II Basic Operations in OpenCV

- **Task 1** (I'm using Jupyter Notebook)
- **Task 2**
  1. *Code:*

```
# Load Images using cv2
elephant_image = cv2.imread('/Users/fneba/elephant.jpeg',
cv2.IMREAD_COLOR)
```

  2. *Code:*

```
# Display Images using matplotlib pt. 1
plt.imshow(elephant_image)
plt.show()

# Write Image to 'elephant_opencv.png'
cv2.imwrite('elephant_opencv.png', elephant_image)
```

     Output:

  3. *Code:*

```
# Convert loaded image to RGB
elephant = cv2.cvtColor(elephant_image, cv2.COLOR_BGR2RGB)

# Display Images using matplotlib pt. 2
plt.imshow(elephant)
plt.show()

# Write Image to 'elephant_matplotlib.png'
cv2.imwrite('elephant_matplotlib.png', elephant)
```

     Output:

- **Task 3**
  1. *Code:*

```
# Convert recent result to grayscale
elephant_gray = cv2.cvtColor(elephant, cv2.COLOR_BGR2GRAY)

plt.imshow(elephant_gray, cmap='gray', vmin=0, vmax=255)
plt.axis('off')
plt.show()

# Write Image to 'elephant_gray.png'
cv2.imwrite('elephant_gray.png', elephant_gray)
```

Output:

  2. *Code:*

```
# Crop and display the baby elephant in regular color
elephant_cropped = elephant[400:950, 0:600]

plt.imshow(elephant_cropped)
plt.axis('off')
plt.show()

# Write Image to 'elephant_baby.png'
```

```
cv2.imwrite('elephant_baby.png', elephant_cropped)
```

Output:

3. Parts:
   a. *Code:*

```
# resizing, display, and write to 'elephant_10xdown.png'
elephant_10xdown = cv2.resize(elephant, None, fx=0.1, fy=0.1)
plt.imshow(elephant_10xdown)
plt.axis('off')
plt.show()
cv2.imwrite('elephant_10xdown.png', elephant_10xdown)
```

Output:

   b. *Code:*

```
# nearest neighbor
elephant_10xupNearest = cv2.resize(elephant_10xdown, None, fx=10,
fy=10, interpolation = cv2.INTER_NEAREST)
plt.imshow(elephant_10xupNearest)
plt.axis('off')
plt.show()
```

```
cv2.imwrite('elephant_10xup_NearestNeighbor.png',
elephant_10xupNearest)

# bicubic
elephant_10xupBicubic = cv2.resize(elephant_10xdown, None, fx=10,
fy=10, interpolation = cv2.INTER_CUBIC)
plt.imshow(elephant_10xupBicubic)
plt.axis('off')
plt.show()
cv2.imwrite('elephant_10xup_Bicubic.png', elephant_10xupBicubic)
```

Output:

Nearest Neighbor

Bicubic

c. *Code:*

```
# ground truth image and nearest neighbor resizing absolute diff and
write out
NNad = cv2.absdiff(elephant, elephant_10xupNearest)
cv2.imwrite('AbsDiff_Nearest.png', NNad)
```

```
# ground truth image and bicubic resizing absolute diff and write out
BCad = cv2.absdiff(elephant, BCad)
cv2.imwrite('AbsDiff_Bicubic.png', BCad)

# Sum for pixels in NNad
sum_NNad = cv2.sumElems(NNad)
numeric_sum_NNad = sum(sum_NNad[:3])

# Sum for pixels in BCad
sum_BCad = cv2.sumElems(BCad)
numeric_sum_BCad = sum(sum_BCad[:3])
```

Output:

Nearest Neighbor

Bicubic

The sum of the pixels from Nearest Neighbor was 46599821. And the sum of the pixels caused by Bicubic was 40261332. So Bicubic caused the least error in upsampling.

