Import Libraries

In [1]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
%matplotlib inline
import matplotlib
matplotlib.rcParams['figure.figsize'] = (9.0, 9.0)
from IPython.display import Image
```

Display Image Directly

We will use the following as our sample images. We will use the ipython image function to load and display the image.

In [2]:
```
# Display image.
Image(filename='imgs/yourImage.png')
```

Out[2]:

Reading images using OpenCV

OpenCV allows reading different types of images (JPG, PNG, etc). You can load grayscale images, color images or you can also load images with Alpha channel. It uses the **`cv2.imread()`** function which has the following syntax:

Function Syntax

```
retval = cv2.imread( filename[, flags] )
```

`retval` : Is the image if it is successfully loaded. Otherwise it is `None` . This may happen if the filename is wrong or the file is corrupt.

The function has **1 required input argument** and one optional flag:

1. `filename` : This can be an **absolute** or **relative** path. This is a **mandatory argument**.
2. `Flags` : These flags are used to read an image in a particular format (for example, grayscale/color/with alpha channel). This is an **optional argument** with a default value of `cv2.IMREAD_COLOR` or `1` which loads the image as a color image.

Before we proceed with some examples, let's also have a look at some of the `flags` available.

Flags

1. **`cv2.IMREAD_GRAYSCALE`** or **`0`** : Loads image in grayscale mode
2. **`cv2.IMREAD_COLOR`** or **`1`** : Loads a color image. Any transparency of image will be neglected. It is the default flag.
3. **`cv2.IMREAD_UNCHANGED`** or **`-1`** : Loads image as such including alpha channel.

OpenCV Documentation

`Imread:` https://docs.opencv.org/4.5.1/d4/da8/group__imgcodecs.html#ga288b8b3da0892bd651f
(https://docs.opencv.org/4.5.1/d4/da8/group__imgcodecs.html#ga288b8b3da0892bd651fce07b3bb

`ImreadModes:`
https://docs.opencv.org/4.5.1/d8/d6a/group__imgcodecs__flags.html#ga61d9b0126a3e57d9277ac4
(https://docs.opencv.org/4.5.1/d8/d6a/group__imgcodecs__flags.html#ga61d9b0126a3e57d9277ac4

In [3]:

```
# Read image as gray scale.
cb_img = cv2.imread("imgs/yourImage.png",0)

# Print the image data (pixel values), element of a 2D numpy array.
# Each pixel value is 8-bits [0,255]
print(cb_img)
```

```
[[ 34  30  26 ... 143 109 122]
 [ 33  29  26 ... 144  90  76]
 [ 30  30  27 ... 124  80  64]
 ...
 [ 24  23  25 ... 100  97  70]
 [ 26  21  20 ...  74 103 100]
 [ 32  26  21 ...  52  83  83]]
```

Display Image attributes

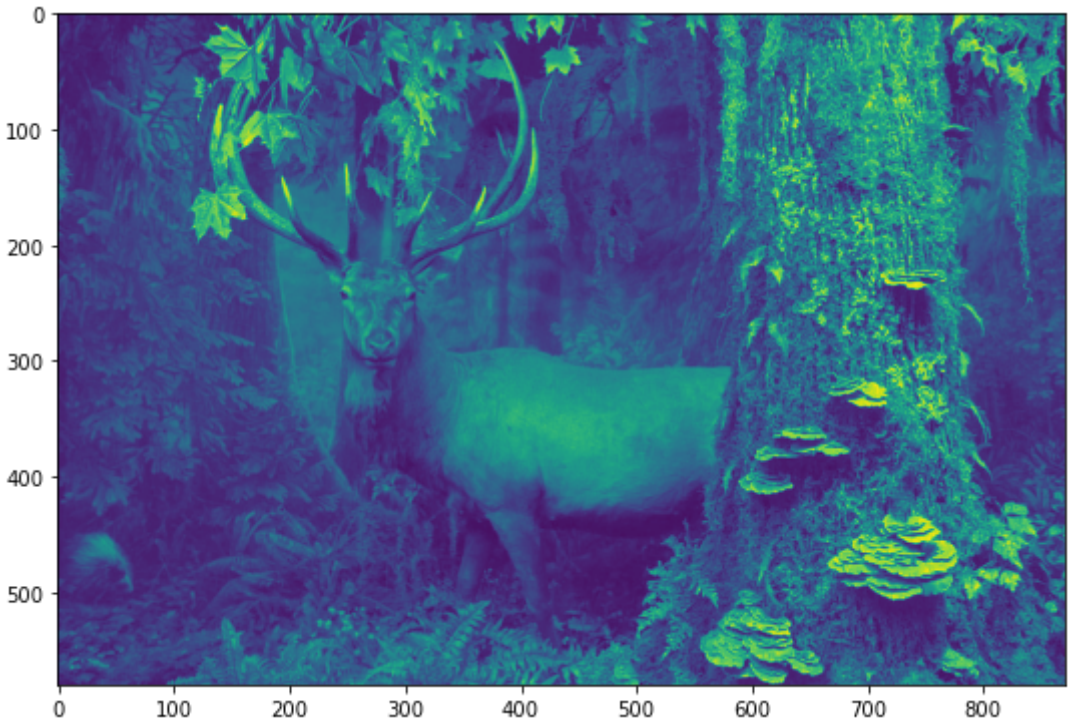
In [4]:
```
# print the size  of image
print("Image size is ", cb_img.shape)

# print data-type of image
print("Data type of image is ", cb_img.dtype)
```

```
Image size is  (580, 870)
Data type of image is  uint8
```

Display Images using Matplotlib

In [5]:
```
# Display image.
plt.imshow(cb_img)
```

Out[5]: <matplotlib.image.AxesImage at 0x7fecf108d2e0>

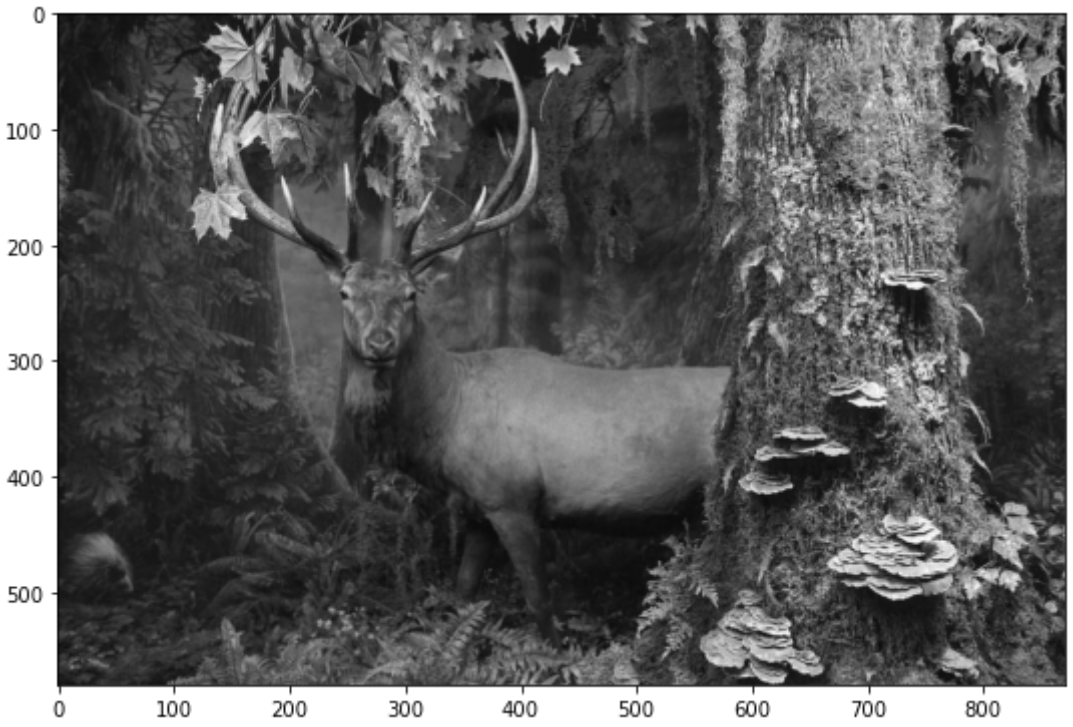


What happened?

Even though the image was read in as a gray scale image, it won't necessarily display in gray scale when using `imshow()`. matplotlib uses different color maps and it's possible that the gray scale color map is not set.

In [6]:
```
# Set color map to gray scale for proper rendering.
plt.imshow(cb_img, cmap='gray')
```

Out[6]: <matplotlib.image.AxesImage at 0x7fec90183c40>

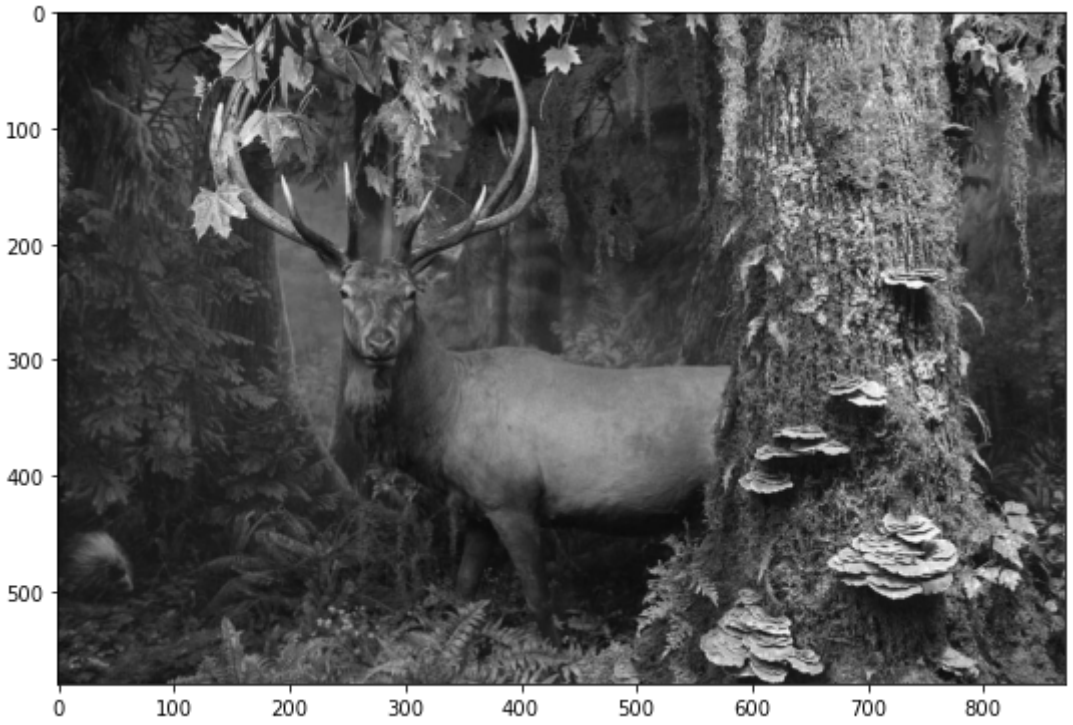


Another example

In [7]:
```
# Read image as gray scale.
cb_img_fuzzy = cv2.imread("imgs/yourImage.png",0)

