Importing libraries

I will be using opencv for basics such as creating a mask or smoothing it. All the major parts will not use opencv.

In [ ]:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
#from utils import *
import os

%matplotlib inline

from time import process_time
import scipy.sparse as sparse
import scipy.sparse.linalg as linalg
from scipy.sparse import lil_matrix
from scipy.sparse.linalg import spsolve
from skimage import filters
from scipy.ndimage import convolve
```

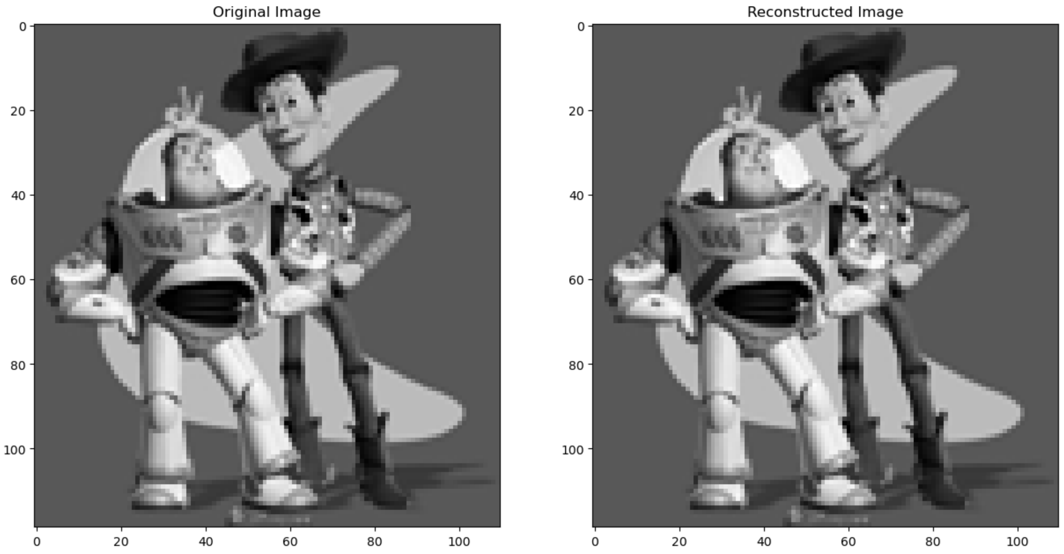
Toy Problem

In [ ]:

```
def toy_reconstruct(toy_img):
    """
    Objectives in step by step from docs:
        1. minimize (v(x+1,y)-v(x,y) - (s(x+1,y)-s(x,y)))^2
        2. minimize (v(x,y+1)-v(x,y) - (s(x,y+1)-s(x,y)))^2
        3. minimize (v(0,0)-s(0,0))^2
    """

    h, w = toy_img.shape
    im_out = np.zeros(toy_img.shape, dtype=np.double)
    im2ind = np.reshape(np.arange(0,h*w),[h,w]) # maps pixel positions to unique indexes

    e = 0 # num of equations
    A = sparse.lil_matrix((2*(h*w) + 1, h*w)) # sparse matrix
    b = np.zeros((2*(h*w) + 1, 1))

    # Objective 1
    for x in range(w - 1): # Adjusted loop range to exclude the last column
        for y in range(h):
            e = e + 1
            A[e, im2ind[y][x+1]] = 1
            A[e, im2ind[y][x]] = -1
            b[e] = toy_img[y][x+1] - toy_img[y][x]

    # Objective 2
    for y in range(h - 1): # Adjusted loop range to exclude the last row
        for x in range(w):
            e = e + 1
            A[e, im2ind[y + 1, x]] = 1
            A[e, im2ind[y, x]] = -1
            b[e] = toy_img[y + 1, x] - toy_img[y, x]


    # Objective 3
    e = e + 1
    A[e, im2ind[0][0]] = 1
    b[e] = toy_img[0][0]

    v = linalg.lsqr(A, b)[0] # Av = b, we solve for v = lsqr(A, b). Index 0 is the solution vector, as this function returns a tuple with more information.
    v = np.reshape(v, (h, w))

    for x in range(h):
        for y in range(w):
            im_out[x, y] = v[x, y] # Reconstruct image.

    return im_out
```

In [ ]:

```
#%matplotlib inline
toy_img = cv2.cvtColor(cv2.imread('./toy_problem.png'), cv2.COLOR_BGR2RGB)
toy_img = cv2.cvtColor(toy_img, cv2.COLOR_BGR2GRAY).astype('double') / 255.0

im_out = toy_reconstruct(toy_img)

fig, axes = plt.subplots(1,2, figsize = (15,15))
axes[0].set_title('Original Image')
axes[0].imshow(toy_img, cmap="gray")
axes[1].set_title('Reconstructed Image')
axes[1].imshow(im_out, cmap="gray")
plt.show()
```



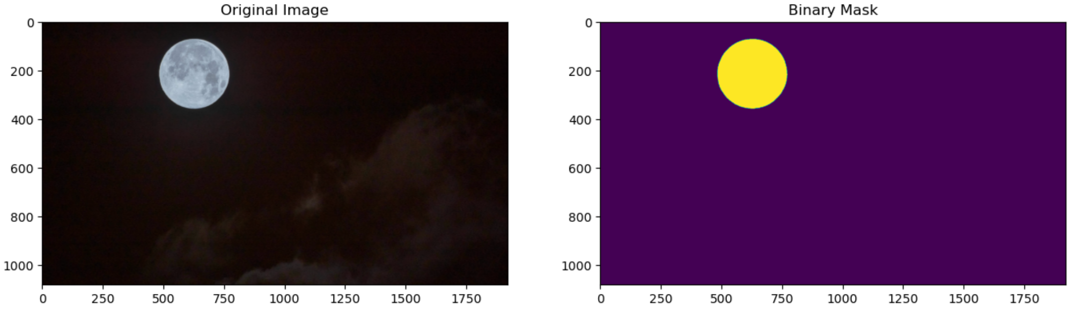
Generate mask for gradient domain boosting

In [ ]:

```
def create_binary_mask(image, tval):
    '''
    Function used to create a binary mask, in case there is no provided mask.
    '''
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    _, binary_mask = cv2.threshold(gray_image, tval, 255, cv2.THRESH_BINARY)
    return binary_mask
```

In [ ]:

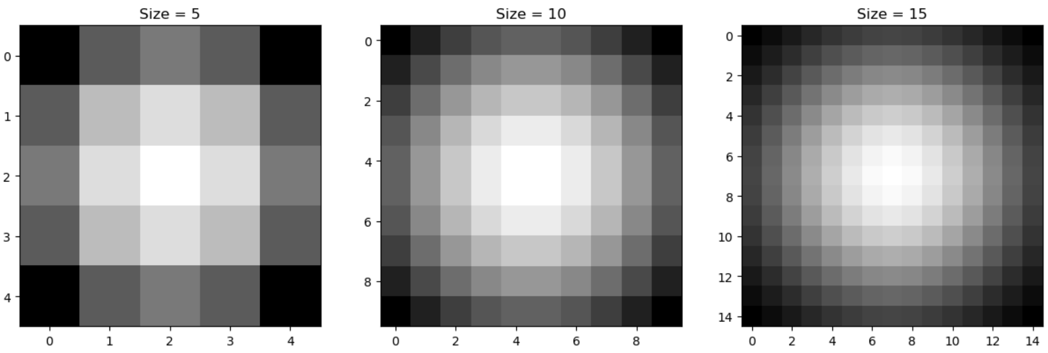
```
image = cv2.imread('./moon.jpg')
binary_mask = create_binary_mask(image, 127) / 255 # manually adjust tval

fig, axes = plt.subplots(1,2, figsize = (15,15))
axes[0].set_title('Original Image')
axes[0].imshow(image)
axes[1].set_title('Binary Mask')
axes[1].imshow(binary_mask)
plt.show()
```



For alpha blending, let's create some help functions to copy pixels of an image to another based on where mask value is 1, and to smooth mask.

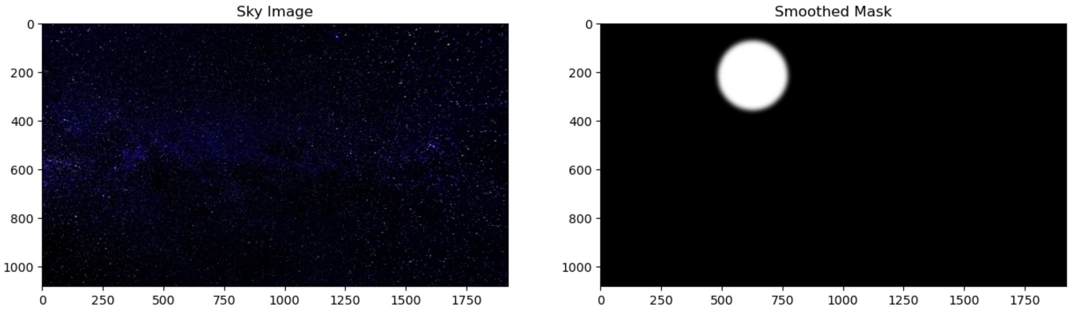
In [ ]:

```
from scipy.stats import norm

# From written homework 1, convolution function.
def convolution(image, kernel):
    # changed convolution function to work with 1 or 3 channels.
    if len(image.shape) == 3:
        rows, cols, channels = image.shape
    else:
        rows, cols = image.shape
        channels = 1

    kernel_rows, kernel_cols = kernel.shape

    # Padding for height and width
    pad_height = (kernel_rows // 2)
    pad_width = (kernel_cols // 2)

    # initialize the padded image and result image
    if channels > 1:
        pad_image = np.pad(image, ((pad_height, pad_height), (pad_width, pad_width), (0, 0)), mode='constant', constant_values=0)
        result = np.zeros_like(image)
    else:
        pad_image = np.pad(image, ((pad_height, pad_height), (pad_width, pad_width)), mode='constant', constant_values=0)
        result = np.zeros(image.shape)

    # perform convolution
    for c in range(channels) if channels > 1 else range(1):
        for i in range(rows):
            for j in range(cols):
                if channels > 1:
                    region = pad_image[i:i + kernel_rows, j:j + kernel_cols, c]
                    conv = np.sum(kernel * region)
                    result[i, j, c] = conv
                else:
                    region = pad_image[i:i + kernel_rows, j:j + kernel_cols]
                    conv = np.sum(kernel * region)
                    result[i, j] = conv

    return result

def gaussian_kernel(size, sigma):
    '''
    Returns a gaussian kernel of size=size and std=sigma.
    '''
    n = (size // 2)
    kernel = np.linspace(-n, n, size)

    for i in range(size):
        kernel[i] = norm.pdf(kernel[i], 0, sigma) # density function of normal/gaussian distribution.

    kernel_transpose = kernel.T
    kernel = np.outer(kernel_transpose, kernel_transpose) # computes outer product of the transpose kernels.
    kernel = kernel * (1.0 / kernel.max()) # normalizes the kernel, so max value becomes 1.
    return kernel

def apply_gaussian_filter(image, kernel_size, sigma):
    '''
    Smooths/blurs the mask. Uses GaussianBlur from opencv.

    Input: image, kernel size and sigma for gaussian.
    Output: blured mask.
    '''
    kernel = gaussian_kernel(kernel_size, sigma) # creates an nxn gaussian kernel, with sigma value
    result = convolution(image, kernel) # applies convolution to smooth mask.
    result = result / result.max() # normalizes the result
    return result

# Print gaussian kernels.
kernel = [gaussian_kernel(x, 5) for x in range(5, 16, 5)]
fig, axs = plt.subplots(1, 3, figsize=(15, 5))  # 1 row, 3 columns
for i in range(3):
    axs[i].imshow(kernel[i], interpolation='none', cmap='gray')
    axs[i].set_title(f"Size = {(i+1)*5}")
plt.show()
```



In [ ]:

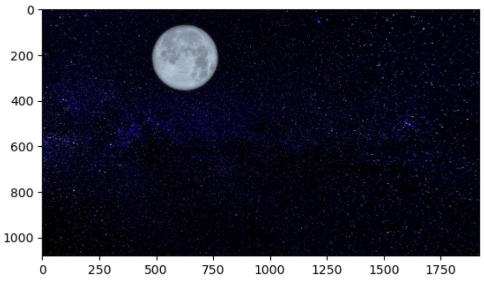
```python
def alpha_blending(image, image2, binary_mask, filter=False):
    '''
    Defining the function for alpha blending as:
    b = (alpha)*image1 + (1 - alpha)*image2
    Where b is the blended image

    Input: image, image2, binary_mask
    Output: blended image
    '''
    # check mask shape
    if binary_mask.ndim == 2:
        binary_mask = np.repeat(binary_mask[:, :, np.newaxis], 3, axis=2)

    # convert images to float32. it was necessary for blending
    image = image.astype(np.float32)
    image2 = image2.astype(np.float32)
    binary_mask = binary_mask.astype(np.float32)