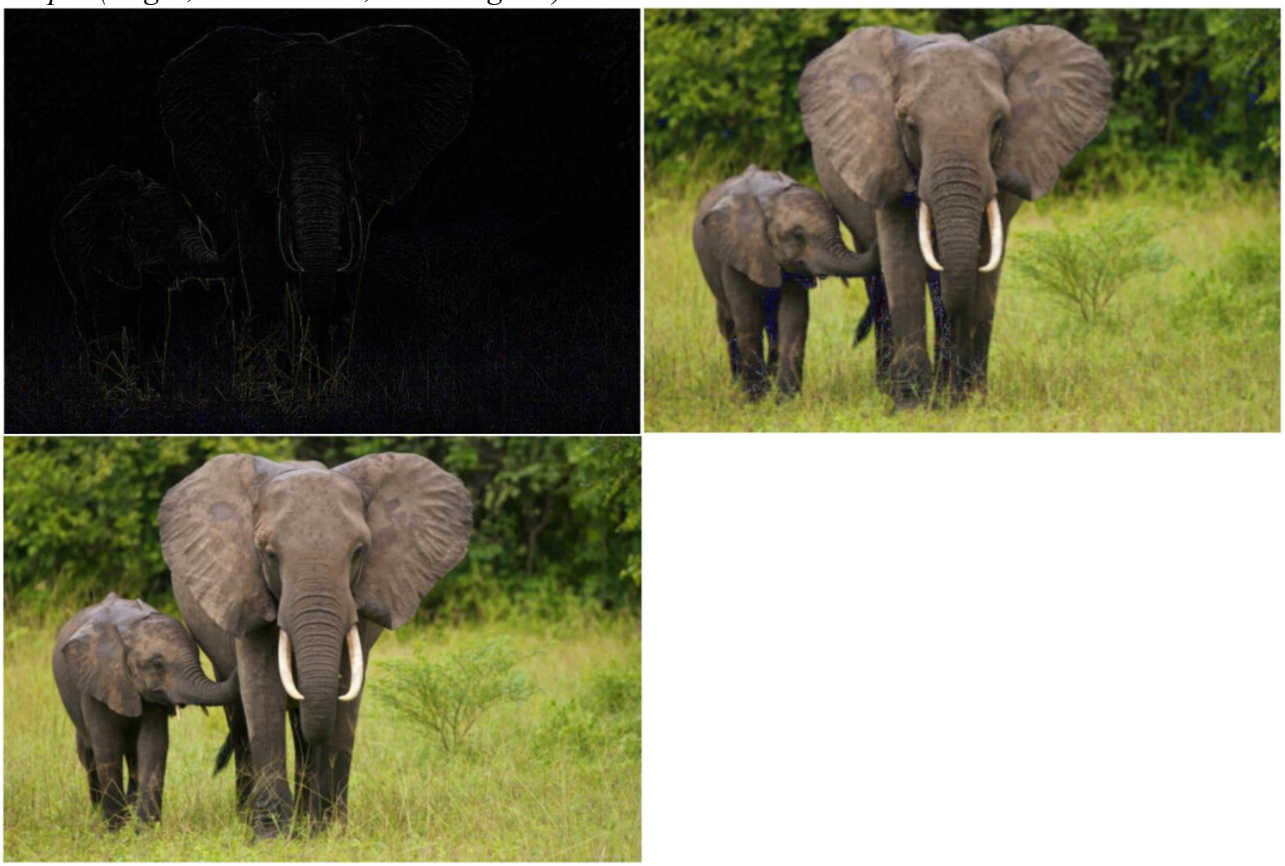
- **Task 4**
  1. *Code:*

```
edgeDetectKernel = np.array([[-1, -1, -1],
                             [-1, 8, -1],
                             [-1, -1, -1]])
def edgeDetect(image):

    # Apply the kernel to image using 2D convolution
    edges = cv2.filter2D(image, -1, edgeDetectKernel)

    return edges
```

2. *Code:*

```
blurKernel = np.array([[1, 2, 1],
                       [2, 4, 2],
                       [1, 2, 1]])/16
def blur(image):

    # Apply the kernel to image using 2D convolution
    blurred = cv2.filter2D(image, -1, blurKernel)

    return blurred
```