# print image
print(cb_img_fuzzy)

# Display image.
plt.imshow(cb_img_fuzzy,cmap='gray')
```

```
[[ 34  30  26 ... 143 109 122]
 [ 33  29  26 ... 144  90  76]
 [ 30  30  27 ... 124  80  64]
 ...
 [ 24  23  25 ... 100  97  70]
 [ 26  21  20 ...  74 103 100]
 [ 32  26  21 ...  52  83  83]]
```

Out[7]: <matplotlib.image.AxesImage at 0x7fece0b53d30>



Working with Color Images

Until now, we have been using gray scale images in our discussion. Let us now discuss color images.

In [8]:
```
1 # Read and display Coca-Cola logo.
2 Image("imgs/coca-cola-logo.png")
```

Out[8]:

Read and display color image

Let us read a color image and check the parameters. Note the image dimension.

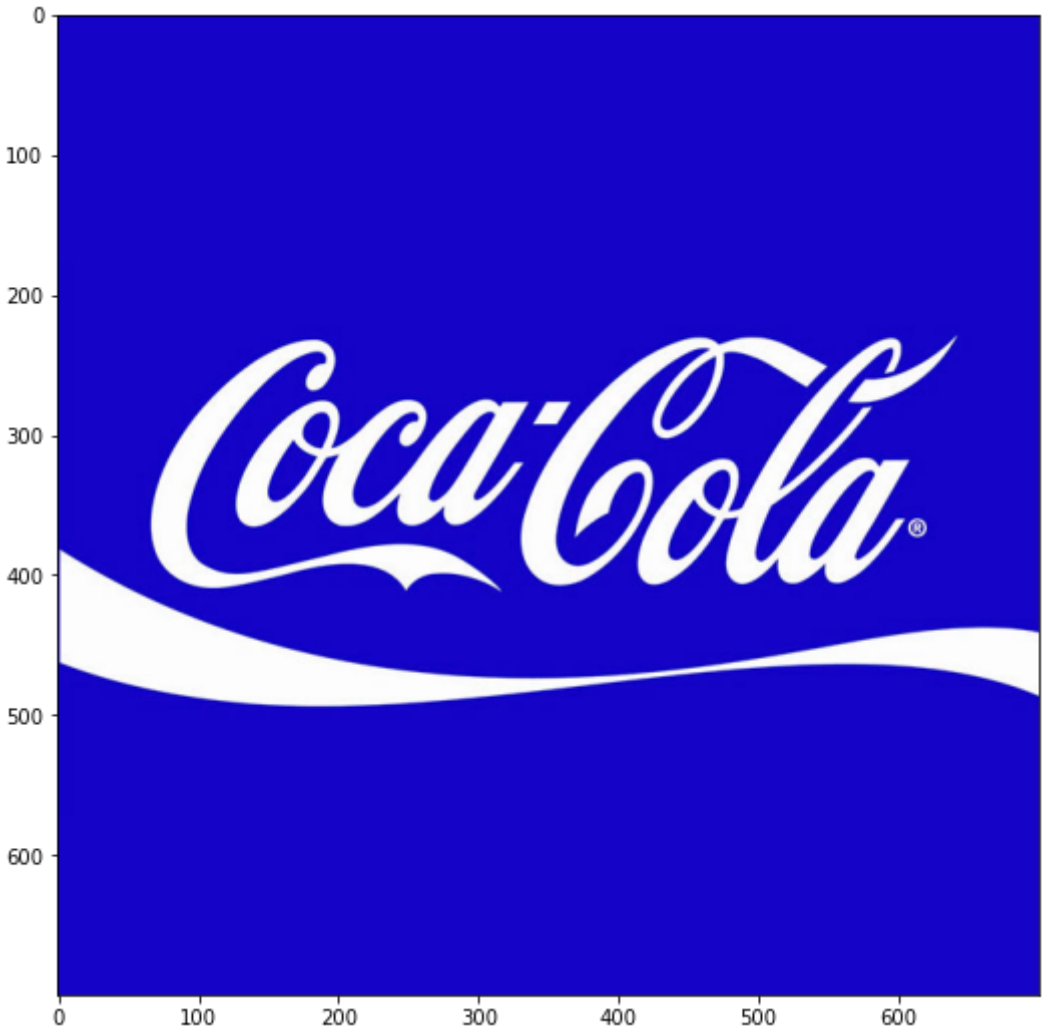
In [9]:
```
# Read in image
coke_img = cv2.imread("imgs/coca-cola-logo.png",1)

# print the size  of image
print("Image size is ", coke_img.shape)

# print data-type of image
print("Data type of image is ", coke_img.dtype)

print("")
```

```
Image size is  (700, 700, 3)
Data type of image is  uint8
```

Display the Image

In [10]:
```
plt.imshow(coke_img)
#  What happened?
```

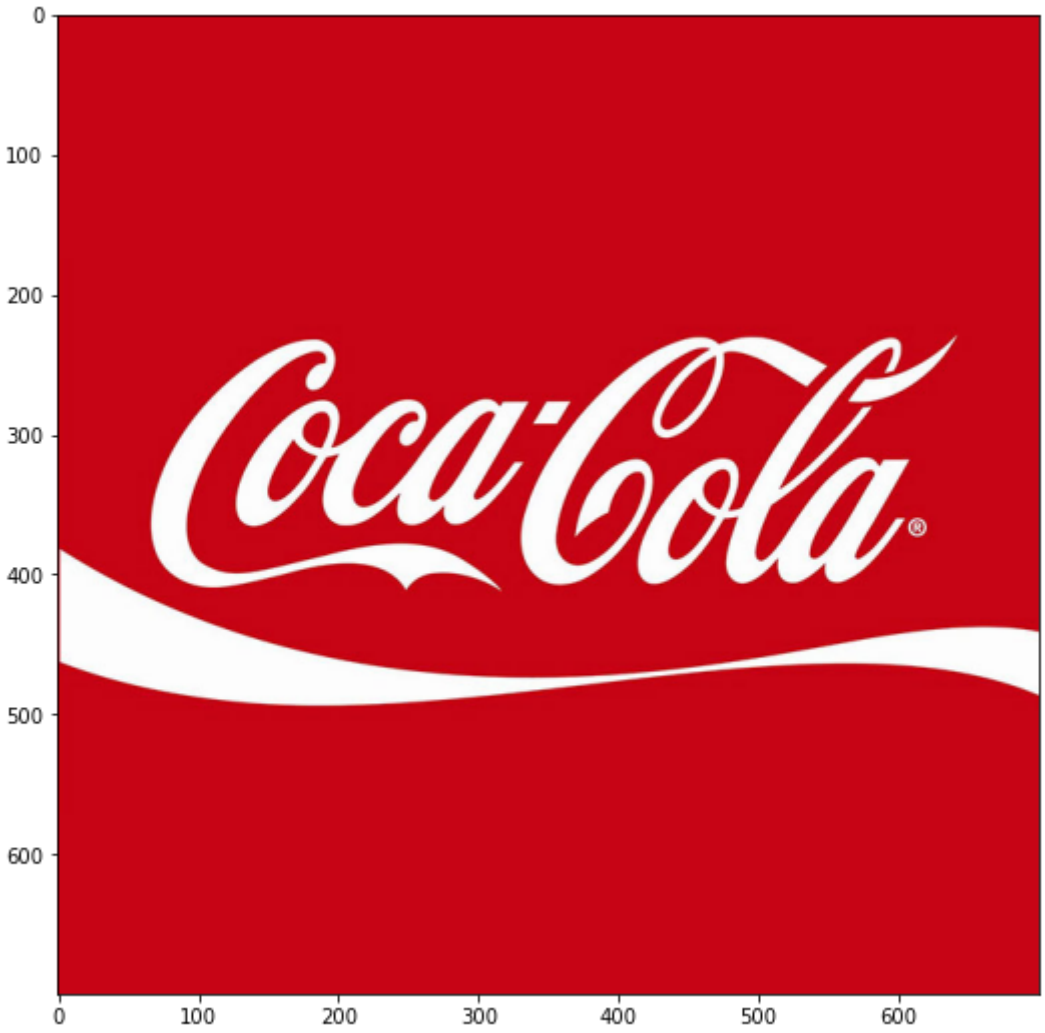
Out[10]: <matplotlib.image.AxesImage at 0x7fec901fe370>



The color displayed above is different from the actual image. This is because matplotlib expects the image in RGB format whereas OpenCV stores images in BGR format. Thus, for correct display, we need to reverse the channels of the image. We will discuss about the channels in the sections below.

In [15]:
```
coke_img_channels_reversed = coke_img[:, :, ::-1]
plt.imshow(coke_img_channels_reversed)
```

Out[15]: <matplotlib.image.AxesImage at 0x7fbe98cc6d30>



Splitting and Merging Color Channels

`cv2.split()` Divides a multi-channel array into several single-channel arrays.

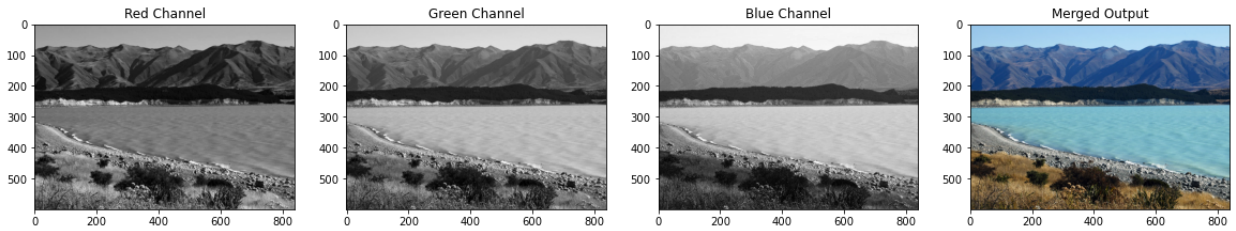
`cv2.merge()` Merges several arrays to make a single multi-channel array. All the input matrices must have the same size.

OpenCV Documentation

https://docs.opencv.org/4.5.1/d2/de8/group__core__array.html#ga0547c7fed86152d7e9d0096029c8
(https://docs.opencv.org/4.5.1/d2/de8/group__core__array.html#ga0547c7fed86152d7e9d0096029c

In [16]:

```
# Split the image into the B,G,R components
img_NZ_bgr = cv2.imread("imgs/New_Zealand_Lake.jpg",cv2.IMREAD_COLOR)
b,g,r = cv2.split(img_NZ_bgr)

# Show the channels
plt.figure(figsize=[20,5])
plt.subplot(141);plt.imshow(r,cmap='gray');plt.title("Red Channel");
plt.subplot(142);plt.imshow(g,cmap='gray');plt.title("Green Channel");
plt.subplot(143);plt.imshow(b,cmap='gray');plt.title("Blue Channel");

# Merge the individual channels into a BGR image
imgMerged = cv2.merge((b,g,r))
# Show the merged output
plt.subplot(144);plt.imshow(imgMerged[:,:,::-1]);plt.title("Merged Outp
```



Converting to different Color Spaces

`cv2.cvtColor()` Converts an image from one color space to another. The function converts an input image from one color space to another. In case of a transformation to-from RGB color space, the order of the channels should be specified explicitly (RGB or BGR). Note that the default color format in OpenCV is often referred to as RGB but it is actually BGR (the bytes are reversed). So the first byte in a standard (24-bit) color image will be an 8-bit Blue component, the second byte will be Green, and the third byte will be Red. The fourth, fifth, and sixth bytes would then be the second pixel (Blue, then Green, then Red), and so on.

Function Syntax

