Project 1 Writeup: Hybrid Images

Sherwyn Braganza

October 4, 2022

Summary of Contents

- The Aim of the Project
- The Approach and Algorithm
- The Results
- Project Document Questions
- Appendices

1 The Project

The aim of this project was to learn about Hybrid Images, as described by Oliva, Torralba and Schyns in SIGGRAPH06. As showed in their report, the human eye tends to focus more on high frequency stuff when the object is nearby and focus on low frequency stuff when the the image is far. This can be exploited to create Hybrid Images - images that appear different when they are nearby as to when they are far. Blending filtered high-frequency content from one image on filtered low-frequency data from another image, one can create a Hybrid Image as show below (Fig 1). The following document explores this method of creating hybrid images using two methods - Spatial Domain Convolution and Fourier Domain Manipulation.

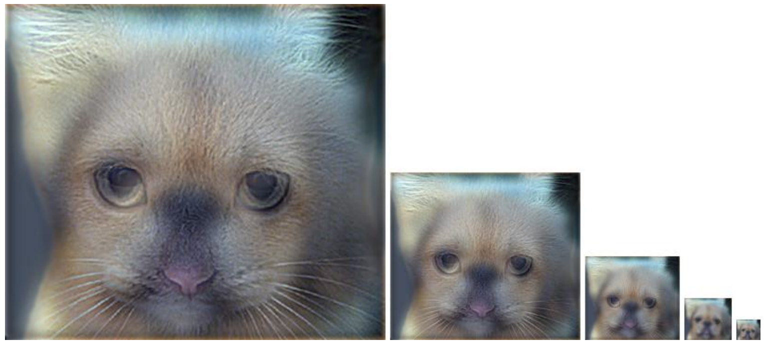

Figure 1: A hybrid image

2 The Algorithm and Approach

As mentioned in the document by Oliva et al., main process of generating hybrid images can be summarized in Equation 1 (shown below).

$$H = I_1 \circledast G_1 + I_2 \circledast (1 - G_2) \tag{1}$$

$I_1 \circledast G_1$ corresponds to the first image convoluted with a Gaussian kernel G_1, which ends up being the low-pass filtered version of the first image. $I_2 \circledast (1 - G_2)$ corresponds to the second image being convoluted with $I_2 \circledast (1 - G_2)$ which ends up being the high-pass version of the second image. This equation represents the spatial domain method for generating the hybrid image, in the Fourier Domain, the convolution operator just turns into a normal multiplication operator. G_1 and G_2 are Gaussian Filters generated from two different values of sigma or cut-off Frequencies (Fourier Domain linguistics). The generateGaussianKernel function computes these Gaussians and its code can be found in Appendix III.

2.1 Spatial Domain Convolution

As mentioned above, the Hybrid image in the Spatial Domain is generated following Equation 1. In my implementation, I exploit the fact that image convolution with a kernel is the same as image correlation with that exact same kernel, except its flipped vertically and horizontally. The following code snippet shows that exact implementation.

```python
def imConvolute(image: np.ndarray, kernel: np.ndarray) -> np.ndarray:
    if kernel.shape[0] % 2 == 0 or kernel.shape[1] % 2 == 0:
        raise Exception('Kernel with even dimensions provided.')

    # flip the kernel along rows and cols
    kernel = np.flip(np.flip(kernel, axis=1), axis=0)

    # call correlation function and get the kernel correlated image as well as the impulse version of it
    filtered = imCorrelate(image, kernel)

    return filtered
```

The listing below shows my Image Correlation Function. It was designed to work almost exactly like signal.correlate2D. My correlate function implements correlation by linearly moving the kernel over each pixel and calculating the element wise product of the kernel element and the corresponding pixel it overlaps. The sum of these element products is the convolution result for that pixel (lines 28 - 33). Before doing that, it pads the image with 0(s) based on the kernel size (line 15 - 20). The amount to be fairly easy to compute - its basically the half the number of rows and columns rounded down to the lower integer(line 16).

```python
1  def imCorrelate(image: np.ndarray, kernel: np.ndarray) -> np.ndarray:
2       if kernel.shape[0] % 2 == 0 or kernel.shape[1] % 2 == 0:
3          raise Exception('Kernel with even dimensions provided.')
4
5      # if grayscale or colored
6      color = True if len(image.shape) > 2 else False
7
8      image = skimage.img_as_float32(image)  # convert to floats in [0,1] to make computations uniform
9
10     # if grayscale, create a third dimension with only one channel
11     if not color:
12         image = image.reshape(image.shape[0], image.shape[1], 1)
13
14     # Padding section
15     pad_row, pad_col = kernel.shape[0] // 2, kernel.shape[1] // 2  # calculate pad_width for rows and cols
16     pad_params = ((pad_row, pad_row), (pad_col, pad_col), (0, 0))
17     padded_img = np.pad(image,
18                         pad_params,
19                         mode='constant',
20                         constant_values=0)  # pad image along rows and cols but not channels (if it exists)
21
22     # create a container for the kernel filtered image
23     filtered = np.zeros(image.shape)
24
25     # Check if it is an rgb or grayscale img
26     channel = 3 if color else 1
27
28     for i in range(pad_row, image.shape[0] + pad_row):
29         for j in range(pad_col, image.shape[1] + pad_col):
30             for k in range(channel):
31                 filtered[i - pad_row, j - pad_col, k] = np.sum(
32                     padded_img[i - pad_row:i + pad_row + 1, j - pad_col:j + pad_col + 1,
33                     k] * kernel)  # convolution step
34
35     # clip images and convert them back to ubytes before returning
36
37     return skimage.img_as_ubyte(filtered.clip(0, 1)) if color \
38         else skimage.img_as_ubyte(filtered.clip(0, 1))[:, :, 0]
```