3. *Output (Edges, then Blurred, then Original):*



III Fourier Domain Image Blending

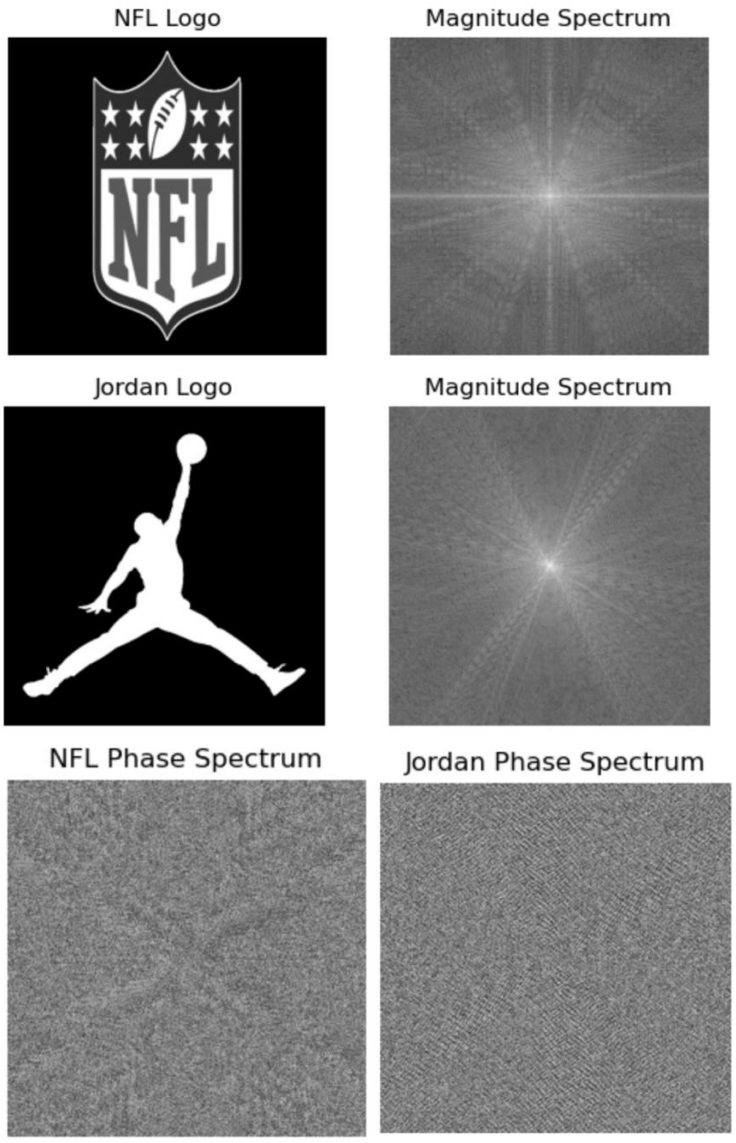
- **Task 1**
  1. *Select two images, compute their Fourier transform, and display their magnitude and phase images:*

     Code:

```
# Load images
NFL_unsized = cv2.imread('/Users/fneba/NFL.jpeg', 0)
Jordan_unsized = cv2.imread('/Users/fneba/Jordan.jpeg', 0)

# resized images for later
NFL = cv2.resize(NFL_unsized, dsize=(500, 500),
interpolation=cv2.INTER_CUBIC)
Jordan = cv2.resize(Jordan_unsized, dsize=(500, 500),
interpolation=cv2.INTER_CUBIC)

#Compute fourier transforms
NFL_Fourier = np.fft.fft2(NFL)
Jordan_Fourier = np.fft.fft2(Jordan)

# Shift the zero-frequency component to the center of the spectrum
NFL_fshift = np.fft.fftshift(NFL_Fourier)
Jordan_fshift = np.fft.fftshift(Jordan_Fourier)

# Compute phase images
NFL_phase = np.angle(NFL_fshift)
Jordan_phase = np.angle(Jordan_fshift)

#Compute magnitude images
NFL_magnitude_spectrum = 20*np.log(np.abs(NFL_fshift))
Jordan_magnitude_spectrum = 20*np.log(np.abs(Jordan_fshift))

# Display Magnitude images
plt.subplot(121),plt.imshow(NFL, cmap = 'gray')
plt.title('Input Image'), plt.axis('off')
plt.subplot(122),plt.imshow(NFL_magnitude_spectrum, cmap = 'gray')
plt.title('Magnitude Spectrum'), plt.axis('off')
plt.show()

plt.subplot(121),plt.imshow(Jordan, cmap = 'gray')
plt.title('Input Image'), plt.axis('off')
plt.subplot(122),plt.imshow(Jordan_magnitude_spectrum, cmap = 'gray')
plt.title('Magnitude Spectrum'), plt.axis('off')
plt.show()

# Display Phase images
plt.subplot(122), plt.imshow(NFL_phase, cmap='gray')
plt.title('NFL Phase Spectrum'), plt.axis('off')
plt.show()
```

```
plt.subplot(122), plt.imshow(Jordan_phase, cmap='gray')
plt.title('Jordan Phase Spectrum'), plt.axis('off')
plt.show()
```

Output:



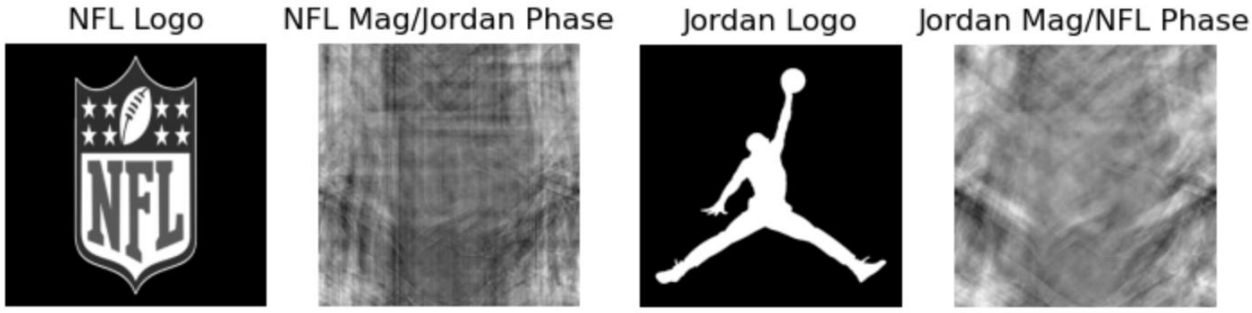
2. *Swap their phase images, perform the inverse Fourier transform, and reconstruct the images (display these):*

Code:

```
## unshifted phase images for later
NFL_phase_unshifted = np.angle(NFL_Fourier)
Jordan_phase_unshifted = np.angle(Jordan_Fourier)

### Swap phase, inverse FT and reconstruct (Source used:
https://numpy.org/doc/stable/reference/routines.fft.html)

# NFL Logo w/ Jordan phase
```

```
NFL_mag_Jordan_phase = np.multiply(NFL_Fourier,
np.exp(1j*Jordan_phase_unshifted))
NFLordan = np.real(np.fft.ifft2(NFL_mag_Jordan_phase))

plt.subplot(131),plt.imshow(NFL, cmap = 'gray')
plt.title('NFL Logo'), plt.axis('off')
plt.subplot(132),plt.imshow(NFLordan, cmap = 'gray')
plt.title('NFL Mag/Jordan Phase'), plt.axis('off')
plt.show()

# Jordan Logo w/ NFL phase
Jordan_mag_NFL_phase = np.multiply(Jordan_Fourier,
np.exp(1j*NFL_phase_unshifted))
JorNFL = np.real(np.fft.ifft2(Jordan_mag_NFL_phase))

plt.subplot(131),plt.imshow(Jordan, cmap = 'gray')
plt.title('Jordan Logo'), plt.axis('off')
plt.subplot(132),plt.imshow(JorNFL, cmap = 'gray')
plt.title('Jordan Mag/NFL Phase'), plt.axis('off')
plt.show()
```

Output:



3. Comments on the images after phases being swapped:

   Simply put the image look nothing like how they once did. There aren't even any patterns that resemble the original pictures.

