscale(template1)
result1 = corr(image1_grey, template1_grey)
#Dealing with the clipping
result1 = corr(image1_grey, template1_grey)/np.amax(result1)
display(result1,'Modified Correlation result:')
ind = np.unravel_index(result1.argmax(), result1.shape)
print('matched patch of modified correlation:')
print(np.unravel_index(result1.argmax(), result1.shape))
patch1 = result1[ind[0]-(Hk//2):ind[0]+(Hk//2),ind[1]-(Wk//2):ind[1]+(Wk//2)]
display(patch1,)
```

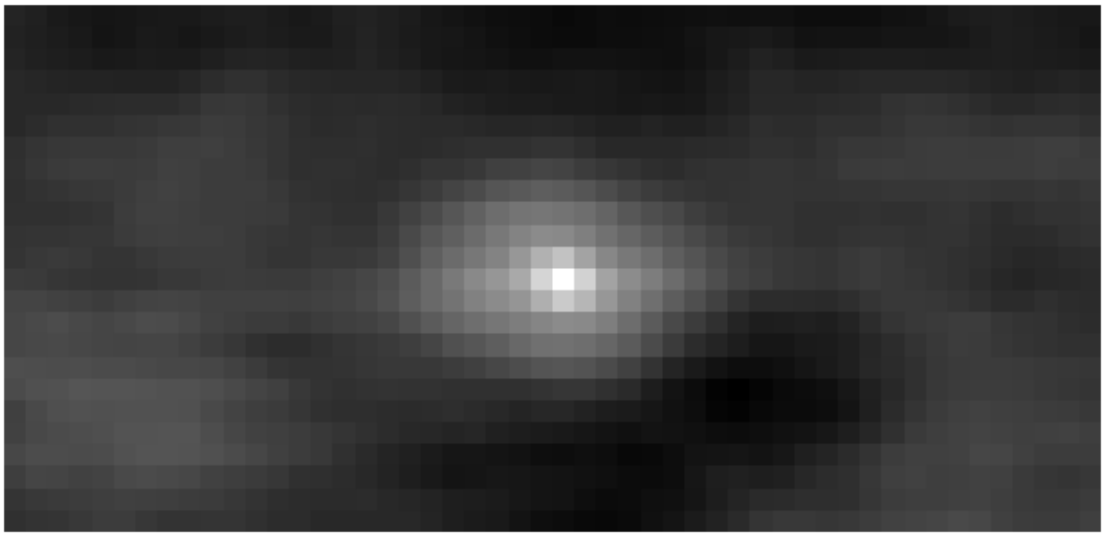
Correlation result image:

```
matched centre:
(126, 194)
```






```
matched patch of modified correlation:
(37, 105)
```



Correlation is a matching operation,i.e. a measurement of the similarity between the template and the image. The output of the correlation is a image that have the most bright (high pixel value) at the similar patch location. In 5.2, the brightest location is at (126,194) which is around the nose location. This is not the correct location, but this makes sense, because the nose is the brighest area, when we do the correlation, the template pixel values times these brightest pixels at nose will give the larger values compared with the correct eye area. This can be solved in 5.3 modified correlation, which simply subtract the mean template pixel value from the template. In this way, we will get a well distributed positive values and negative values on the template. Therefore, when we do the correlation with this modified template, the correct patch will have the sum of all positive products, while the nose area will have the sum of the combination of positive and negative products. Therefore, the correct patch i.e. the eye area will have the largest pixel values in the result image. As shown in the image abouve the modified correlation found the correct location of (37,105) and the patch of the eye.

2.7 Question 6: Normalised cross correlation (postgraduate, 10%)

This question is required for postgraduate students only. PG marks for the other questions will be scaled by 0.9.

Search online for "normalized cross correlation" (NCC). Implement and test NCC, and compare to your previous correlation results.