2.2 Fourier Domain Manipulation

The Algorithm followed in this method implements Equation 1 but in the Fourier. Both the original image and the hybrid image are converted to the Fourier representations of themselves using numpy's fft function (lines 20 - 22). Fc corresponds to the Forier Domain representation of th image and Fk corresponds to that of the kernel. After multiplying them both, the result is then converted back to the Spatial Domain using the ifft2 function.

```python
1  def fourierDomain(image, kernel):
2      if kernel.shape[0] % 2 == 0 or kernel.shape[1] % 2 == 0:
3          raise Exception('Kernel with even dimensions provided.')
4
5      color = True if len(image.shape) > 2 else False
6
7      # if grayscale, create a third dimension with only one channel
8      if not color:
9          image = image.reshape(image.shape[0], image.shape[1], 1)
10
11     image = skimage.img_as_float32(image)
12     pad_values = (image.shape[0] - kernel.shape[0]) // 2, (image.shape[1] - kernel.shape[1]) // 2
13     padded_kernel = np.zeros(image.shape[0:2])
14     padded_kernel[pad_values[0]: pad_values[0] + kernel.shape[0], pad_values[1]: pad_values[1] + kernel.shape[1]
15                 ] = kernel
16
17     output_img = np.zeros(image.shape)
18
19     for i in range(image.shape[2]):
20         Fc = np.fft.fft2(np.fft.ifftshift(image[:, :, i]))
21         Fk = np.fft.fft2(np.fft.ifftshift(padded_kernel))
22         output_img[:, :, i] = np.abs(np.fft.ifftshift(np.fft.ifft2(Fc * Fk)))
23
24     return skimage.img_as_ubyte(output_img) if color else skimage.img_as_ubyte(output_img)[:, :, 0]
```

np.fft.fft2 expects a shifted version of the image, hence the initial ifftshift before the conversion. It then calculates the 2 Dimensional FFT on the image and returns data with the DC part to the exterior. Inorder to visualize the data an fftshift must be performed on the data before plotting it to shift the DC component to the center. The syntax for using this function can be found here. The inverse of fft2 - ifft2, which converts the Fourier Data back to Spacial Representation expects shifted data, hence no fftshift is performed there. The result requires an extra ifftshift before it is plotted.

3 Results

3.1 my_imfilter() on different Kernels

Figure 2 and Figure 3 show the results of my_imfilter() function. It performs image convolution in the Spatial Domain using different kernels. The first image is the original, followed by the Impulse response Filter, Sobel Filter, Sharpen Filter, Emboss Filter and a Gaussian Filter. The function was also tested on an non-square kernel. The output can be found in tests/unevenkernel.jpg.

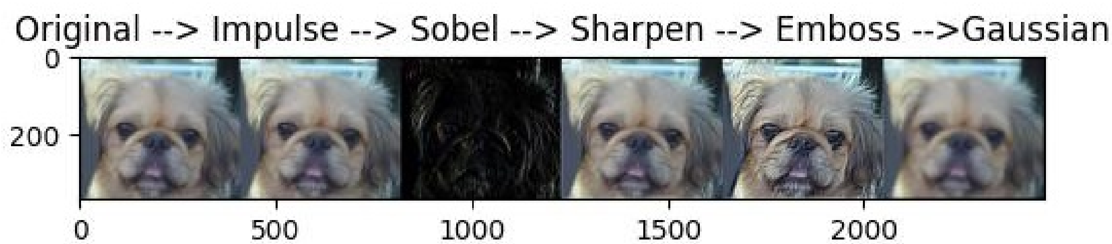

Figure 2: my_imfilter on a color image

Figure 3: my_imfilter on a grayscale image

3.2 Fourier Domain Tests

This section shows the results achieved by manipulating an image with a Guassian Kernel in the Fourier Domain, at every step. Figure 4 and Figure 5 show the results from using my fourierDomain() function to perform convolution in the fourier Domain. Check fourierTests() in Appendix I for the code. The first image is the original image, the second is the Fourier Domain Spectra after calling fft2 and fftshift. The third is the Fourier Domain Spectra of the image with a Gaussian Filter (low pass) imposed on it by multiplication in the Fourier Domain. The fourth is the image converted back to the spatial domain. Our results match our expectations, in which we expect to see only low frequencies.