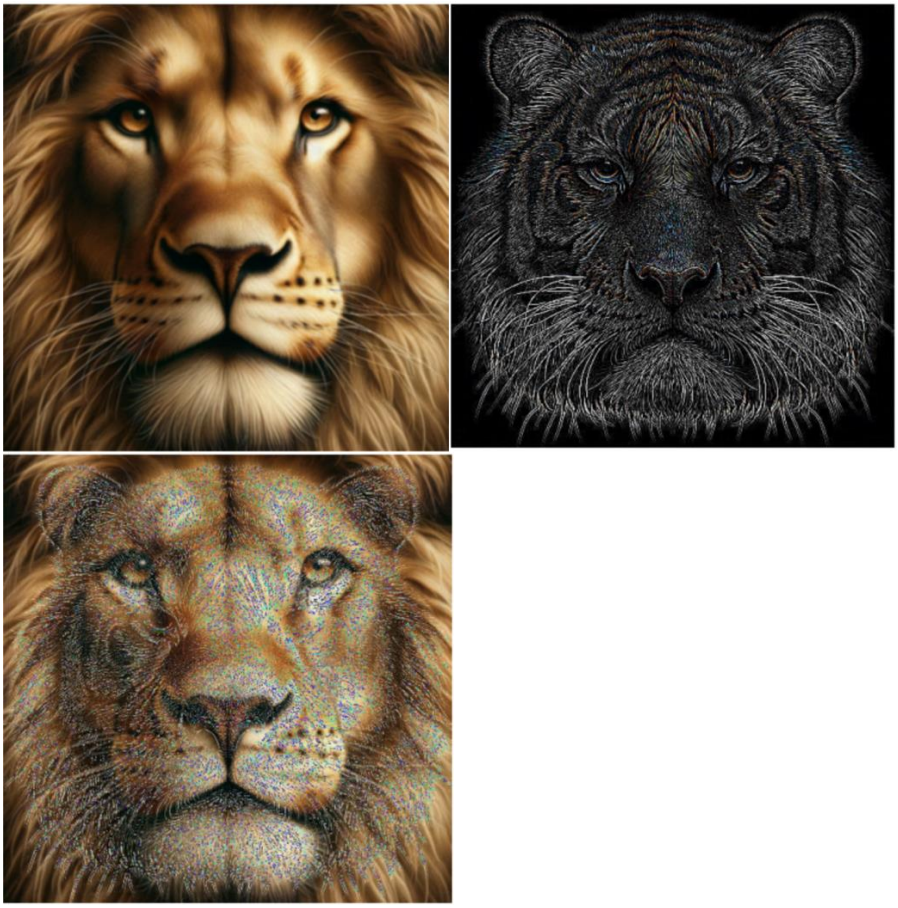
- **Task 2**

Code:

```
# load images
Lion = cv2.imread('/Users/fneba/Lion.jpg', cv2.IMREAD_COLOR)
Tiger = cv2.imread('/Users/fneba/Tiger.jpg', cv2.IMREAD_COLOR)

# Convert to RGB
Tiger_RGB = cv2.cvtColor(Tiger, cv2.COLOR_BGR2RGB)
Lion_RGB = cv2.cvtColor(Lion, cv2.COLOR_BGR2RGB)

# Use blur and edgedetect functions from earlier
```

```
Tiger_edges = edgeDetect(Tiger_RGB)
Lion_blur = blur(Lion_RGB)

# combine and display result
Result = Lion_blur + Tiger_edges

plt.imshow(Result)
plt.axis('off')
plt.show()

plt.imshow(Tiger_edges)
plt.axis('off')
plt.show()

plt.imshow(Lion_blur)
plt.axis('off')
plt.show()
```

Output:



First image is the blurred lion, second is the edges of the tiger image, then the third is the combined. My pictures weren't chosen well, so the result isn't as good. But I think the implementation is correct.