```
dst = cv2.cvtColor( src, code )
```

`dst` : Is the output image of the same size and depth as `src` .

The function has **2 required arguments**:

1. `src` input image: 8-bit unsigned, 16-bit unsigned (CV_16UC...), or single-precision floating-point.
2. `code` color space conversion code (see ColorConversionCodes).

OpenCV Documentation

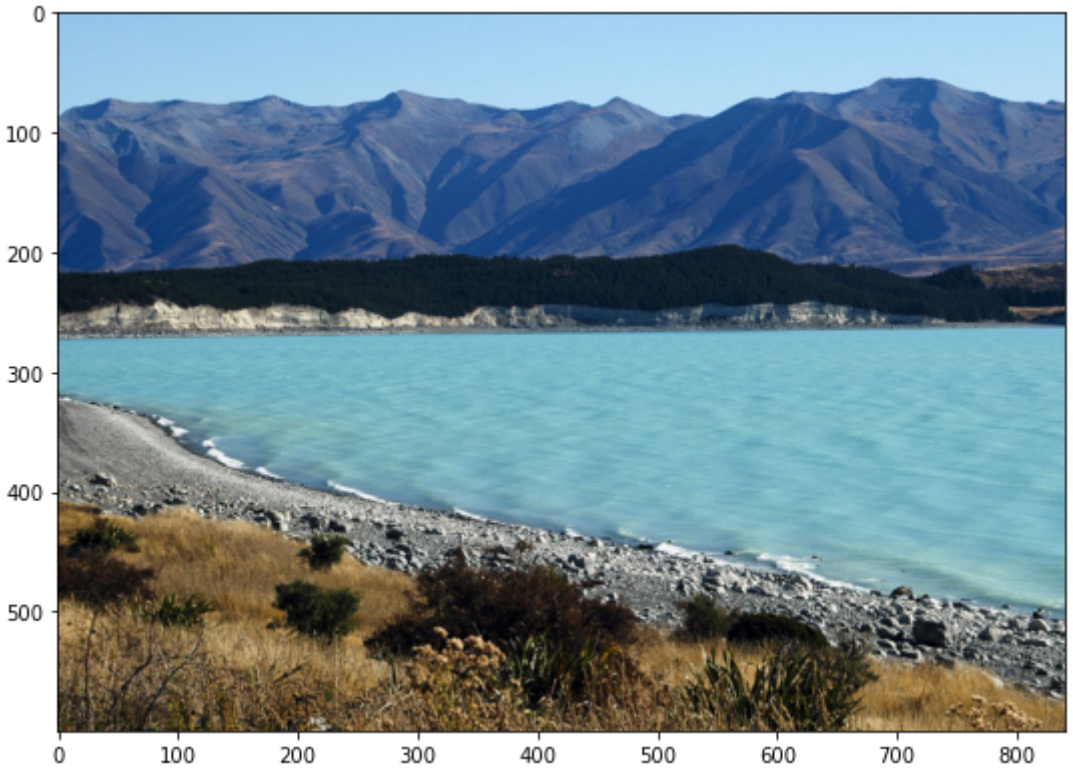
`cv2.cvtColor`:
https://docs.opencv.org/3.4/d8/d01/group__imgproc__color__conversions.html#ga397ae87e1288a8
(https://docs.opencv.org/3.4/d8/d01/group__imgproc__color__conversions.html#ga397ae87e1288a8

ColorConversionCodes:

Changing from BGR to RGB

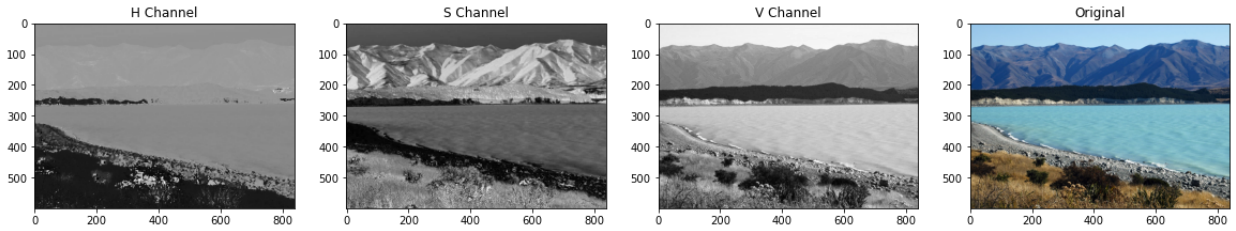
In [17]:
```
# OpenCV stores color channels in a differnet order than most other app
img_NZ_rgb = cv2.cvtColor(img_NZ_bgr, cv2.COLOR_BGR2RGB)
plt.imshow(img_NZ_rgb)
```

Out[17]: <matplotlib.image.AxesImage at 0x7fbe48275550>



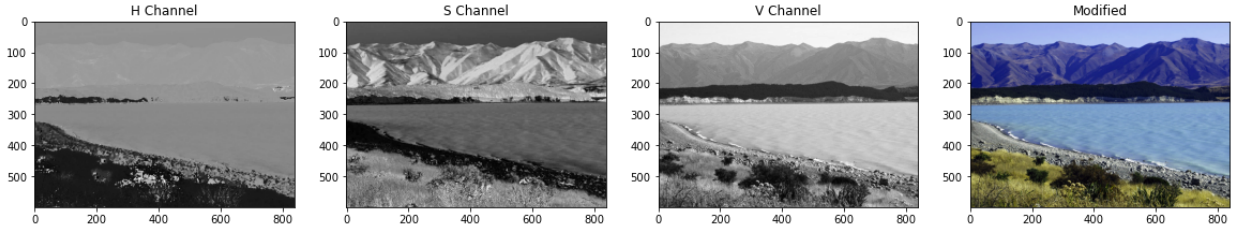
Changing to HSV color space

In [18]:
```
img_hsv = cv2.cvtColor(img_NZ_bgr, cv2.COLOR_BGR2HSV)
# Split the image into the B,G,R components
h,s,v = cv2.split(img_hsv)

# Show the channels
plt.figure(figsize=[20,5])
plt.subplot(141);plt.imshow(h,cmap='gray');plt.title("H Channel");
plt.subplot(142);plt.imshow(s,cmap='gray');plt.title("S Channel");
plt.subplot(143);plt.imshow(v,cmap='gray');plt.title("V Channel");
plt.subplot(144);plt.imshow(img_NZ_rgb);plt.title("Original");

```



Modifying individual Channel

In [19]:
```
h_new = h+10
img_NZ_merged = cv2.merge((h_new,s,v))
img_NZ_rgb = cv2.cvtColor(img_NZ_merged, cv2.COLOR_HSV2RGB)

# Show the channels
plt.figure(figsize=[20,5])
plt.subplot(141);plt.imshow(h,cmap='gray');plt.title("H Channel");
plt.subplot(142);plt.imshow(s,cmap='gray');plt.title("S Channel");
plt.subplot(143);plt.imshow(v,cmap='gray');plt.title("V Channel");
plt.subplot(144);plt.imshow(img_NZ_rgb);plt.title("Modified");
```



Saving Images

Saving the image is as trivial as reading an image in OpenCV. We use the function **`cv2.imwrite()`** with two arguments. The first one is the filename, second argument is the image object.

The function imwrite saves the image to the specified file. The image format is chosen based on the filename extension (see cv::imread for the list of extensions). In general, only 8-bit single-channel or 3-channel (with 'BGR' channel order) images can be saved using this function (see the OpenCV documentation for further details).

Function Syntax

```
cv2.imwrite( filename, img[, params] )
```

The function has **2 required arguments**:

1. `filename` : This can be an **absolute** or **relative** path.
2. `img` : Image or Images to be saved.

OpenCV Documentation

Imwrite:
https://docs.opencv.org/4.5.1/d4/da8/group__imgcodecs.html#gabbc7ef1aa2edfaa87772f1202d67e
(https://docs.opencv.org/4.5.1/d4/da8/group__imgcodecs.html#gabbc7ef1aa2edfaa87772f1202d67e

In [20]:
```
# save the image
cv2.imwrite("New_Zealand_Lake_SAVED.png", img_NZ_bgr)

Image(filename='New_Zealand_Lake_SAVED.png')
```

Out[20]:

In [21]:

```
# read the image as Color
img_NZ_bgr = cv2.imread("New_Zealand_Lake_SAVED.png", cv2.IMREAD_COLOR)
print("img_NZ_bgr shape is: ", img_NZ_bgr.shape)

# read the image as Grayscaled
img_NZ_gry = cv2.imread("New_Zealand_Lake_SAVED.png", cv2.IMREAD_GRAYSC
print("img_NZ_gry shape is: ", img_NZ_gry.shape)
```

```
img_NZ_bgr shape is:  (600, 840, 3)
img_NZ_gry shape is:  (600, 840)
```

Basic Image Manipulations

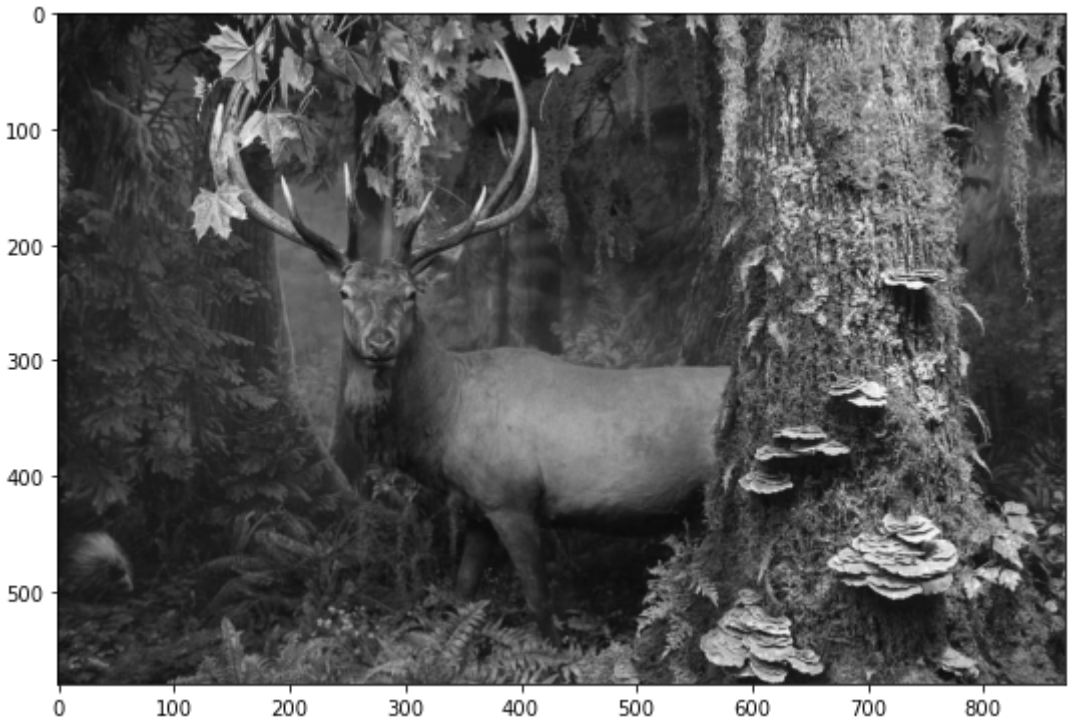
In this notebook we will cover how to perform image transformations including:

- Accessing and manipulating images pixels
- Image resizing
- Cropping
- Flipping

Original checkerboard image

In [23]:
```
# Read image as gray scale.
cb_img = cv2.imread("imgs/yourImage.png",0)