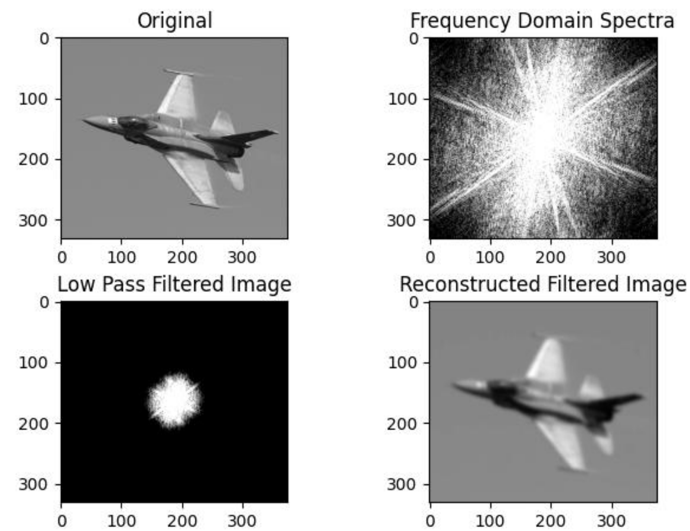

Figure 4: Fourier Domain Manipulation of plane.bmp

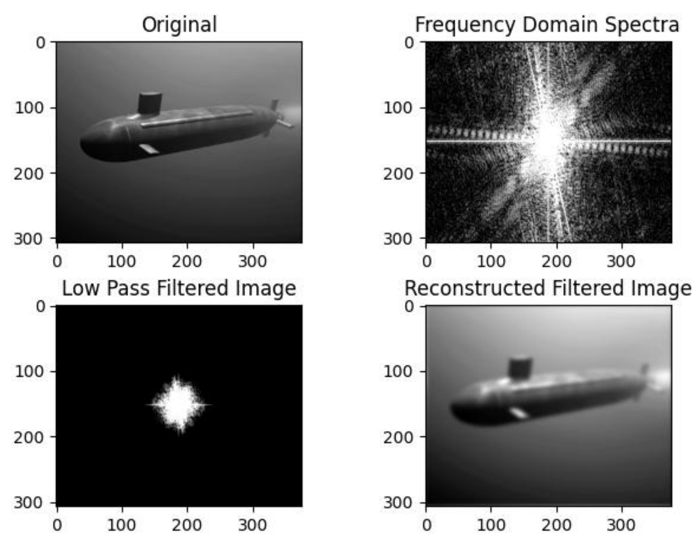

Figure 5: Fourier Domain Manipulation of submarine.bmp

3.3 Hybrid Image Generation in the Spatial Domain

This section shows the results of generating a hybrid image in the Spatial Domain. Appendix III contains the full code structure to achieve this. I have a general hybridise function and based on the value of the boolean variable fourier, it decides whether to use the Spatial Domain Method or the Fourier Domain Method. Figure 6 shows the results achieved in generating a hybrid image in the Spatial domain using the images cat.bmp and dog.bmp.

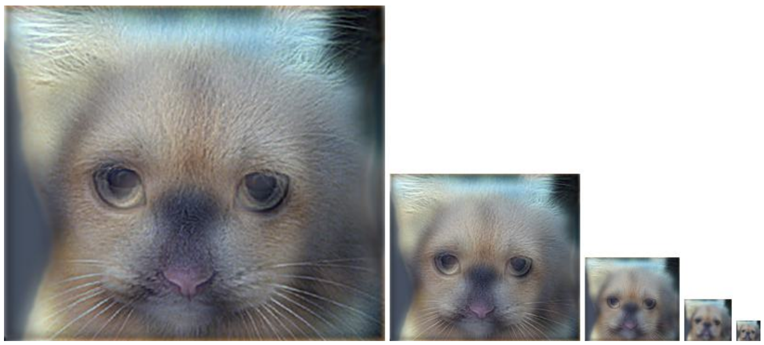

Figure 6: Cat + Dog = Cog Hybrid Image

3.4 Hybrid Image Generation in the Fourier Domain

Like in the previous subsection, a hybrid image was generated using the same hybridise function albeit with fourier=True. This generated the hybrid image using the Fourier Domain Method. The image of Albert Einstein and Marilyn Monroe were melded to form the hybrid image Albert Monroe (Figure 7). From our experiments in Question 4, we saw that the Fourier Domain Method was much faster. More hybrid images were generated using this method and they can be found in Appendix IV.

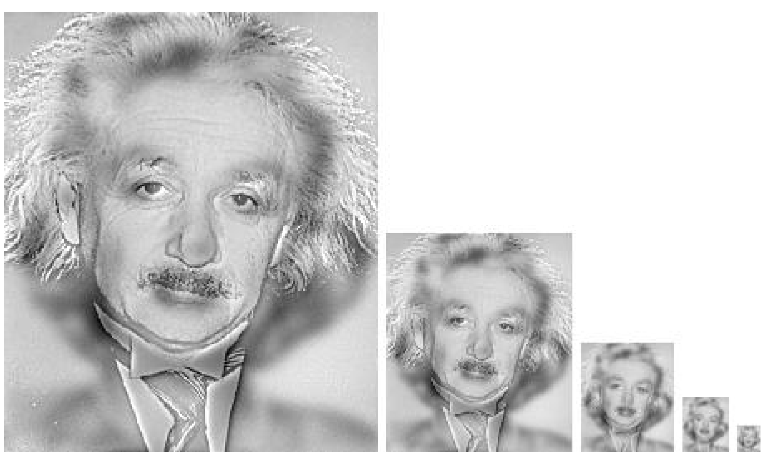

Figure 7: Albert Monroe

4 Project Document Questions

Explicitly describe image convolution: the input, the transformation and the output. Why is it useful in Computer Vision?

Mathematically, the convolution operation is performed by taking the element-wise product of the first matrix with the matrix (that is flipped along its columns and rows) at each element to form a new matrix as shown below(Figure 8). The resulting matrix is usually smaller than the original and is dependent on the second matrix.

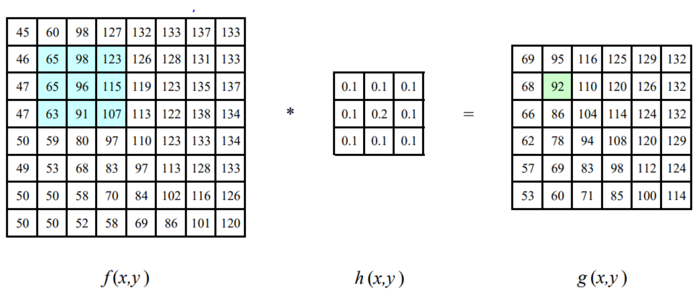

Figure 8: Convolution

Image convolution involves 2 data matrices - the image and the kernel. The image input is usually padded based on the kernel size so that the output is the same size as the original image. Convolution works in the same was as it does mathematically, each pixel value is transformed into a new value that is dependent on the pixels surrounding it. In the Fourier domain convolution turns into multiplication, which is super useful because one can visualize the kernel acting as a filter to filter out unwanted frequencies(data) of the image as seen in Figure 4 and Figure 5. This is one of the main and most useful features of convolution of images - acting as a filter to filter out unwanted frequencies (best visualized in the fourier domain).