IV Multiresolution Image Blending

- **Task 1**

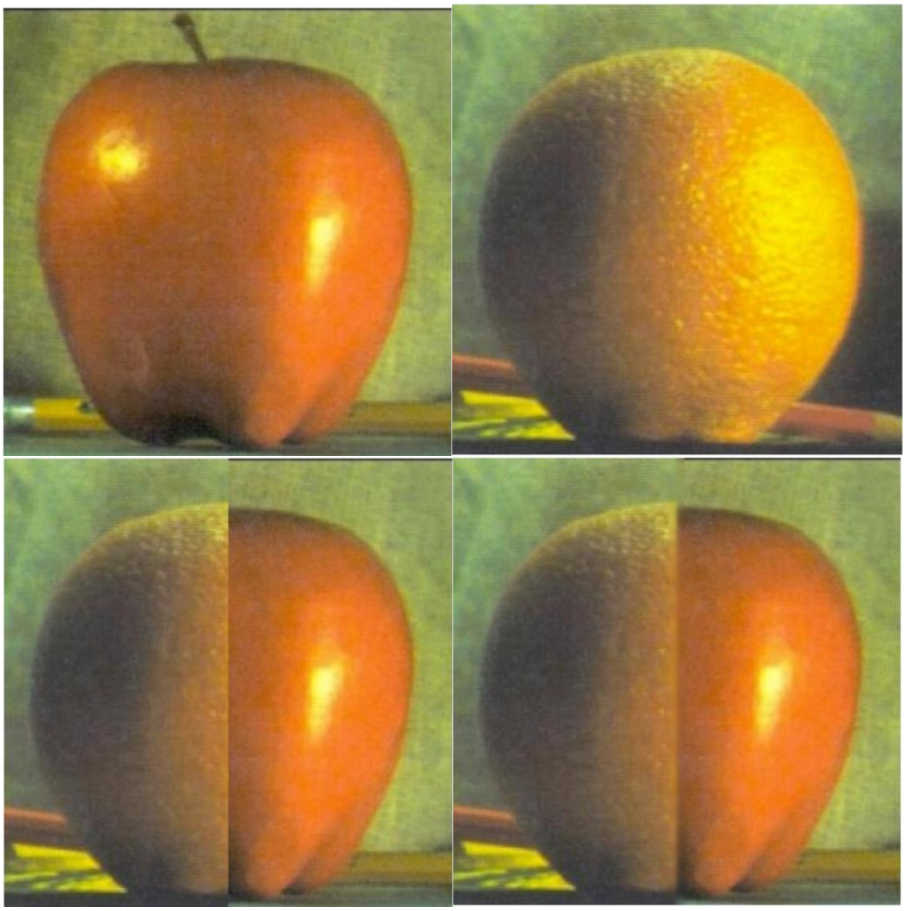
  Code:

```
# Load images
Apple = cv2.imread('/Users/fneba/apple.jpeg', cv2.IMREAD_COLOR)
Orange = cv2.imread('/Users/fneba/orange.jpeg', cv2.IMREAD_COLOR)

# Check shape of apple and orange images [ both are (300, 300, 3)]
np.shape(Apple)
np.shape(Orange)

# create mask1 & mask2 same size as image and fill it with 1s
mask1 = np.ones(Apple.shape[:2], dtype="uint8")
mask2 = np.ones(Orange.shape[:2], dtype="uint8")

# Make left side of mask1 all zeros
mask1[:, :150] = 0

# Make right side of mask2 all zeros
mask2[:, 150:] = 0

# masked images
masked_apple = cv2.bitwise_and(Apple, Apple, mask=mask1)
masked_orange = cv2.bitwise_and(Orange, Orange, mask=mask2)

# Direct Blend
Direct_Orapple = masked_apple + masked_orange

# convert to BGR
Apple_rgb = cv2.cvtColor(Apple, cv2.COLOR_BGR2RGB)
masked_apple_rgb = cv2.cvtColor(masked_apple, cv2.COLOR_BGR2RGB)
Orange_rgb = cv2.cvtColor(Orange, cv2.COLOR_BGR2RGB)
masked_orange_rgb = cv2.cvtColor(masked_orange, cv2.COLOR_BGR2RGB)
Direct_Orapple_rgb = cv2.cvtColor(Direct_Orapple, cv2.COLOR_BGR2RGB)

# Display Apple, Orange, and Direct Orapple
plt.imshow(Apple_rgb)
plt.axis('off')
plt.show()

plt.imshow(Orange_rgb)
plt.axis('off')
plt.show()

plt.imshow(Direct_Orapple_rgb)
plt.axis('off')
plt.show()

# Alpha Blending (Source: https://note.nkmk.me/en/python-opencv-numpy-alpha-
blend-mask/)
```

```
area_to_be_blended = Direct_Orapple[0:300, 145:155]
alpha_blend = cv2.GaussianBlur(area_to_be_blended, (9,9), 0)
Direct_Orapple[0:300, 145:155] = alpha_blend
Alpha_Orapple_rgb = cv2.cvtColor(Direct_Orapple, cv2.COLOR_BGR2RGB)

# Display Alpha Orapple
plt.imshow(Alpha_Orapple_rgb)
plt.axis('off')
plt.show()
```

Output:

Apple, Orange, Direct Blended Orapple, Alpha Blended Orapple



- **Task 2**

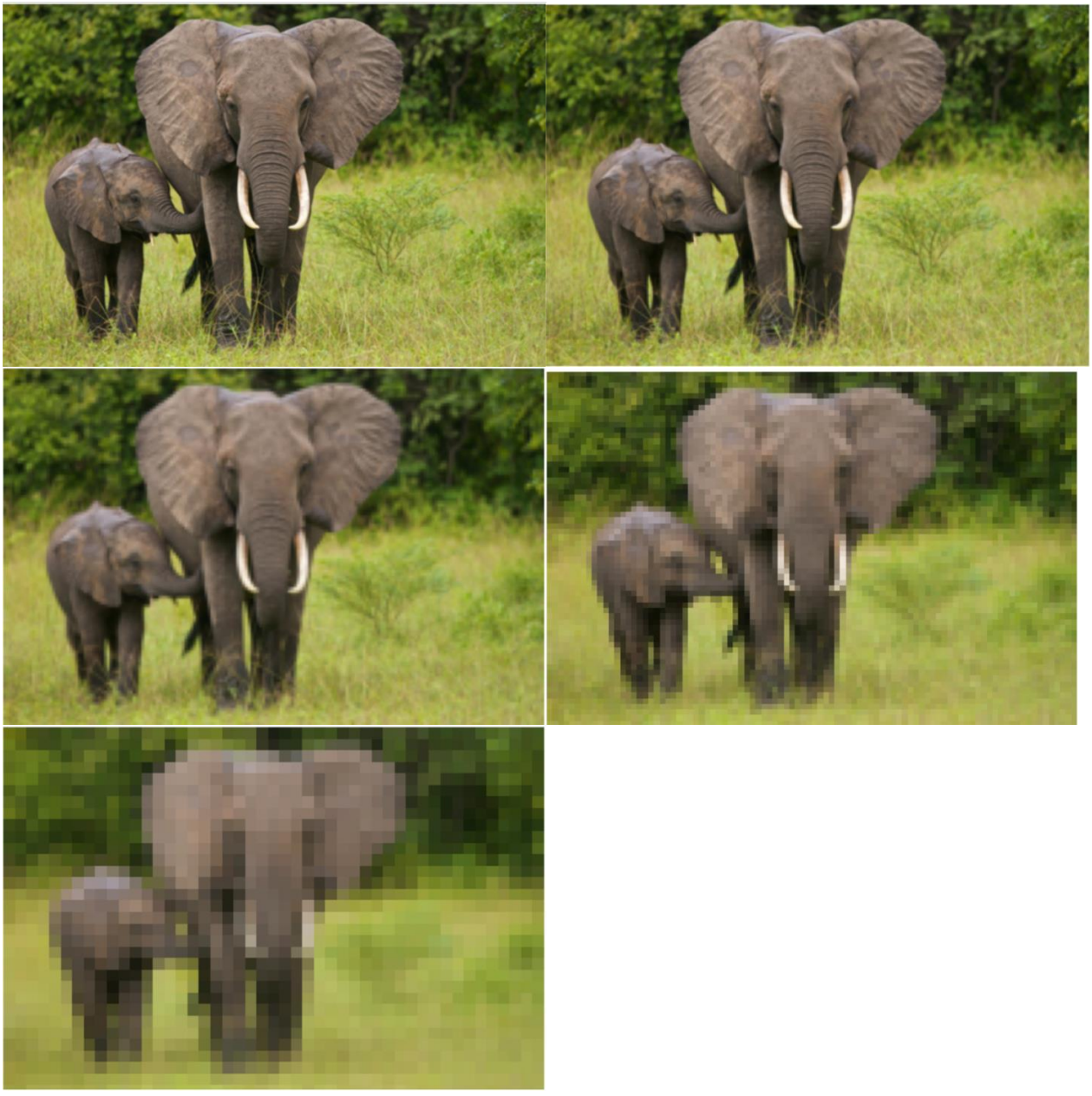
Code:

```
# Gaussian and Laplacian Pyramids

# Custom functions for each functionality of the pyramid(s)

def downsample(image):
    # skip rows and columns, removing them to downsample
    return image[::2, ::2]

def upsample(image, imageSize):
```

```
    # the factor by with we will resize (the factor is determined by the
sizing of the image you put as second input)
    sizing_factor = (imageSize.shape[1],imageSize.shape[0])

    # upsize and smooth
    upsized_image = cv2.resize(image, sizing_factor,
interpolation=cv2.INTER_NEAREST)
    upsampled = cv2.GaussianBlur(upsized_image, (5,5), 0)

    return upsampled

# Gaussian Pyramid on slides only includes the encoding phase not a decoding
phase
def GaussianPyramid(image):
    pyramid = []
    GP_image = image
    for i in range(5):
        GP_image = downsample(blur(GP_image))
        pyramid.append(GP_image)
    return pyramid

def LaplacianPyramid(image):
    pyramid = []
    LP_image = image
    for i in range(5):
        downsampled_image = downsample(LP_image)
        # upsampled downsample to compute laplacian
        upsampled_di = upsample(downsampled_image, LP_image)
        laplacian_image = cv2.subtract(LP_image, upsampled_di)
        pyramid.append(laplacian_image)
        LP_image = downsampled_image
    pyramid.append(LP_image)
    return pyramid

elephant_image = cv2.imread('/Users/fneba/elephant.jpeg', cv2.IMREAD_COLOR)

# Convert loaded image to RGB
elephant = cv2.cvtColor(elephant_image, cv2.COLOR_BGR2RGB)

# compute pyramids and print results at each step
Gauss = GaussianPyramid(elephant)
for image in Gauss:
    plt.imshow(image)
    plt.axis('off')
    plt.show()

Laplace = LaplacianPyramid(elephant)
for image in Laplace:
    plt.imshow(image)
    plt.axis('off')
    plt.show()
```

Output:

Gaussian Pyramid (5 steps)



Laplacian Pyramid (5 steps)

- I was unfortunately unable to finish tasks 3 & 4 of this problem, due to immense fatigue. I tried to do as much of this assignment as possible, though.