# Set color map to gray scale for proper rendering.
plt.imshow(cb_img, cmap='gray')
print(cb_img)
```

```
[[ 34  30  26 ... 143 109 122]
 [ 33  29  26 ... 144  90  76]
 [ 30  30  27 ... 124  80  64]
 ...
 [ 24  23  25 ... 100  97  70]
 [ 26  21  20 ...  74 103 100]
 [ 32  26  21 ...  52  83  83]]
```



Accessing Individual Pixels

Let us see how to access a pixel in the image.

For accessing any pixel in a numpy matrix, you have to use matrix notation such as matrix[r,c], where the r is the row number and c is the column number. Also note that the matrix is 0-indexed.

For example, if you want to access the first pixel, you need to specify matrix[0,0]. Let us see with some examples. We will print one black pixel from top-left and one white pixel from top-center.

In [24]:
```
# print the first pixel of the first black box
print(cb_img[0,0])
# print the first white pixel to the right of the first black box
print(cb_img[0,6])
```

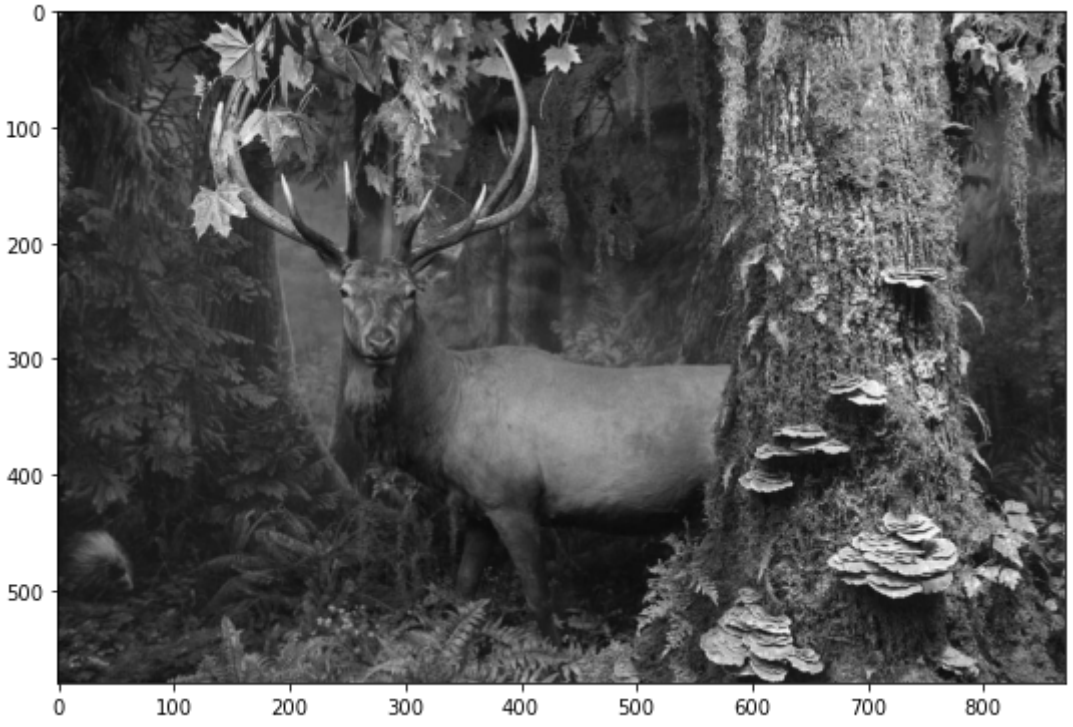
```
34
24
```

Modifying Image Pixels

We can modify the intensity values of pixels in the same manner as described above.

In [25]:
```
cb_img_copy = cb_img.copy()
cb_img_copy[2,2] = 200
cb_img_copy[2,3] = 200
cb_img_copy[3,2] = 200
cb_img_copy[3,3] = 200

# Same as above
# cb_img_copy[2:3,2:3] = 200

plt.imshow(cb_img_copy, cmap='gray')
print(cb_img_copy)
```

```
[[ 34  30  26 ... 143 109 122]
 [ 33  29  26 ... 144  90  76]
 [ 30  30 200 ... 124  80  64]
 ...
 [ 24  23  25 ... 100  97  70]
 [ 26  21  20 ...  74 103 100]
 [ 32  26  21 ...  52  83  83]]
```



Cropping Images

Cropping an image is simply achieved by selecting a specific (pixel) region of the image.

In [27]:
```
1 img_NZ_bgr = cv2.imread("imgs/New_Zealand_Boat.jpg",cv2.IMREAD_COLOR)
2 img_NZ_rgb = img_NZ_bgr[:,:,::-1]
3
4 plt.imshow(img_NZ_rgb)
```

Out[27]: <matplotlib.image.AxesImage at 0x7fbe4824e430>

Crop out the middle region of the image

In [28]:
```
cropped_region = img_NZ_rgb[200:400, 300:600]
plt.imshow(cropped_region)
```

Out[28]: <matplotlib.image.AxesImage at 0x7fbe6963e490>

Resizing Images

The function **`resize`** resizes the image `src` down to or up to the specified size. The size and type are derived from the `src` , `dsize` , `fx` , and `fy` .

Function Syntax

```
dst = resize( src, dsize[, dst[, fx[, fy[, interpolation]]]] )
```

`dst` : output image; it has the size dsize (when it is non-zero) or the size computed from src.size(), fx, and fy; the type of dst is the same as of src.

The function has **2 required arguments**:

1. `src` : input image
2. `dsize` : output image size

Optional arguments that are often used include:

1. `fx` : Scale factor along the horizontal axis; when it equals 0, it is computed as `(double)dsize.width/src.cols`

2. `fy` : Scale factor along the vertical axis; when it equals 0, it is computed as (double)dsize.height/src.rows

The output image has the size `dsize` (when it is non-zero) or the size computed from `src.size()` , `fx` , and `fy` ; the type of dst is the same as of src.

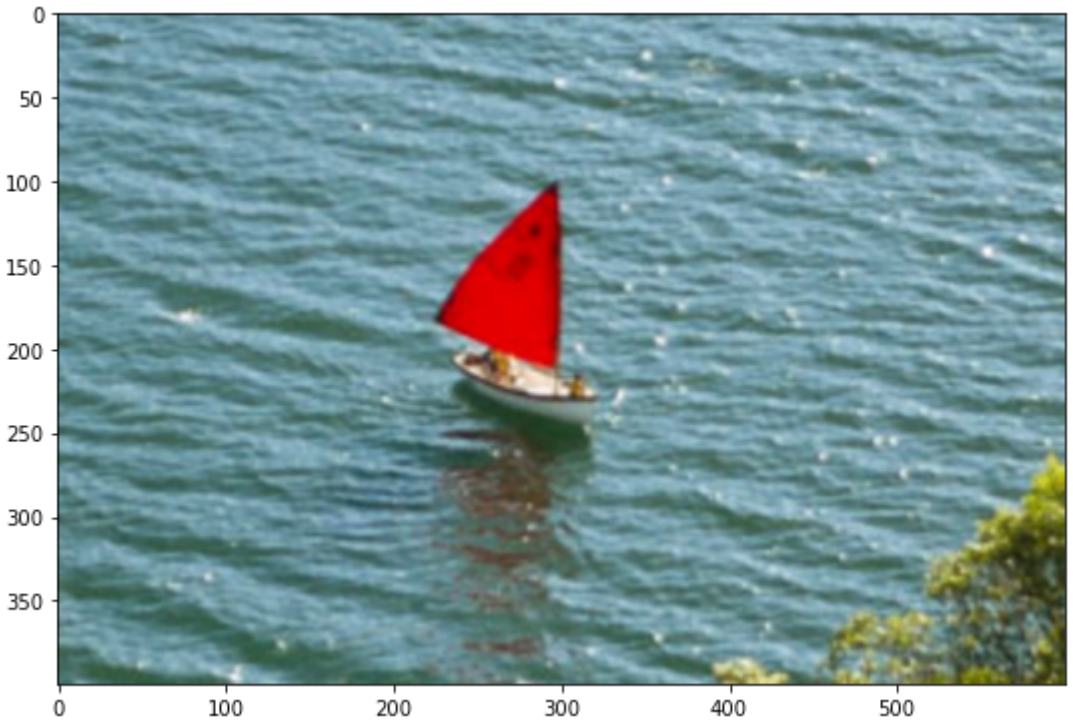
OpenCV Documentation

`resize()`:
https://docs.opencv.org/4.5.0/da/d54/group__imgproc__transform.html#ga47a974309e9102f5f08231

Method 1: Specifying Scaling Factor using fx and fy

In [29]:
```
1 resized_cropped_region_2x = cv2.resize(cropped_region,None,fx=2, fy=2)
2 plt.imshow(resized_cropped_region_2x)
```

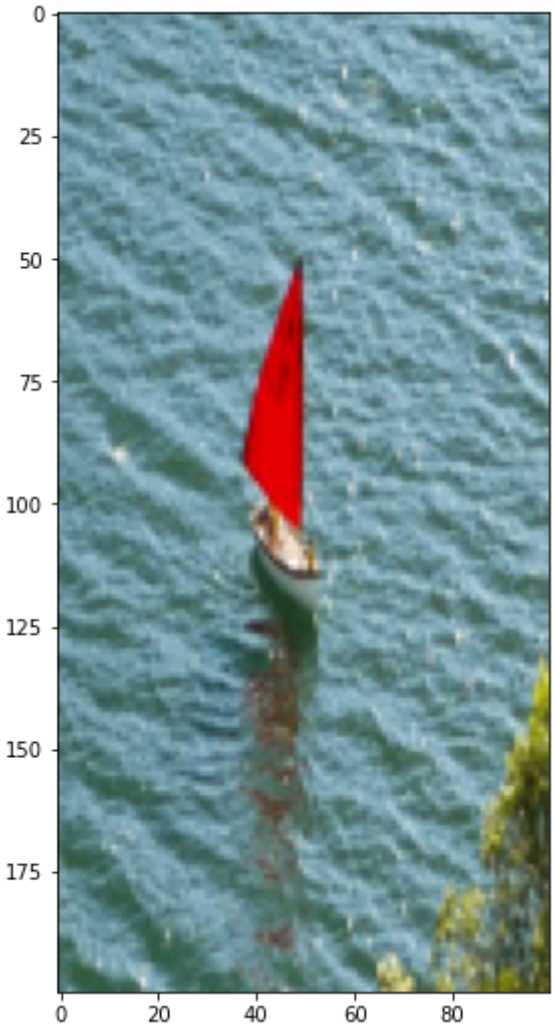
Out[29]: `<matplotlib.image.AxesImage at 0x7fbe6994b6d0>`



Medthod 2: Specifying exact size of the output image

In [30]:
```
desired_width = 100
desired_height = 200
dim = (desired_width, desired_height)