This is super useful in Computer Vision as kernels can be designed to pick up only certain features (and frequencies) from an image, like edges and curves (Sobel Filters) whos data can then be used in Image Recognition Software or to train Deep Neural Networks (CNNs).

What is the difference between convolution and correlation? Construct a scenario which produces a different output between both operations.

Correlation and Convolution are very similar. They implement the same element-wise multiplication as explained above - the only difference is that convolution does it with the vertically and horizontally flipped version of the kernel. This results in similar, but slightly different outcomes.

In most cases where the kernel is a symmetrical matrix, convolution and correlation are the same as shown below (Table 1).

However when the kernel is not symmetrical (Sobel Filter), the kernels end up being a little different as seen in Table 2

0	-1	0		0	-1	0
-1	5	-1		-1	5	-1
0	-1	0		0	-1	0

Table 1: *Left:* Correlation ; *Right:* Convolution

-1	0	1		1	0	-1
-2	0	2		2	0	-2
-1	0	1		1	0	-1

Table 2: *Left:* Correlation ; *Right:* Convolution

What is the difference between a high pass filter and a low pass filter in how they are constructed, and what they do to the image? Please provide example kernels and output images.
High Pass Filters allow only high frequency data through while blocking low frequency data; a Low Pass Filter is the converse of that.

A common low pass filter is a Gaussian or Normal Distribution Filter with a mean of 0. The Fourier Spectrum of this Image turns out to be a circle with 0s on the outside and positive values in the inside, which when multiplied with another image creates a low pass filtered version of that image as we know that all lower frequency content lies close to the center. (shown below and in Figure 4).

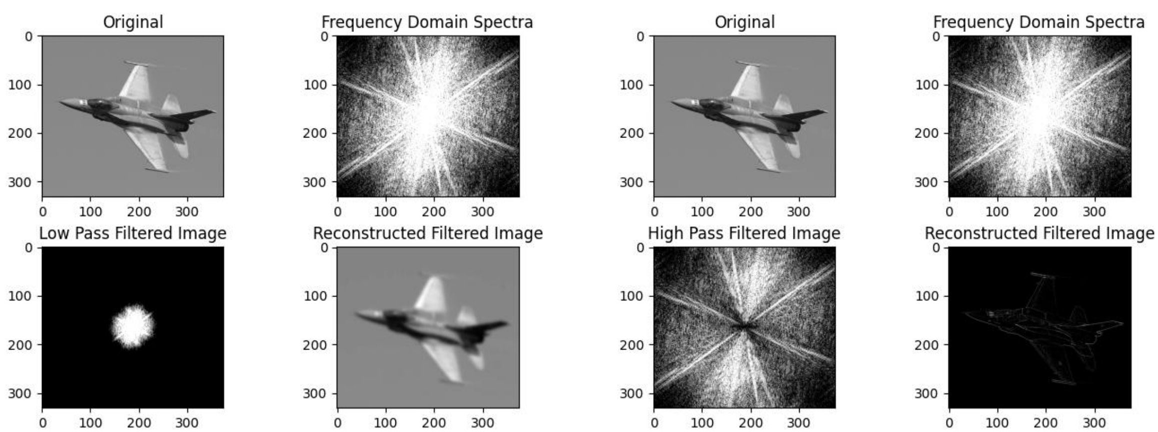

Figure 9: Low Pass and High Pass Filtered plane.bmp

Subtracting this newly obtained low frequency image from the original image in the Fourier Domain, results in a high frequency version of that image. From this we can infer that a high pass filter is just $1 - G$ where G is a low-pass Gaussian Filter. The figure above shows this in play (Figure 9).

How does computation time vary with filter sizes from 3×3 to 15×15 (for all odd and square sizes), and with image sizes from 0.25 MPix to 8 MPix (choose your own intervals - you can use the image in project 0 if you'd like)? Measure both using imf ilter to produce a matrix of values. Use the imresize function to vary the size of an image. Use an appropriate charting function to plot your matrix of results, such as scatter3 or surf . Do the results match your expectation given the number of multiply and add operations in convolution?

For the most part the results matched my expectations - the computation time should increase non linearly with the increase in image size and kernel size. My results are show below in Figure 10.