V Required for 691 Section

- **Prompt 1:**

  Assuming no or little prior knowledge of either fruit, or the fact that an orapple doesn't exist, with Direct and Alpha blending, they blend images, however, there are small errors in the result produced that allow us to know that the resulting image is blended. With multiresolution blending, the result has much less errors that the two other methods, so multiresolution blending can be used to created deepfakes.

- **Prompt 2:**

Est. 2009

Aside from the fact that every known orange does not have a face on it, you know this is fake because with the annoying orange, the orange would be completely stationary and the only pieces that would move were the eyes (to blink) and the mouth (to speak). However, we know that facial expressions come with speech, blinking, etc. So, we know that this is a fake.

- **Prompt 3**

1. Paper 1:

   Perez, P., Gangnet, M., & Blake, A. (n.d.). *Poisson Image Editing*. https://doi.org/chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cs.jhu.edu/~misha/Fall07/Papers/Perez03.pdf

   Summary: This paper is focused on image editing using and solving a variety of Poisson equations. The paper introduces tools based on this idea, that will allow for fine tuning of the features in images down to texture, color, brightness, etc.

   Paper 2:

   Talebi H, Milanfar P. Nonlocal image editing. IEEE Trans Image Process. 2014 Oct;23(10):4460-73. doi: 10.1109/TIP.2014.2348870. Epub 2014 Aug 18. PMID: 25148666.

   Summary: This paper basically details a form of image editing using "the spectrum of a global filter computed from image affinities. The idea is that the filter uses eigenvectors and eigenfunctions that extract better details from the images, which should lead to better editing using this filter.

2. Paper 1:

   Omer Tov, Yuval Alaluf, Yotam Nitzan, Or Patashnik, and Daniel Cohen-Or. 2021. Designing an encoder for StyleGAN image manipulation. ACM Trans. Graph. 40, 4, Article 133 (August 2021), 14 pages. https://doi.org/10.1145/3450626.3459838

   Summary: This paper examines the current image editing methods which invert images to their latent space, which is just an internal (meaning using the GAN, Generative adversarial networks, of

choice) representations of its features, and then proposes new encoding principles to follow to achieve better image manipulation. They tested their proposed principles and encoding methods on various images such as “cars and horses.”

Paper 2:

Bau, D., Strobelt, H., Peebles, W., Wulff, J., Zhou, B., Zhu, J.Y., & Torralba, A. (2019). Semantic photo manipulation with a generative image prior. ACM Transactions on Graphics, 38(4), 1–11.

Summary: This paper talks about image manipulation and reproduction using GANs. It touches on the difficulties for current GANs to reproduce images from an actual image instead of other input methods, say text. The authors then address this by proposing a new method of reconstruction and test it using a variety of tasks including “synthesizing new objects consistent with background, removing unwanted objects, and changing the appearance of an object.”

3. Paper 1:

Raza, A.; Munir, K.; Almutairi, M. A Novel Deep Learning Approach for Deepfake Image Detection. *Appl. Sci.* **2022**, *12*, 9820. https://doi.org/10.3390/app12199820

Summary: This paper was motivated by the illegal and dangerous uses behind deepfakes, e.g., fake obscene videos of people, cyber extortion, and other deep fake crimes. The goal of the paper is to be able to detect deepfakes efficiently using a “novel deepfake predictor (DFP) approach based on a hybrid of VGG16 and convolutional neural network architecture.” The researchers build their framework and then train/test it on a large dataset of deepfakes to see its efficiency. After training and testing it was found that this method is much more accurate at detecting deepfakes than other more conventional methods.

Paper 2:

Jang H, Hou JU. Exposing Digital Image Forgeries by Detecting Contextual Abnormality Using Convolutional Neural Networks. Sensors (Basel). 2020 Apr 16;20(8):2262. doi: 10.3390/s20082262. PMID: 32316220; PMCID: PMC7219587.

Summary: This paper is about the use of convolutional neural networks to find contextual abnormalities in images to detect if they are edited, forged or otherwise fraudulent. The motivation behind this paper is that previous methods of image forgery detection relied mostly on abnormalities in edge/feature detection, but with CNNs they can extract meaning from the image and then looking at spatial and relational context find abnormalities.

Sources:

https://numpy.org/doc/stable/reference/routines.fft.html