# Resize background image to sae size as logo image
resized_cropped_region = cv2.resize(cropped_region, dsize=dim, interpol
plt.imshow(resized_cropped_region)
```

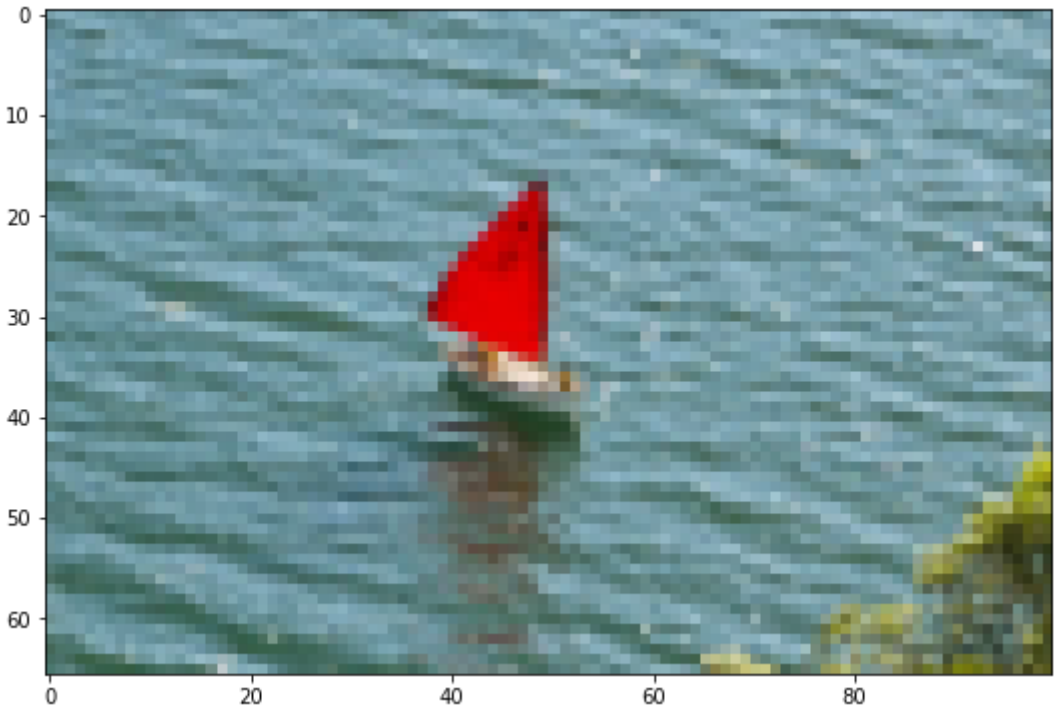
Out[30]: <matplotlib.image.AxesImage at 0x7fbe99185a30>



Resize while maintaining aspect ratio

In [31]:
```
# Method 2: Using 'dsize'
desired_width = 100
aspect_ratio = desired_width / cropped_region.shape[1]
desired_height = int(cropped_region.shape[0] * aspect_ratio)
dim = (desired_width, desired_height)

# Resize image
resized_cropped_region = cv2.resize(cropped_region, dsize=dim, interpol
plt.imshow(resized_cropped_region)
```

Out[31]: <matplotlib.image.AxesImage at 0x7fbe992f4c70>



Let's actually show the (cropped) resized image.

In [32]:
```
# Swap channel order
resized_cropped_region_2x = resized_cropped_region_2x[:,:,::-1]

# Save resized image to disk
cv2.imwrite("resized_cropped_region_2x.png", resized_cropped_region_2x)

# Display the cropped and resized image
Image(filename='resized_cropped_region_2x.png')
```

Out[32]:

In [33]:
```
# Swap channel order
cropped_region = cropped_region[:,:,::-1]

# Save cropped 'region'
cv2.imwrite("cropped_region.png", cropped_region)

# Display the cropped and resized image
Image(filename='cropped_region.png')
```

Out[33]:

Flipping Images

The function **`flip`** flips the array in one of three different ways (row and column indices are 0-based):

Function Syntax

```
dst = cv.flip( src, flipCode )
```

`dst` : output array of the same size and type as src.

The function has **2 required arguments**:

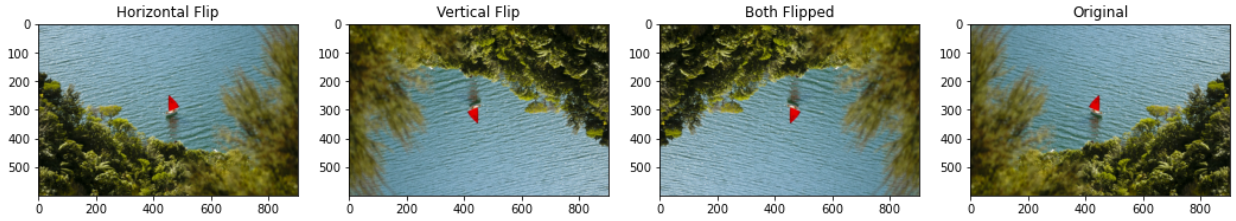
1. `src` : input image
2. `flipCode` : a flag to specify how to flip the array; 0 means flipping around the x-axis and positive value (for example, 1) means flipping around y-axis. Negative value (for example, -1) means flipping around both axes.

OpenCV Documentation

`flip`:
https://docs.opencv.org/4.5.0/d2/de8/group__core__array.html#gaca7be533e3dac7feb70fc60635adf
(https://docs.opencv.org/4.5.0/d2/de8/group__core__array.html#gaca7be533e3dac7feb70fc60635ad

In [34]:

```
img_NZ_rgb_flipped_horz = cv2.flip(img_NZ_rgb, 1)
img_NZ_rgb_flipped_vert = cv2.flip(img_NZ_rgb, 0)
img_NZ_rgb_flipped_both = cv2.flip(img_NZ_rgb, -1)

# Show the images
plt.figure(figsize=[18,5])
plt.subplot(141);plt.imshow(img_NZ_rgb_flipped_horz);plt.title("Horizon
plt.subplot(142);plt.imshow(img_NZ_rgb_flipped_vert);plt.title("Vertica
plt.subplot(143);plt.imshow(img_NZ_rgb_flipped_both);plt.title("Both Fl
plt.subplot(144);plt.imshow(img_NZ_rgb);plt.title("Original");
```



Annotating Images

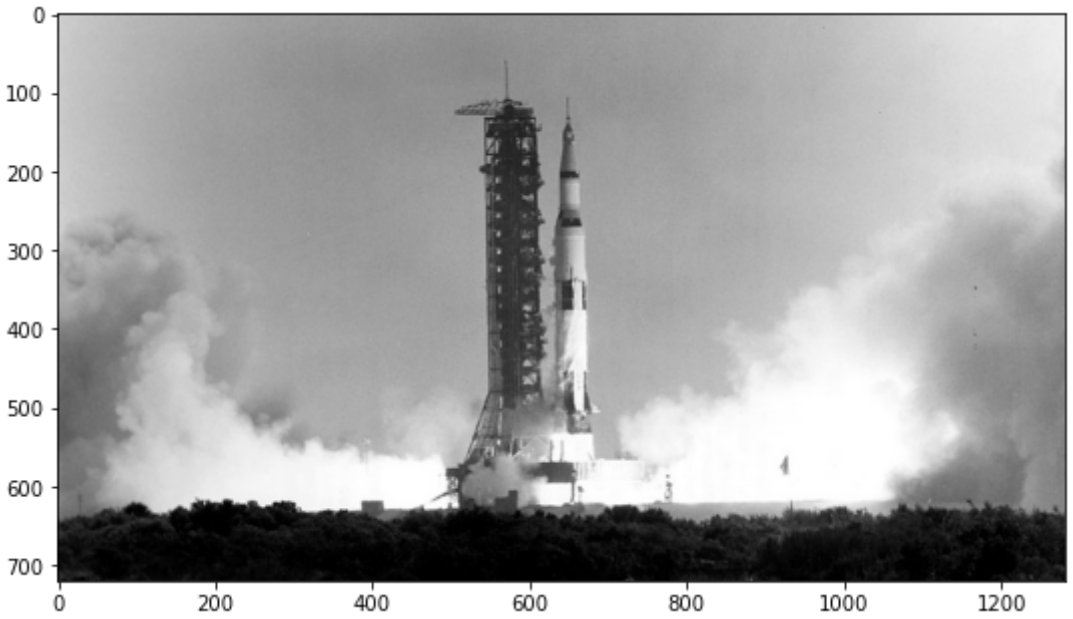
In this notebook we will cover how to annotate images using OpenCV. We will learn how to peform the following annotations to images.

- Draw lines
- Draw circles
- Draw rectangles
- Add text

These are useful when you want to annotate your results for presentations or show a demo of your application. Annotations can also be useful during development and debugging.

In [35]:
```
# Read in an image
image = cv2.imread("imgs/Apollo_11_Launch.jpg", cv2.IMREAD_COLOR)

# Display the original image
plt.imshow(image[:,:,::-1])
```

Out[35]: <matplotlib.image.AxesImage at 0x7fbe9910e4c0>



Drawing a Line

Let's start off by drawing a line on an image. We will use cv2.line function for this.

Function Syntx

```
    img = cv2.line(img, pt1, pt2, color[, thickness[, lineType[, sh
ift]]])
```

`img` : The output image that has been annotated.

The function has **4 required arguments**:

1. `img` : Image on which we will draw a line
2. `pt1` : First point(x,y location) of the line segment
3. `pt2` : Second point of the line segment
4. `color` : Color of the line which will be drawn

Other optional arguments that are important for us to know include:

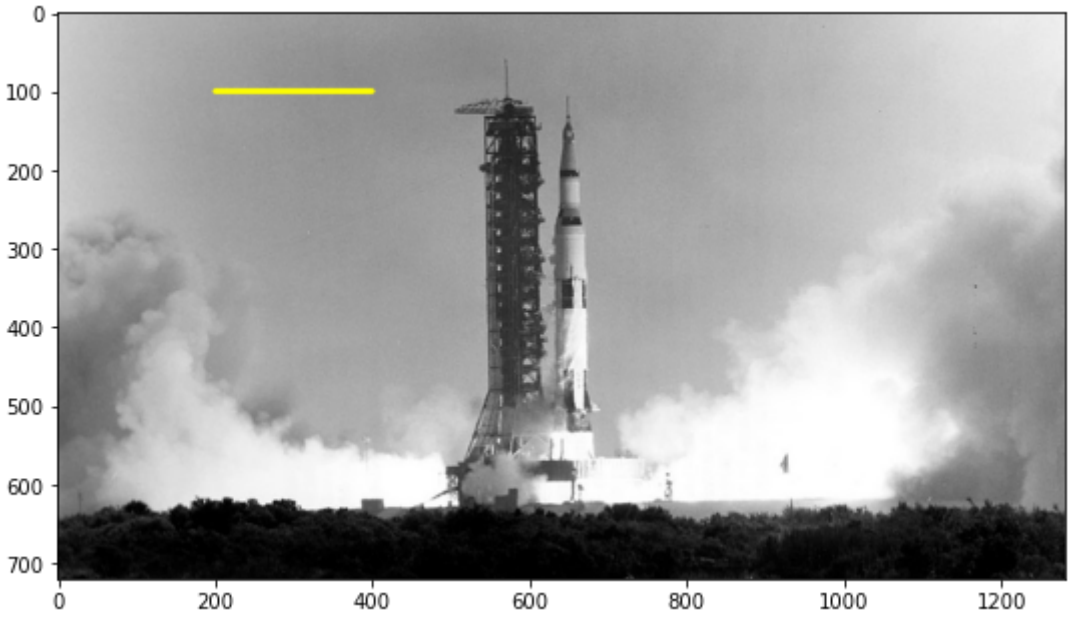
1. `thickness` : Integer specifying the line thickness. Default value is 1.
2. `lineType` : Type of the line. Default value is 8 which stands for an 8-connected line. Usually, cv2.LINE_AA (antialiased or smooth line) is used for the lineType.

OpenCV Documentation

line: https://docs.opencv.org/4.5.1/d6/d6e/group__imgproc__draw.html#ga7078a9fae8c7e7d13d (https://docs.opencv.org/4.5.1/d6/d6e/group__imgproc__draw.html#ga7078a9fae8c7e7d13d24dac2

In [36]:
```
imageLine = image.copy()