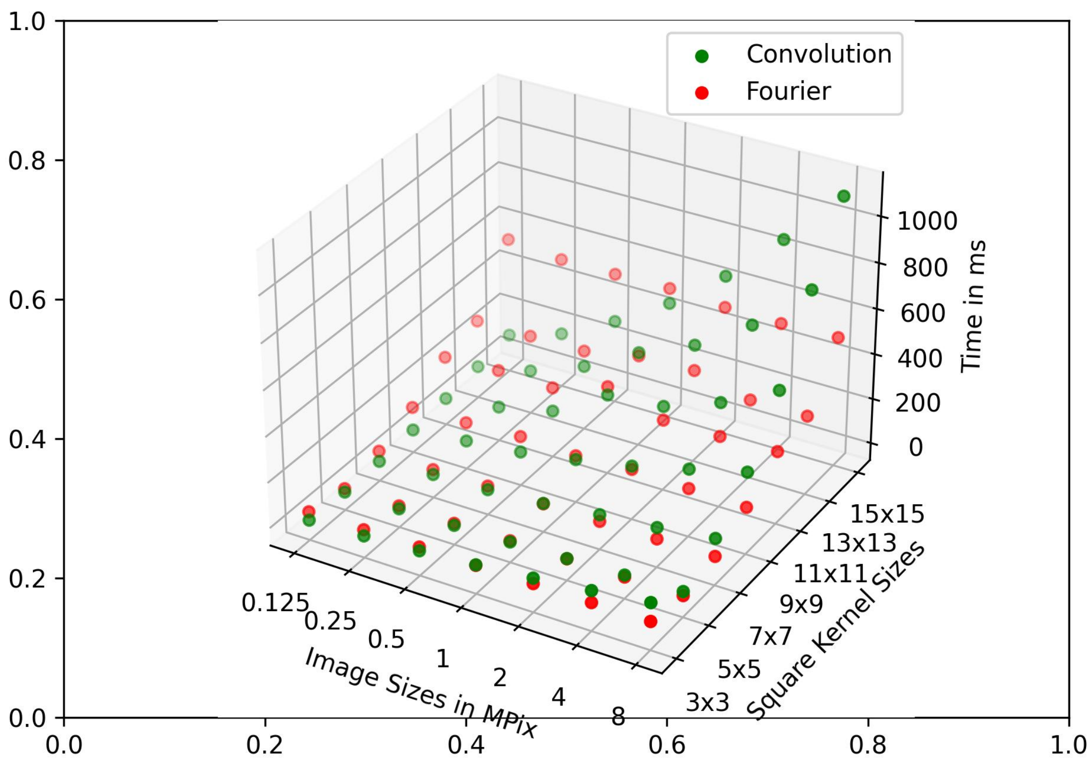

Figure 10: Image Size and Kernel Size Computation Metrics

Apart from plotting Convolution computation times, I also plotted Fourier Domain Manipulation Convolution times. Fourier Domain Manipulation performed almost as good Convolution for smaller kernels and smaller images but outshone convolution with when the image sizes and kernel sizes got bigger. However what was surprising was that Convolution performed better than Fourier Domain Manipulation when sizes of images were small but kernels were big.