    # for the case of laplacian, this is where I will use gaussian for the mask
    if filter == True:
        # I comment this later, but my own gaussian for some reason was not working properly in the other functions.
        # binary_mask = cv2.GaussianBlur(temp_image, (15,15), 5)
        binary_mask = filters.gaussian(binary_mask.astype(bool), 20)

    # normalize binary_mask to have values between 0 and 1 if necessary
    if binary_mask.max() > 1:
        binary_mask /= 255.0

    # alpha blending
    b = binary_mask * image + (1 - binary_mask) * image2

    # clip values so they are between 0 and 255
    b = np.clip(b, 0, 255).astype(np.uint8)
    return b

image2 = cv2.cvtColor(cv2.imread('./sky.jpg'), cv2.COLOR_BGR2RGB)
image2 = cv2.resize(image2, (image.shape[1], image.shape[0]), interpolation=cv2.INTER_NEAREST)
smoothed_mask = apply_gaussian_filter(binary_mask, 50, 10)

fig, axes = plt.subplots(1,2, figsize = (15,15))
axes[0].set_title('Sky Image')
axes[0].imshow(image2, cmap="gray")
axes[1].set_title('Smoothed Mask')
axes[1].imshow(smoothed_mask, cmap="gray")
plt.show()
```



Alpha Blending

In [ ]:

```python
b = alpha_blending(image, image2, smoothed_mask)
plt.imshow(b)
plt.show()
```



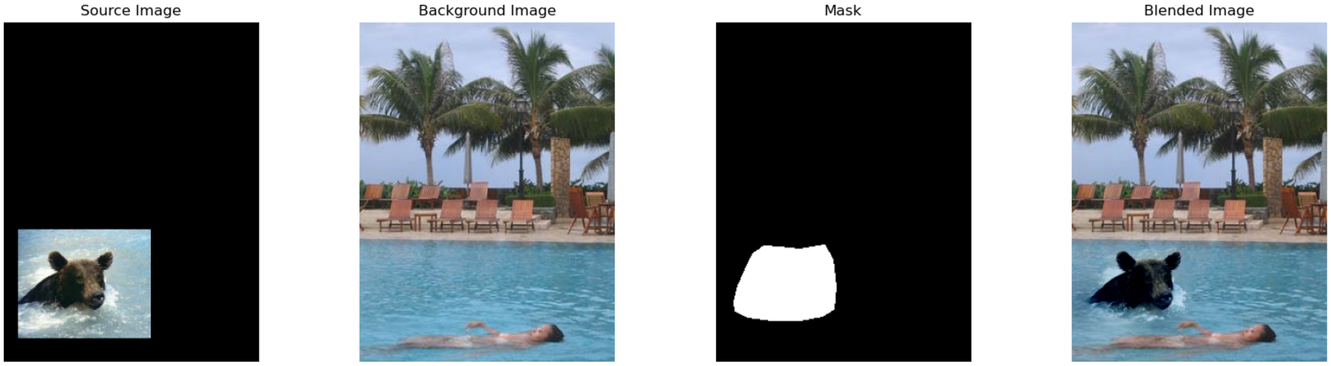
Gradient Domain Editing

In [ ]:

```python
def gradient_domain_editing(image, background, binary_mask):
    # Ensure all inputs are float type in the range [0, 1]
    image = image.astype(np.float32) / 255.0
    background = background.astype(np.float32) / 255.0
    binary_mask = binary_mask.astype(np.float32) / 255.0
    if image.shape[:2] != background.shape[:2] or background.shape[:2] != binary_mask.shape:
        print('Please provide images of equal size.')
        return None

    blended_image = np.zeros(background.shape, dtype=np.float32)

    for i in range(3): # For every channel
        # seems like the toy problem with a couple changes. checks binary_mask.
        image_ = image[:, :, i]
        background_ = background[:, :, i]

        h, w = image_.shape
        im2ind = np.reshape(np.arange(h*w), (h, w))

        e = 0
        A = sparse.lil_matrix((2*h*w + 1, h*w), dtype=np.float32)
        b = np.zeros(2* h * w + 1, dtype=np.float32)

        for y in range(0, h):
            for x in range(0, w):
                if x + 1 < w:
                    if binary_mask[y,x] == 1: # uses image
                        b[e] = image_[y,x] - image_[y,x + 1]
                    else: # uses background
                        b[e] = background_[y,x] - background_[y,x+1]
                    A[e, im2ind[y,x]] = 1
                    A[e, im2ind[y, x + 1]] = -1
                    e = e + 1
                if y + 1 < h:
                    if binary_mask[y,x] == 1:
                        b[e] = image_[y,x] - image_[y + 1,x]
                    else:
                        b[e] = background_[y,x] - background_[y + 1,x]
                    A[e, im2ind[y,x]] = 1
                    A[e, im2ind[y + 1, x ]] = -1
                    e = e + 1