# The line starts from (200,100) and ends at (400,100)
# The color of the line is YELLOW (Recall that OpenCV uses BGR format)
# Thickness of line is 5px
# Linetype is cv2.LINE_AA

cv2.line(imageLine, (200, 100), (400, 100), (0, 255, 255), thickness=5,

# Display the image
plt.imshow(imageLine[:,:,::-1])
```

Out[36]: <matplotlib.image.AxesImage at 0x7fbe4828ff40>



Drawing a Circle

Let's start off by drawing a circle on an image. We will use cv2.circle function for this.

Functional syntx

```
img = cv2.circle(img, center, radius, color[, thickness[, lineType
[, shift]]])
```

`img` : The output image that has been annotated.

The function has **4 required arguments**:

1. `img` : Image on which we will draw a line
2. `center` : Center of the circle

3. `radius` : Radius of the circle
4. `color` : Color of the circle which will be drawn

Next, let's check out the (optional) arguments which we are going to use quite extensively.

1. `thickness` : Thickness of the circle outline (if positive). If a negative value is supplied for this argument, it will result in a filled circle.
2. `lineType` : Type of the circle boundary. This is exact same as lineType argument in **cv2.line**

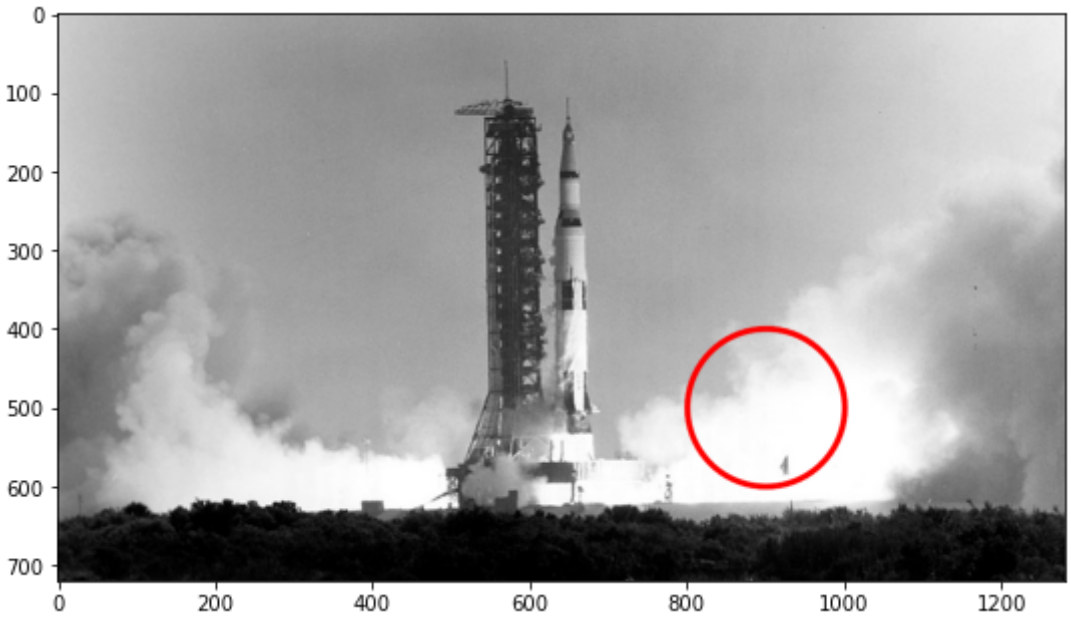
OpenCV Documentation

circle:
https://docs.opencv.org/4.5.1/d6/d6e/group__imgproc__draw.html#gaf10604b069374903dbd0f0488
(https://docs.opencv.org/4.5.1/d6/d6e/group__imgproc__draw.html#gaf10604b069374903dbd0f048

Let's see an example of this

In [37]:
```
# Draw a circle
imageCircle = image.copy()

cv2.circle(imageCircle, (900,500), 100, (0, 0, 255), thickness=5, lineT

# Display the image
plt.imshow(imageCircle[:,:,::-1])
```

Out[37]: <matplotlib.image.AxesImage at 0x7fbe6853e9a0>



Drawing a Rectangle

We will use **cv2.rectangle** function to draw a rectangle on an image. The function syntax is as follows.

Functional syntx

```
img = cv2.rectangle(img, pt1, pt2, color[, thickness[, lineType[, s
hift]]])
```

`img` : The output image that has been annotated.

The function has **4 required arguments**:

1. `img` : Image on which the rectangle is to be drawn.
2. `pt1` : Vertex of the rectangle. Usually we use the **top-left vertex** here.
3. `pt2` : Vertex of the rectangle opposite to pt1. Usually we use the **bottom-right** vertex here.
4. `color` : Rectangle color

Next, let's check out the (optional) arguments which we are going to use quite extensively.

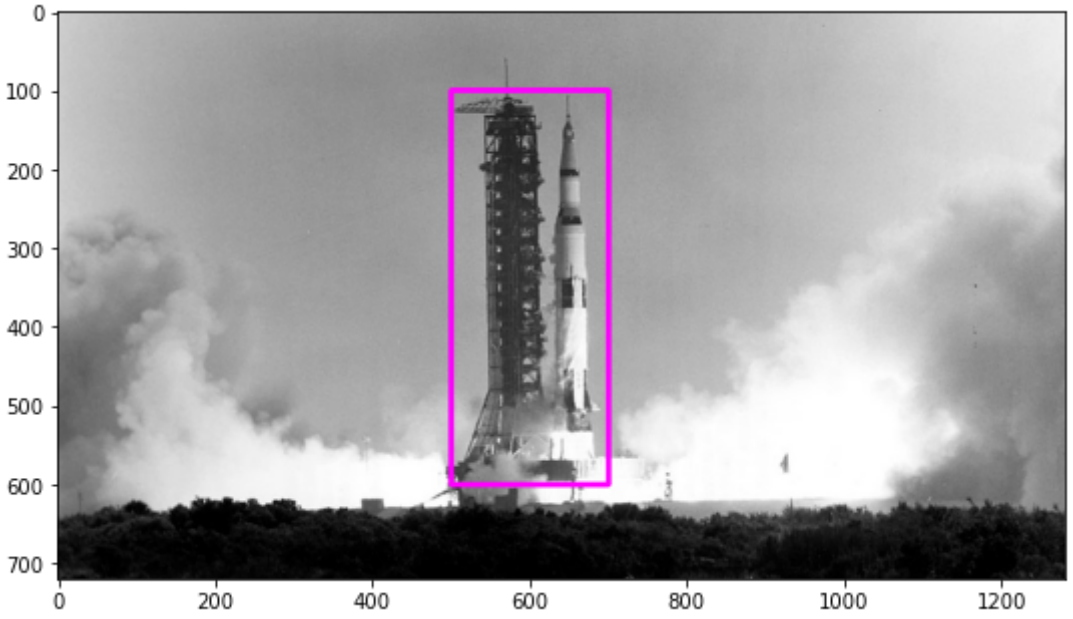
1. `thickness` : Thickness of the circle outline (if positive). If a negative value is supplied for this argument, it will result in a filled rectangle.
2. `lineType` : Type of the circle boundary. This is exact same as lineType argument in **cv2.line**

OpenCV Documentation Links

`rectangle:` https://docs.opencv.org/4.5.1/d6/d6e/group__imgproc__draw.html#ga07d2f74cadcf8 (https://docs.opencv.org/4.5.1/d6/d6e/group__imgproc__draw.html#ga07d2f74cadcf8e305e810ce8e

In [38]:
```
# Draw a rectangle (thickness is a positive integer)
imageRectangle = image.copy()

cv2.rectangle(imageRectangle, (500, 100), (700,600), (255, 0, 255), thi

# Display the image
plt.imshow(imageRectangle[:,:,::-1])
```

Out[38]: `<matplotlib.figure.AxesImage at 0x7fbe68545790>`



Adding Text

Finally, let's see how we can write some text on an image using **cv2.putText** function.

Functional syntx

```
img = cv2.putText(img, text, org, fontFace, fontScale, color[, thic
kness[, lineType[, bottomLeftOrigin]]])
```

`img` : The output image that has been annotated.

The function has **6 required arguments**:

1. `img` : Image on which the text has to be written.
2. `text` : Text string to be written.
3. `org` : Bottom-left corner of the text string in the image.
4. `fontFace` : Font type
5. `fontScale` : Font scale factor that is multiplied by the font-specific base size.
6. `color` : Font color

Other optional arguments that are important for us to know include:

1. `thickness` : Integer specifying the line thickness for the text. Default value is 1.
2. `lineType` : Type of the line. Default value is 8 which stands for an 8-connected line. Usually, cv2.LINE_AA (antialiased or smooth line) is used for the lineType.

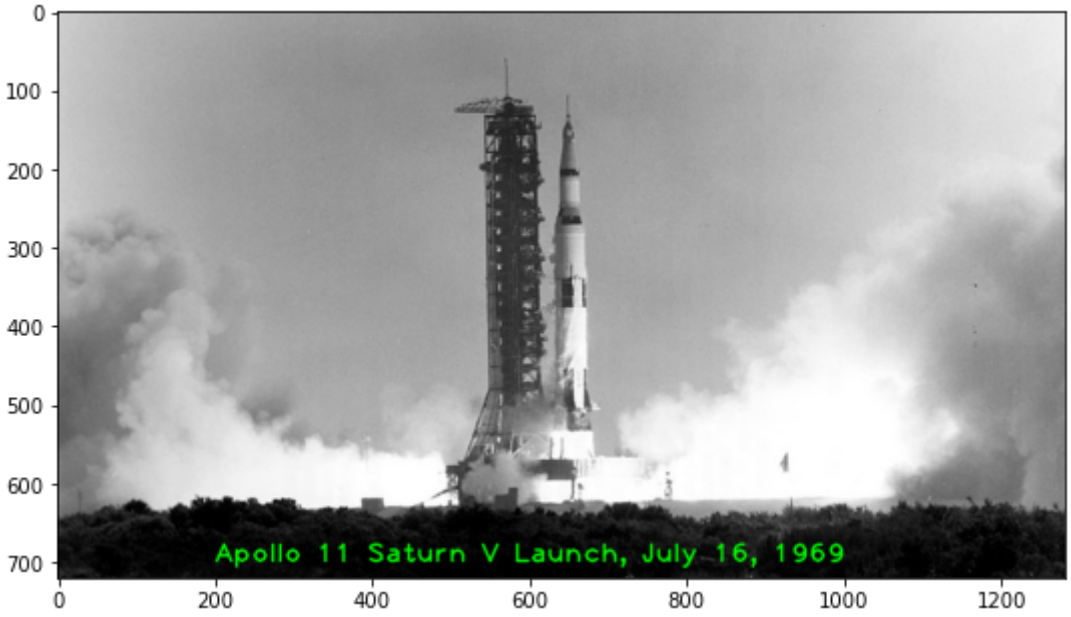
OpenCV Documentation

`putText:` https://docs.opencv.org/4.5.1/d6/d6e/group__imgproc__draw.html#ga5126f47f883d730
(https://docs.opencv.org/4.5.1/d6/d6e/group__imgproc__draw.html#ga5126f47f883d730f633d74f07

Let's see an example of this.

In [39]:
```
imageText = image.copy()
text = "Apollo 11 Saturn V Launch, July 16, 1969"
fontScale = 2.3
fontFace = cv2.FONT_HERSHEY_PLAIN
fontColor = (0, 255, 0)
fontThickness = 2