5 Appendix I

```python
import os
import numpy as np
import skimage
from skimage import io
import matplotlib.pyplot as plt

IMPULSE = np.asarray([[0,0,0],
                      [0,1,0],
                      [0,0,0]])
SOBEL = np.asarray([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]])
UNEVEN_SOBEL = np.asarray([[-2, -1, 0, 1,2],
                    [-4, -2, 0, 2, 4],
                    [-2, -1, 0, 1, 2]])
SHARPEN = np.array([[0, -1, 0],
                    [-1, 5, -1],
                    [0, -1, 0]])
EMBOSS = np.array([[-2, -1, 0],
                   [-1, 1, 1],
                   [0, 1, 2]])


def my_imfilter(image: np.ndarray, kernel: np.ndarray) -> (np.ndarray, np.ndarray):
    """
        Wrapper Function for imConvolute. Created to meet assignment
        document specifications
    """
    return imConvolute(image, IMPULSE), imConvolute(image, kernel)


def imConvolute(image: np.ndarray, kernel: np.ndarray) -> np.ndarray:
    """
        Image Convolution Filter

        Author: Sherwyn Braganza
        Sept 28, 2022 - Initial Creation
        Sept 28, 2022 - Added more descriptive comments.
        Sept 28, 2022 - Finished off most of convolve
        Sept 28, 2022 - Compared results with signal.convolve2D
        Sept 30, 2022 - Changes made to return only filtered image

        Function that implements convolution through correlation. Designed to work
        like signal.convolve2D from the scipy library. The kernel used for convolution
        is presented by the user. This function is designed to handle both rgb and grayscale images.

        It exploits the fact that convolution is basically correlation using a flipped kernel.
        It therefore flips the kernel and passes it to the function that gives the correlational
        version of an image with the flipped kernel.

        The function returns a np.ndarray, corresponding to the image correlated with the kernel provided.

            :param image: Numpy matrix containing image data
            :param kernel: Numpy matrix containing data for an
                                            odd dimension kernel

            :return filter: Kernel Convolved Version of the Image
    """
    if kernel.shape[0] % 2 == 0 or kernel.shape[1] % 2 == 0:
        raise Exception('Kernel with even dimensions provided.')

    # flip the kernel along rows and cols
    kernel = np.flip(np.flip(kernel, axis=1), axis=0)

    # call correlation function and get the kernel correlated image as well as the impulse version of it
    filtered = imCorrelate(image, kernel)

    return filtered


def imCorrelate(image: np.ndarray, kernel: np.ndarray) -> np.ndarray:
    """
        Image Correlation Filter

        Author: Sherwyn Braganza
        Sept 28, 2022 - Initial Creation
        Sept 28, 2022 - Added more descriptive comments.
        Sept 28, 2022 - Finished off most of correlate
        Sept 28, 2022 - Compared results with signal.correlate2D
        Sept 30, 2022 - Changes made to return only filtered image
        Setp 30, 2022 - Increased compatibility with grayscales

        Function that implements correlation image processing. Designed to work
        like signal.correlate2D from the scipy library. The kernel used for correlation
        is presented by the user.

        This function convolves the kernel with the image using matrix convolution. It
        initially converts the image to a float and then pads it with 0s along the borders
```

```
        according to the shape of the convolutional kernel. It finally clips the image pixel values to [0,1]
        before converting it back to ubyte format and returning it.

        The function returns a np.ndarray, corresponding to the image correlated with the kernel provided.

        TODO - Try to use logical indexing insted of loops

            :param image: Numpy matrix containing image data
            :param kernel: Numpy matrix containing data for an odd dimension kernel

            :return filter: Kernel Correlated Version of the Image
    """
    if kernel.shape[0] % 2 == 0 or kernel.shape[1] % 2 == 0:
        raise Exception('Kernel with even dimensions provided.')

    # if grayscale or colored
    color = True if len(image.shape) > 2 else False

    image = skimage.img_as_float32(image)  # convert to floats in [0,1] to make computations uniform

    # if grayscale, create a third dimension with only one channel
    if not color:
        image = image.reshape(image.shape[0], image.shape[1], 1)

    # Padding section
    pad_row, pad_col = kernel.shape[0] // 2, kernel.shape[1] // 2  # calculate pad_width for rows and cols
    pad_params = ((pad_row, pad_row), (pad_col, pad_col), (0, 0))
    padded_img = np.pad(image,
                        pad_params,
                        mode='constant',
                        constant_values=0)  # pad image along rows and cols but not channels (if it exists)

    # create a container for the kernel filtered image
    filtered = np.zeros(image.shape)

    # Check if it is an rgb or grayscale img
    channel = 3 if color else 1

    for i in range(pad_row, image.shape[0] + pad_row):
        for j in range(pad_col, image.shape[1] + pad_col):
            for k in range(channel):
                filtered[i - pad_row, j - pad_col, k] = np.sum(
                    padded_img[i - pad_row:i + pad_row + 1, j - pad_col:j + pad_col + 1,
                    k] * kernel)  # convolution step

    # clip images and convert them back to ubytes before returning

    return skimage.img_as_ubyte(filtered.clip(0, 1)) if color \
        else skimage.img_as_ubyte(filtered.clip(0, 1))[:, :, 0]


def hybridise(image1: np.ndarray, image2: np.ndarray, sigma1: float, sigma2: float, fourier: bool):
    """
        Hybrid Image Generator

        Generates a hybrid image according to the process described by Oliva, Torralba and Schyns
        in Siggraph(2006) (http://olivalab.mit.edu/hybrid/Talk_Hybrid_Siggraph06.pdf).

        The basis of this process is to take the high pass version of one image and
        superimpose it on the low pass version of the other. These images have to be normalized
        and centered for best results. This function implements the same process
        using 2 approaches - Spatial Convolution and Fourier Domain Multiplication (Hadamard Product)

        The function generates 2 different Gaussian Filters based on the sigmas provided and
        uses them to get a low pass and high pass filtered images

        The generated hybrid image is rescaled 4 times and stacked horizontally to get the
        final image that is then returned.

        Author: Sherwyn Braganza
        29 Sept, 2022 - Implemented a rudimentary hybrid image generator
                        that uses Spatial Convolution
        30 Sept, 2022 - Implemented hybrid image rescaling and stacking
        30 Sept, 2022 - Expanded it to use Spatial Convolution and
                        Fourier Domain Hadamard (fourier domain still need to be coded up)

        :param image1: The low pass intended image
        :param image2: The high pass intended image
        :param sigma1: The sigma value corresponding to the low pass image
        :param sigma2: The sigma value corresponding to the high pass image
        :param fourier: False if you want to use Spatial Convolution, True
                        if you want to use Fourier Domain Processing

        :return: The stacked hybrid image
    """
    hybrid = np.asarray([])
    gaussian_low = generateGaussianKernel(sigma1)
    gaussian_high = generateGaussianKernel(sigma2)
    lowpass_image = lowpass_image2 = None


    if not fourier:
```

```python
        lowpass_image = imConvolute(image1, gaussian_low)
        lowpass_image2 = imConvolute(image2, gaussian_high)
    else:
        lowpass_image = fourierDomain(image1, gaussian_low)
        lowpass_image2 = fourierDomain(image2, gaussian_high)

    hybrid = skimage.img_as_ubyte((
                                skimage.img_as_float32(lowpass_image) +