        A[e,im2ind[0,0]] = 1
        b[e] = background_[0,0]

        A = A.tocsr()
        v = sparse.linalg.lsqr(A, b)[0]
        v = np.reshape(v, (h, w))

        # changed the for loop for this line of code.
        blended_image[:, :, i] = v

    # Convert the blended image back to 8-bit format
    blended_image = np.clip(blended_image * 255, 0, 255).astype('uint8')
    return blended_image
```

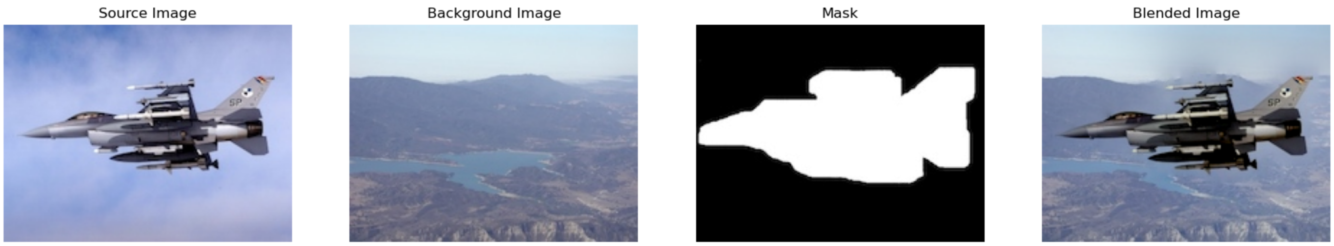
In [ ]:

```
# Gradient Domain Editing, testing

# Load and process images
image = cv2.cvtColor(cv2.imread('./source_1.png'), cv2.COLOR_BGR2RGB)
background = cv2.cvtColor(cv2.imread('./target_01.jpg'), cv2.COLOR_BGR2RGB)
mask = cv2.imread('./target_01_mask.png', cv2.IMREAD_GRAYSCALE)

blended_image = gradient_domain_editing(image, background, mask)

fig, axs = plt.subplots(1, 4, figsize=(20, 5))

axs[0].imshow(image)
axs[0].set_title('Source Image')
axs[0].axis('off')
axs[1].imshow(background)
axs[1].set_title('Background Image')
axs[1].axis('off')
axs[2].imshow(mask, cmap='gray')
axs[2].set_title('Mask')
axs[2].axis('off')
axs[3].imshow(blended_image)
axs[3].set_title('Blended Image')
axs[3].axis('off')
plt.show()
```



In [ ]:

```
# Another Example
image = cv2.cvtColor(cv2.imread('./plane.jpg'), cv2.COLOR_BGR2RGB)
background = cv2.cvtColor(cv2.imread('./back.jpg'), cv2.COLOR_BGR2RGB)
mask = cv2.imread('./binary.jpg', cv2.IMREAD_GRAYSCALE)

blended_image = gradient_domain_editing(image, background, mask)

fig, axs = plt.subplots(1, 4, figsize=(20, 5))

axs[0].imshow(image)
axs[0].set_title('Source Image')
axs[0].axis('off')
axs[1].imshow(background)
axs[1].set_title('Background Image')
axs[1].axis('off')
axs[2].imshow(mask, cmap='gray')
axs[2].set_title('Mask')
axs[2].axis('off')
axs[3].imshow(blended_image)
axs[3].set_title('Blended Image')
axs[3].axis('off')
plt.show()
```

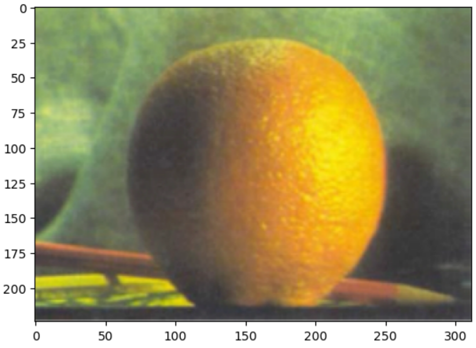


Laplacian Blend Used the specific orange/apple image from lecture to blend.

In [ ]:

```
# define up sampling and down sampling functions.
def down_sampling(image):
    '''
    Downsamples an image by a factor of 2.
    '''
    # new dimensions before downsampling
    _, _, channels = image.shape
    new_height = int(image.shape[0] / 2)
    new_width = int(image.shape[1] / 2)

    # initialize the new image with zeros
    new_image = np.zeros((new_height, new_width, 3), dtype=np.uint8)
    for i in range(3): # Iterate over each channel
        for y in range(0, image.shape[0], 2): # For every other pixel in height
            for x in range(0, image.shape[1], 2): # For every other pixel in width
                new_image[y//2, x//2, i] = image[y, x, i] # copy every other pixel

    return new_image

def up_sampling(image, mask=False):
    '''
    Correctly upsamples an image by a factor of 2.
    '''
    # new dimensions after upsampling
    new_height, new_width, channels = image.shape
    new_width *= 2
    new_height *= 2

    # initialize the new image with zeros
    new_image = np.zeros((new_height, new_width, 3), dtype=np.uint8)
    for i in range(channels):  # if image has 3 channels (RGB)
        for y in range(image.shape[0]):
            for x in range(image.shape[1]):
                new_image[2*y:2*y+2, 2*x:2*x+2, i] = image[y, x, i]

    return new_image

# TEST FUNCTIONS
example = cv2.cvtColor(cv2.imread('./orange.png'), cv2.COLOR_BGR2RGB)
print(example.shape)
example1 = up_sampling(example)
print(example1.shape)
example2 = down_sampling(example)
print(example2.shape)
plt.imshow(example2)
plt.show()
```

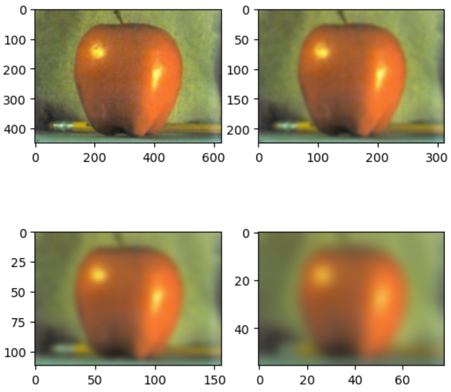
```
(448, 624, 3)
(896, 1248, 3)
(224, 312, 3)
```



In [ ]:

```
# Gaussian Pyramid based on formula provided
def gaussian_pyramid(image, layers):
    '''
    Generates the gaussian_pyramid. Takes an image as input, and number of layers.
    '''
    gaussian_pyr = []
    gaussian_pyr.append(image) # based on formula, GP0(I) = I

    temp_image = image.copy() # so it does not change the original image
    for i in range(layers):
        # I was getting weird outputs this time with my gaussian filter function, so I decided to use cv2 only for this purpose and nothing else.
        # temp_image = apply_gaussian_filter(temp_image, 25, 5) # apply gaussian filter
        temp_image = cv2.GaussianBlur(temp_image, (15,15), 5)
        temp_image = down_sampling(temp_image)
        gaussian_pyr.append(temp_image) # GP1(I) =D(GP0(I)*g

    return gaussian_pyr

# I was having weird outputs with my gaussian filter, so I decided to test it. It turns out there is something I am forgetting to do on my filter,
# so I decided to use opencv just for this problem.

# TEST GAUSSIAN PYRAMID
image = cv2.cvtColor(cv2.imread('./apple.png'), cv2.COLOR_BGR2RGB)
pyramid = gaussian_pyramid(image, 3) # try out with n = 3

fig, axes = plt.subplots(2,2, figsize=(6,6))
x = 0
for i in range(2):
    for j in range(2):
        axes[i, j].imshow(pyramid[x])
        print(pyramid[x].shape)
        x += 1
```