cv2.putText(imageText, text, (200, 700), fontFace, fontScale, fontColor

# Display the image
plt.imshow(imageText[:,:,::-1])
```

Out[39]: <matplotlib.figure.AxesImage at 0x7fbe69738880>



HDR

Basic Image Enhancement Using Mathematical Operations

Image Processing techniques take advantage of mathematical operations to achieve different results. Most often we arrive at an enhanced version of the image using some basic operations. We will take a look at some of the fundamental operations often used in computer vision pipelines. In this notebook we will cover:

- Arithmetic Operations like addition, multiplication
- Thresholding & Masking
- Bitwise Operations like OR, AND, XOR

Original image

In [41]:
```
1 img_bgr = cv2.imread("imgs/New_Zealand_Coast.jpg",cv2.IMREAD_COLOR)
2 img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
3
4 # Display 18x18 pixel image.
5 Image(filename='imgs/New_Zealand_Coast.jpg')
```

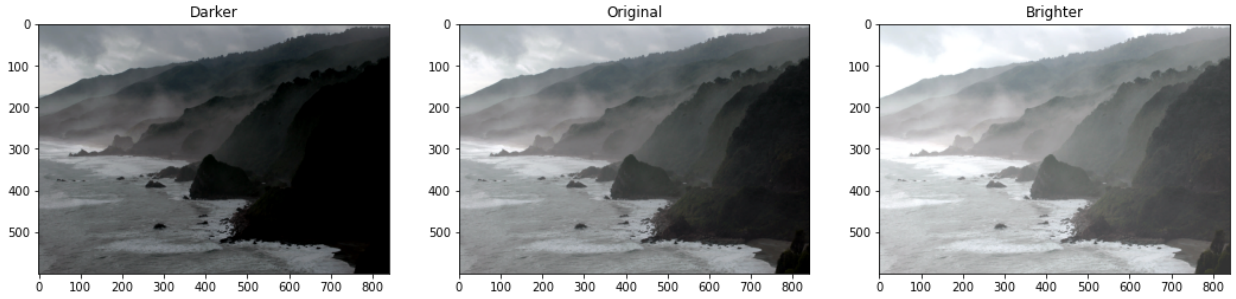
Out[41]:

Addition or Brightness

The first operation we discuss is simple addition of images. This results in increasing or decreasing the brightness of the image since we are eventually increasing or decreasing the intensity values of each pixel by the same amount. So, this will result in a global increase/decrease in brightness.

In [42]:

```
matrix = np.ones(img_rgb.shape, dtype = "uint8") * 50

img_rgb_brighter = cv2.add(img_rgb, matrix)
img_rgb_darker   = cv2.subtract(img_rgb, matrix)

# Show the images
plt.figure(figsize=[18,5])
plt.subplot(131); plt.imshow(img_rgb_darker);  plt.title("Darker");
plt.subplot(132); plt.imshow(img_rgb);         plt.title("Original");
plt.subplot(133); plt.imshow(img_rgb_brighter);plt.title("Brighter");
```



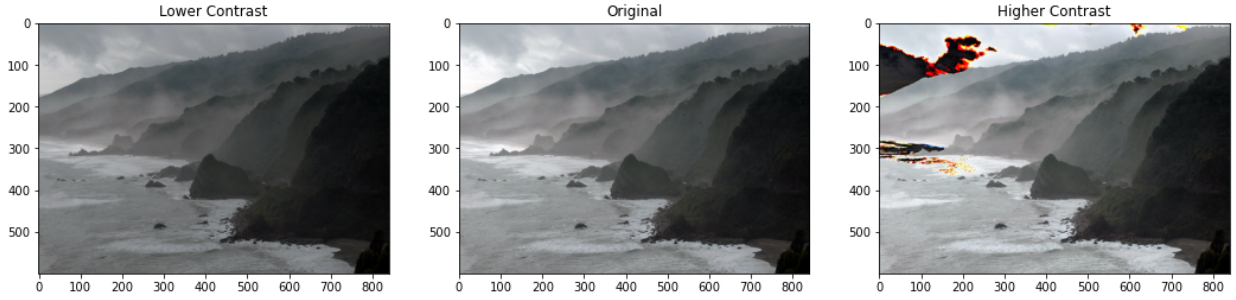
Multiplication or Contrast

Just like addition can result in brightness change, multiplication can be used to improve the contrast of the image.

Contrast is the difference in the intensity values of the pixels of an image. Multiplying the intensity values with a constant can make the difference larger or smaller (if multiplying factor is < 1).

In [43]:

```
matrix1 = np.ones(img_rgb.shape) * .8
matrix2 = np.ones(img_rgb.shape) * 1.2

img_rgb_darker   = np.uint8(cv2.multiply(np.float64(img_rgb), matrix1))
img_rgb_brighter = np.uint8(cv2.multiply(np.float64(img_rgb), matrix2))

# Show the images
plt.figure(figsize=[18,5])
plt.subplot(131); plt.imshow(img_rgb_darker);  plt.title("Lower Contras
plt.subplot(132); plt.imshow(img_rgb);         plt.title("Original");
plt.subplot(133); plt.imshow(img_rgb_brighter);plt.title("Higher Contra
```



What happened?

Can you see the weird colors in some areas of the image after multiplication?

The issue is that after multiplying, the values which are already high, are becoming greater than 255. Thus, the overflow issue. How do we overcome this?

Handling Overflow using np.clip

In [44]:
```
matrix1 = np.ones(img_rgb.shape) * .8
matrix2 = np.ones(img_rgb.shape) * 1.2

img_rgb_lower   = np.uint8(cv2.multiply(np.float64(img_rgb), matrix1))
img_rgb_higher  = np.uint8(np.clip(cv2.multiply(np.float64(img_rgb), ma

# Show the images
plt.figure(figsize=[18,5])
plt.subplot(131); plt.imshow(img_rgb_lower);  plt.title("Lower Contrast
plt.subplot(132); plt.imshow(img_rgb);         plt.title("Original");
plt.subplot(133); plt.imshow(img_rgb_higher);plt.title("Higher Contrast
```

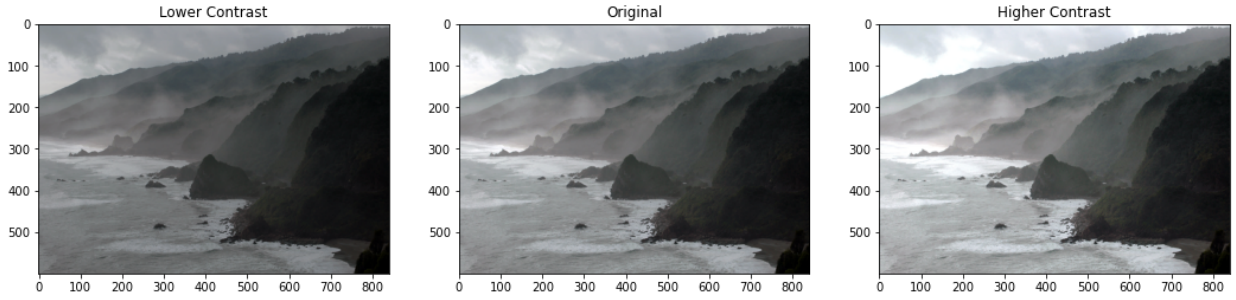


Image Thresholding

Binary Images have a lot of use cases in Image Processing. One of the most common use cases is that of creating masks. Image Masks allow us to process on specific parts of an image keeping the other parts intact. Image Thresholding is used to create Binary Images from grayscale images. You can use different thresholds to create different binary images from the same original image.

Function Syntax

```
retval, dst = cv2.threshold( src, thresh, maxval, type[, dst] )
```

`dst` : The output array of the same size and type and the same number of channels as `src` .

The function has **4 required arguments**:

1. `src` : input array (multiple-channel, 8-bit or 32-bit floating point).
2. `thresh` : threshold value.
3. `maxval` : maximum value to use with the THRESH_BINARY and THRESH_BINARY_INV thresholding types.

4. `type` : thresholding type (see ThresholdTypes).

Function Syntax