                                skimage.img_as_float32(image2) -
                                skimage.img_as_float32(lowpass_image2)
                                ).clip(0, 1))

    # if grayscale, create a third dimension with only one channel
    if len(hybrid.shape) < 3:
        hybrid = hybrid.reshape(hybrid.shape[0], hybrid.shape[1], 1)

    ######################################
    # Image stacking and padding section
    ######################################
    image_stack = [hybrid]

    # create scale down versions of the original
    for i in range(0, 4):
        image_stack.append(skimage.img_as_ubyte(
            skimage.transform.rescale(image_stack[i], 0.5, anti_aliasing=True, channel_axis=2)
        ))

    # padding the rescaled images along axis 0 to make them the same size vertically
    for i in range(1, 5):
        image_stack[i] = np.pad(
            image_stack[i],
            ((image_stack[0].shape[0] - image_stack[i].shape[0], 0),
             (0, 0),
             (0, 0)),
            mode='constant',
            constant_values=255
        )

    # padding along axis 1
    for i in range(1, 5):
        image_stack[i] = np.pad(
            image_stack[i],
            ((0, 0),
             (5, 0),
             (0, 0)),
            mode='constant',
            constant_values=255
        )

    return np.hstack(image_stack) if len(image1.shape) > 2 else np.hstack(image_stack)[:,:,0]


def fourierDomain(image, kernel):
    """
        Fourier Domain Image Convolution

        Author: Sherywn Braganza

        Implements image convolution by shifting the image and kernel into the Frequency Domain, computing
        the Hadamard Product of them both and then converting them back to the spacial domain using the
        inverse fourier transform.

        :param image: The image to be convolved
        :param kernel: The kernel to be convolved
        :return: The convolved image and kernel
    """
    if kernel.shape[0] % 2 == 0 or kernel.shape[1] % 2 == 0:
        raise Exception('Kernel with even dimensions provided.')

    color = True if len(image.shape) > 2 else False

    # if grayscale, create a third dimension with only one channel
    if not color:
        image = image.reshape(image.shape[0], image.shape[1], 1)

    image = skimage.img_as_float32(image)
    pad_values = (image.shape[0] - kernel.shape[0]) // 2, (image.shape[1] - kernel.shape[1]) // 2
    padded_kernel = np.zeros(image.shape[0:2])
    padded_kernel[pad_values[0]: pad_values[0] + kernel.shape[0], pad_values[1]: pad_values[1] + kernel.shape[1]
                ] = kernel

    output_img = np.zeros(image.shape)

    for i in range(image.shape[2]):
        Fc = np.fft.fft2(np.fft.ifftshift(image[:, :, i]))
        Fk = np.fft.fft2(np.fft.ifftshift(padded_kernel))
        output_img[:, :, i] = np.abs(np.fft.ifftshift(np.fft.ifft2(Fc * Fk))).clip(-1,1)

    return skimage.img_as_ubyte(output_img) if color else skimage.img_as_ubyte(output_img)[:, :, 0]


def generateGaussianKernel(sigma: float) -> np.ndarray:
```

```
    """
        Generates a 2-D Gaussian Kernel

        Author: Sherwyn Braganza
        Sept 29, 2020 - Added function and base code for generating it
        Sept 29, 2020 - Implemented a weighted mean based kernel generator
        Sept 30, 2020 - Changed implementation to generate a true gaussian
                        based kernel

        Generates a 1D Gaussian distribution from the Gaussian equation
        using the value of sigma. Matrix multiplies the transpose of
        the 1D Gaussian with itself to form a 2D square Gaussian kernel

        :param sigma: The standard deviation of the gaussian
        :return: kernel: The 2D Gaussian kernel generated in float format
    """
    size = int(8 * sigma + 1)
    # enforce an odd sized kernel
    if not size % 2:
        size = size + 1

    center = size // 2
    kernel = np.zeros(size)

    # Generate Gaussian blur.
    for x in range(size):
        diff = (x - center) ** 2
        kernel[x] = np.exp(-diff / (2 * sigma ** 2))

    kernel = np.asarray(kernel.reshape(-1, 1).T * kernel.reshape(-1, 1))
    kernel = kernel / np.sum(kernel)

    return kernel


def testConvolutionColor():
    """
        Script to test out Colored Image Convolution

        Author: Sherwyn Braganza

        :param: NONE
        :return: NONE
    """

    img1 = io.imread('data/dog.bmp', as_gray=False)
    GAUSSIAN = generateGaussianKernel(3)
    img_impulse = imConvolute(img1, IMPULSE)
    img_sobel = imConvolute(img1, SOBEL)
    img_sharpen = imConvolute(img1, SHARPEN)
    img_emboss = imConvolute(img1, EMBOSS)
    img_gaussian = imConvolute(img1, GAUSSIAN)

    joined = np.hstack((img1, img_impulse, img_sobel, img_sharpen, img_emboss, img_gaussian))
    fig, axs = plt.subplots()
    axs.set_title('Original --> Impulse --> Sobel --> Sharpen --> Emboss -->Gaussian')
    axs.imshow(joined)
    fig.savefig('tests/my_filter_test_colored.jpg')


def testConvolutionGray():
    """
        Script to test out Colored Grayscale Convolution

        Author: Sherwyn Braganza

        :param: NONE
        :return: NONE
    """

    img1 = skimage.img_as_ubyte(io.imread('data/marilyn.bmp', as_gray=True))
    GAUSSIAN = generateGaussianKernel(3)
    img_impulse = imConvolute(img1, IMPULSE)
    img_sobel = imConvolute(img1, SOBEL)
    img_sharpen = imConvolute(img1, SHARPEN)
    img_emboss = imConvolute(img1, EMBOSS)
    img_gaussian = imConvolute(img1, GAUSSIAN)

    joined = np.hstack((img1, img_impulse, img_sobel, img_sharpen, img_emboss, img_gaussian))
    fig, axs = plt.subplots()
    axs.set_title('Original --> Impulse --> Sobel --> Sharpen --> Emboss -->Gaussian')
    axs.imshow(joined, cmap='gray')
    fig.savefig('tests/my_filter_test_gray.jpg')


def testFFT(image_name):
    """
        Test image processing in the Fourier Space

        Coverts the image to its fourier space representation, applies a gaussian
        filter to it and then converts it back to the spatial domain. Saves images
        at each of these steps.
```

```

        :param image_name: Name of the image to test
        :return: None
    """
    image = io.imread(image_name, as_gray=True)
    fig, axs = plt.subplots(2,2)
    fig.tight_layout(h_pad=2)
    plt.subplots_adjust(top=0.9)
    padded_gaussian = np.zeros(image.shape)
    gaussian = generateGaussianKernel(3)
    pad_params = (image.shape[0]-gaussian.shape[0])//2, (image.shape[1]-gaussian.shape[1])//2
    padded_gaussian[pad_params[0]:pad_params[0]+gaussian.shape[0],
                    pad_params[1]:pad_params[1]+gaussian.shape[1]] = gaussian

    axs[0,0].imshow(image, cmap='gray')
    axs[0,0].set_title('Original')

    img_fft = np.fft.fft2(np.fft.ifftshift(image))
    axs[0,1].imshow(skimage.img_as_ubyte(np.real(np.log10(np.fft.fftshift(img_fft))).clip(-1,1)), cmap='gray')
    axs[0,1].set_title('Frequency Domain Spectra')

    filtered_fft = img_fft * np.fft.fft2(np.fft.ifftshift(padded_gaussian))
    axs[1,0].imshow(skimage.img_as_ubyte(np.real(np.log10(np.fft.fftshift(filtered_fft))).clip(-1,1)), cmap='gray')
    axs[1,0].set_title('Low Pass Filtered Image')

    reconstructed = np.real(np.fft.ifftshift(np.fft.ifft2(filtered_fft))).clip(-1,1)
    axs[1,1].imshow(skimage.img_as_ubyte(reconstructed), cmap='gray')
    axs[1,1].set_title('Reconstructed Filtered Image')

    fig.savefig('tests/fft_transformed_'+image_name[5:-3]+'jpg')


def testUnevenKernel():
    # Uneven Kernel Test
    img1 = skimage.img_as_ubyte(io.imread('data/marilyn.bmp', as_gray=True))
    convolved = imConvolute(img1, UNEVEN_SOBEL)
    fig, axs = plt.subplots()
    axs.set_title('Uneven Kernel')
    axs.imshow(convolved, cmap='gray')
    fig.savefig('tests/unevenkernel.jpg')


if __name__ == '__main__':
    # check if directories exist
    if not os.path.exists('data'):
        os.makedirs('data')
    if not os.path.exists('tests'):
        os.makedirs('tests')

    testConvolutionColor()
    testConvolutionGray()
    testFFT("data/submarine.bmp")
    testFFT("data/plane.bmp")
    testUnevenKernel()
```