```
(448, 624, 3)
(224, 312, 3)
(112, 156, 3)
(56, 78, 3)
```

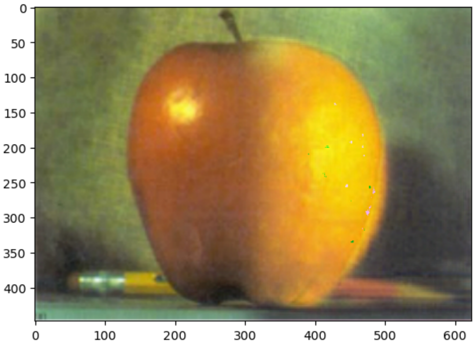


In [ ]:
```
# # Laplacian Pyramid based on formula provided.
def laplacian_pyramid(img, levels):

    laplacian_pyr = []
    gaussian_pyr = gaussian_pyramid(img, levels)

    for i in range(levels):
        if i  < levels:
            i_ = gaussian_pyr[i+1]
            up_samp = up_sampling(i_) # up samples one level above

            if gaussian_pyr[i].shape[0] != up_samp.shape[0] or gaussian_pyr[i].shape[1] != up_samp.shape[1]: # checks the shape, and make adjustments.
                up_samp = np.resize(up_samp, gaussian_pyr[i].shape)

            new_val = (gaussian_pyr[i] - up_samp).astype(np.uint8) # GPn(I)-U(GPn+1(I))
            laplacian_pyr.append(new_val)

    return laplacian_pyr
```

In [ ]:
```
# Finally, the laplacian blending function
def laplacian_blending(source, target, mask, levels):
    '''
    From the assignment page, source is S, target is T, and mask is M.
    '''
    # First, create 'mask pyramid.' From one of the sources I found over the internet, the Youtube video, the professor
    # mentions also about creating a list and downsampling the mask, and that is required. Since I was a little bit confused with it,
    # I used his approach together with the one from assignment.

    # pyramids
    if mask.ndim == 2:  # check if mask has only 2 dimensions
        mask = np.stack((mask,)*3, axis=-1) # creates 3rd dimension for every channel.

    mask_pyr = [mask]
    for _ in range(levels):
        mask = down_sampling(mask)
        mask_pyr.append(mask)

    gaussian_pyr_s = gaussian_pyramid(source, levels)
    gaussian_pyr_t = gaussian_pyramid(target, levels)
    laplacian_pyr_s = laplacian_pyramid(source, levels)
    # laplacian_pyr_t = laplacian_pyramid(target, levels)

    # Following the formula given to us from the assignment page, plus the youtube video from the source list, I came up with this.
    result = []
    for i in range(len(gaussian_pyr_s)):
        result.append(alpha_blending(gaussian_pyr_s[i], gaussian_pyr_t[i], mask_pyr[i], filter=True))

    # Use levels until reaching GP(B)_Zero, which is the solution from the formula.
    for i in range(levels, 0, -1):
        alpha_blend = alpha_blending(up_sampling(gaussian_pyr_s[i]),up_sampling(gaussian_pyr_t[i]), up_sampling(mask_pyr[i]), filter = True)
        result[i - 1] = laplacian_pyr_s[i - 1] + alpha_blend

    return result[0]
```

In [ ]:
```
# source, target, and mask from lecture slides.
source = cv2.cvtColor(cv2.imread('./apple.png'), cv2.COLOR_BGR2RGB)
target = cv2.cvtColor(cv2.imread('./orange.png'), cv2.COLOR_BGR2RGB)
mask = cv2.cvtColor(cv2.imread('./mask.png'), cv2.COLOR_BGR2RGB)

# result
lap_blending = laplacian_blending(source, target, mask, 3)
```

In [ ]:
```
plt.imshow(lap_blending)
plt.show()
```



Sources:

1. https://www.youtube.com/watch?v=N4EPJJx7xVo
2. https://www.adeveloperdiary.com/data-science/computer-vision/applying-gaussian-smoothing-to-an-image-using-python-from-scratch/
3. https://www.phatcode.net/articles.php?id=233

4. https://www.andrew.cmu.edu/course/16-726/projects/juyongk/proj2/
5. https://learning-image-synthesis.github.io/sp23/assignments/hw2
6. https://cs.brown.edu/courses/cs129/results/proj2/taox/
7. https://pavancm.github.io/pdf/AIP_Mid_Report.pdf
8. https://github.com/willemmanuel/poisson-image-editing/tree/master?tab=readme-ov-file

I also used NumPy, SciPy, OpenCV, and Python standard libraries documentation to assist me with the assignment. In addition, the last source used (github page) is from where I took the image. I was having a hard time finding good images to test my part 1 and 2, so I used the airplane one from the readme of that github page.