6 Appendix II

```python
def generateGaussianKernel(sigma: float) -> np.ndarray:
    """
        Generates a 2-D Gaussian Kernel

        Author: Sherwyn Braganza
        Sept 29, 2020 - Added function and base code for generating it
        Sept 29, 2020 - Implemented a weighted mean based kernel generator
        Sept 30, 2020 - Changed implementation to generate a true gaussian
                        based kernel

        Generates a 1D Gaussian distribution from the Gaussian equation
        using the value of sigma. Matrix multiplies the transpose of
        the 1D Gaussian with itself to form a 2D square Gaussian kernel

        :param sigma: The standard deviation of the gaussian
        :return: kernel: The 2D Gaussian kernel generated in float format
    """
    size = int(8 * sigma + 1)
    # enforce an odd sized kernel
    if not size % 2:
        size = size + 1

    center = size // 2
    kernel = np.zeros(size)

    # Generate Gaussian blur.
    for x in range(size):
        diff = (x - center) ** 2
        kernel[x] = np.exp(-diff / (2 * sigma ** 2))

    kernel = np.asarray(kernel.reshape(-1, 1).T * kernel.reshape(-1, 1))
    kernel = kernel / np.sum(kernel)

    return kernel
```

7 Appendix III

```
def hybridise(image1: np.ndarray, image2: np.ndarray, sigma1: float, sigma2: float, fourier: bool):
hybrid = np.asarray([])
    gaussian_low = generateGaussianKernel(sigma1)
    gaussian_high = generateGaussianKernel(sigma2)
    lowpass_image = lowpass_image2 = None


    if not fourier:
        lowpass_image = imConvolute(image1, gaussian_low)
        lowpass_image2 = imConvolute(image2, gaussian_high)
    else:
        lowpass_image = fourierDomain(image1, gaussian_low)
        lowpass_image2 = fourierDomain(image2, gaussian_high)

    hybrid = skimage.img_as_ubyte((
                                 skimage.img_as_float32(lowpass_image) +
                                 skimage.img_as_float32(image2) -
                                 skimage.img_as_float32(lowpass_image2)
                                 ).clip(0, 1))

    # if grayscale, create a third dimension with only one channel
    if len(hybrid.shape) < 3:
        hybrid = hybrid.reshape(hybrid.shape[0], hybrid.shape[1], 1)

    #######################################
    # Image stacking and padding section
    #######################################
    image_stack = [hybrid]

    # create scale down versions of the original
    for i in range(0, 4):
        image_stack.append(skimage.img_as_ubyte(
            skimage.transform.rescale(image_stack[i], 0.5, anti_aliasing=True, channel_axis=2)
        ))

    # padding the rescaled images along axis 0 to make them the same size vertically
    for i in range(1, 5):
        image_stack[i] = np.pad(
            image_stack[i],
            ((image_stack[0].shape[0] - image_stack[i].shape[0], 0),
             (0, 0),
             (0, 0)),
            mode='constant',
            constant_values=255
        )

    # padding along axis 1
    for i in range(1, 5):
        image_stack[i] = np.pad(
            image_stack[i],
            ((0, 0),
             (5, 0),
             (0, 0)),
            mode='constant',
            constant_values=255
        )

    return np.hstack(image_stack) if len(image1.shape) > 2 else np.hstack(image_stack)[:,:,0]
```

8 Appendix IV

Figure 11: Bicycle + Motorcycle = Bimotorcycle

Figure 12: Plane + Bird = Plird

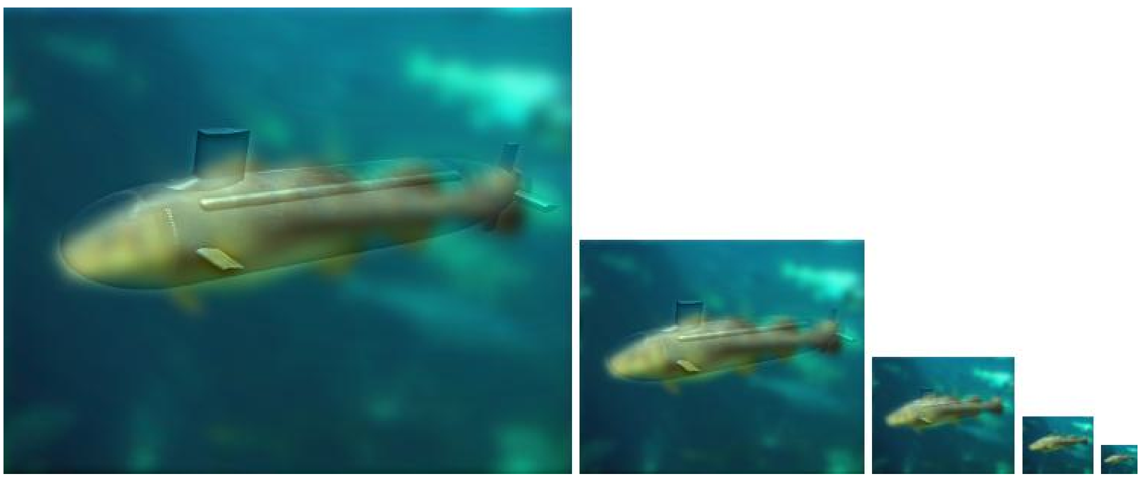

Figure 13: Fish + Submarine = Fishmarine