```
dst = cv.adaptiveThreshold( src, maxValue, adaptiveMethod, threshol
dType, blockSize, C[, dst] )
```

`dst` Destination image of the same size and the same type as src.

The function has **6 required arguments**:

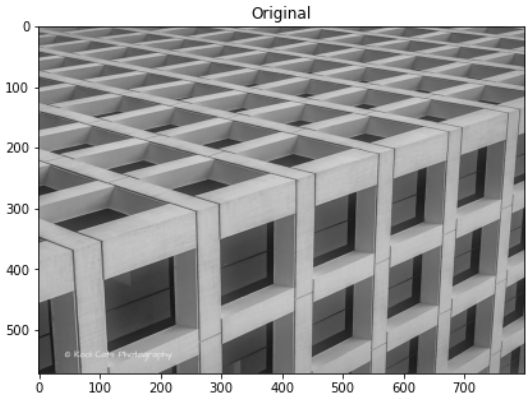
1. `src` : Source 8-bit single-channel image.
2. `maxValue` : Non-zero value assigned to the pixels for which the condition is satisfied
3. `adaptiveMethod` : Adaptive thresholding algorithm to use, see AdaptiveThresholdTypes. The BORDER_REPLICATE | BORDER_ISOLATED is used to process boundaries.
4. `thresholdType:` Thresholding type that must be either THRESH_BINARY or THRESH_BINARY_INV, see ThresholdTypes.
5. `blockSize` : Size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on.
6. `C` : Constant subtracted from the mean or weighted mean (see the details below). Normally, it is positive but may be zero or negative as well.

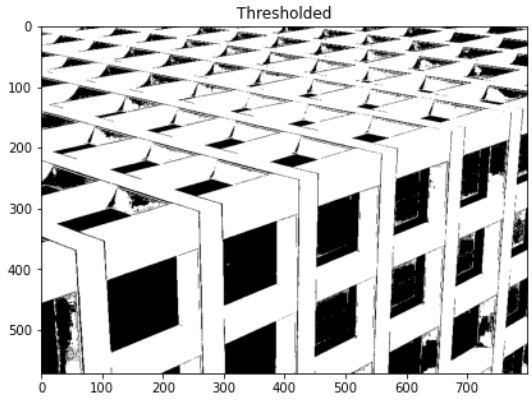
OpenCV Documentation

https://docs.opencv.org/4.5.1/d7/d1b/group__imgproc__misc.html#gae8a4a146d1ca78c626a53577
(https://docs.opencv.org/4.5.1/d7/d1b/group__imgproc__misc.html#gae8a4a146d1ca78c626a53577
https://docs.opencv.org/4.5.1/d7/d4d/tutorial_py_thresholding.html
(https://docs.opencv.org/4.5.1/d7/d4d/tutorial_py_thresholding.html)

In [46]:
```
img_read = cv2.imread("imgs/building-windows.jpg", cv2.IMREAD_GRAYSCALE
retval, img_thresh = cv2.threshold(img_read, 100, 255, cv2.THRESH_BINAR

# Show the images
plt.figure(figsize=[18,5])
plt.subplot(121); plt.imshow(img_read, cmap="gray");          plt.title(
plt.subplot(122); plt.imshow(img_thresh, cmap="gray");        plt.title(

print(img_thresh.shape)
```

```
(572, 800)
```



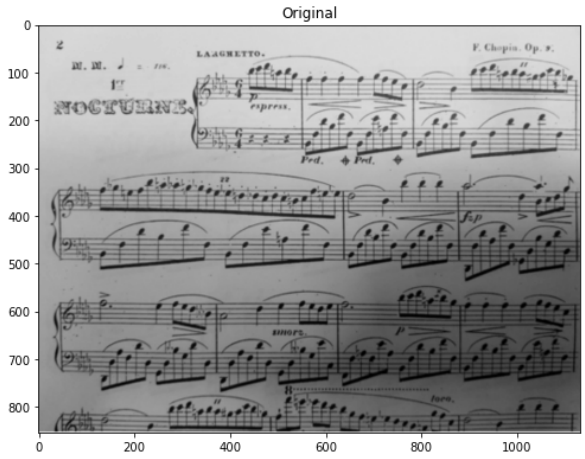


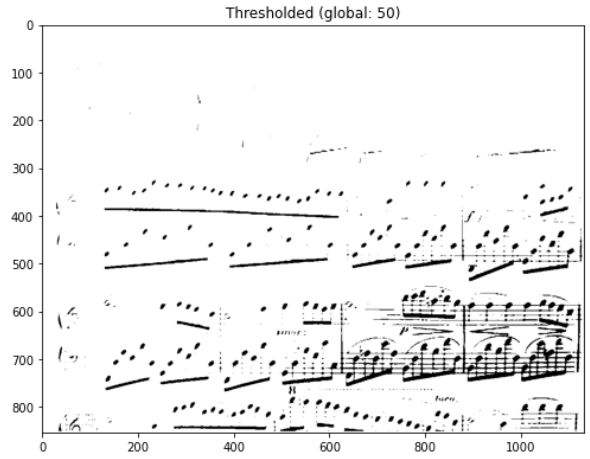
Application: Sheet Music Reader

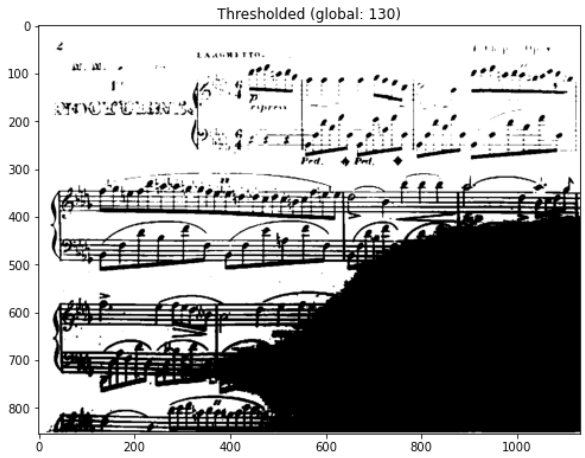
Suppose you wanted to build an application that could read (decode) sheet music. This is similar to Optical Character Recognigition (OCR) for text documents where the goal is to recognize text characters. In either application, one of the first steps in the processing pipeline is to isolate the important information in the image of a document (separating it from the background). This task can be accomplished with thresholding techniques. Let's take a look at an example.

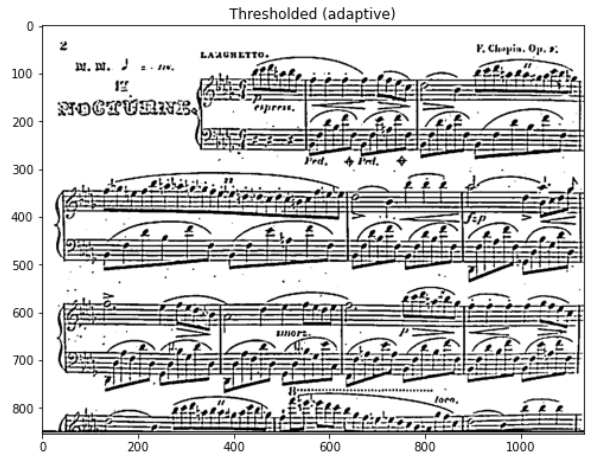
In [48]:

```
# Read the original image
img_read = cv2.imread("imgs/Piano_Sheet_Music.png", cv2.IMREAD_GRAYSCAL

# Perform global thresholding
retval, img_thresh_gbl_1 = cv2.threshold(img_read,50, 255, cv2.THRESH_B

# Perform global thresholding
retval, img_thresh_gbl_2 = cv2.threshold(img_read,130, 255, cv2.THRESH_

# Perform adaptive thresholding
img_thresh_adp = cv2.adaptiveThreshold(img_read, 255, cv2.ADAPTIVE_THRE

# Show the images
plt.figure(figsize=[18,15])
plt.subplot(221); plt.imshow(img_read,        cmap="gray");  plt.title(
plt.subplot(222); plt.imshow(img_thresh_gbl_1,cmap="gray");  plt.title(
plt.subplot(223); plt.imshow(img_thresh_gbl_2,cmap="gray");  plt.title(
plt.subplot(224); plt.imshow(img_thresh_adp,  cmap="gray");  plt.title(
```









Bitwise Operations

Function Syntax

Example API for `cv2.bitwise_and()`. Others include: `cv2.bitwise_or()`, `cv2.bitwise_xor()`, `cv2.bitwise_not()`

```
dst = cv2.bitwise_and( src1, src2[, dst[, mask]] )
```

`dst` : Output array that has the same size and type as the input arrays.

The function has **2 required arguments**:

1. `src1` : first input array or a scalar.
2. `src2` : second input array or a scalar.

An important optional argument is:

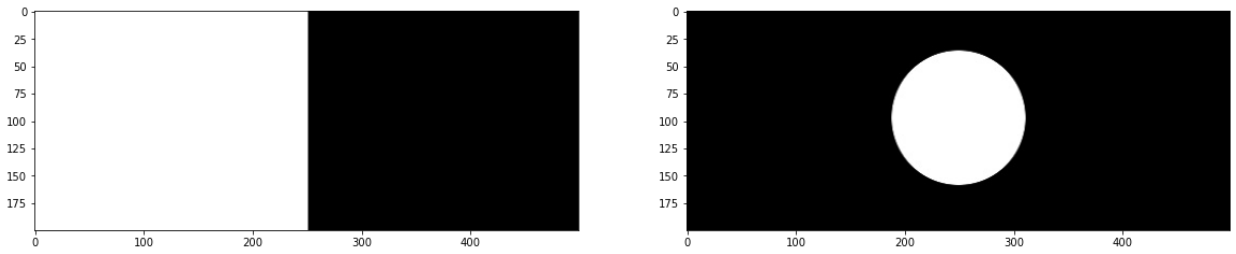
1. `mask` : optional operation mask, 8-bit single channel array, that specifies elements of the output array to be changed.

OpenCV Documentation

https://docs.opencv.org/4.5.1/d0/d86/tutorial_py_image_arithmetics.html
(https://docs.opencv.org/4.5.1/d0/d86/tutorial_py_image_arithmetics.html)
https://docs.opencv.org/4.5.0/d2/de8/group__core__array.html#ga60b4d04b251ba5eb1392c344254
(https://docs.opencv.org/4.5.0/d2/de8/group__core__array.html#ga60b4d04b251ba5eb1392c344254

In [51]:
```
1  img_rec = cv2.imread("imgs/rectangle.jpg", cv2.IMREAD_GRAYSCALE)
2
3  img_cir = cv2.imread("imgs/circle.jpg", cv2.IMREAD_GRAYSCALE)
4
5  plt.figure(figsize=[20,5])
6  plt.subplot(121);plt.imshow(img_rec,cmap='gray')
7  plt.subplot(122);plt.imshow(img_cir,cmap='gray')
8  print(img_rec.shape)
```

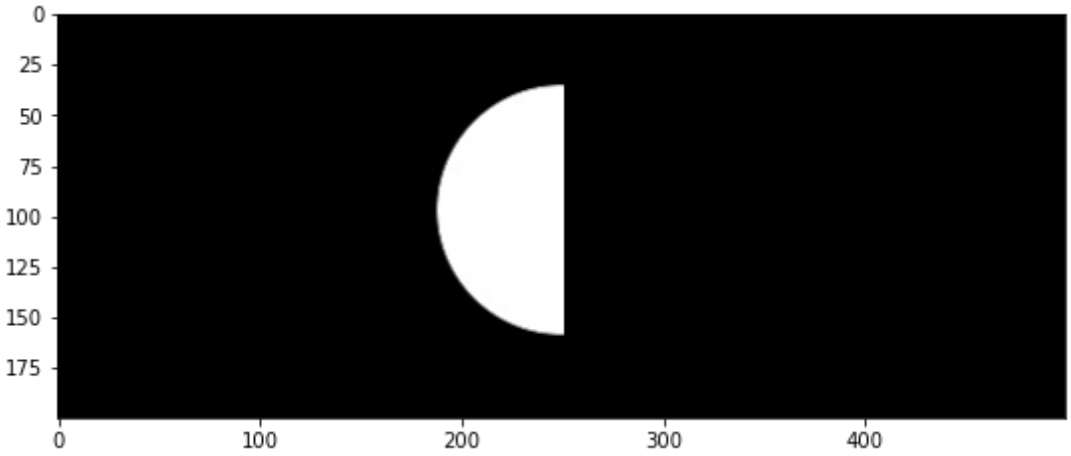
```
(200, 499)
```



Bitwise AND Operator

In [52]:
```
result = cv2.bitwise_and(img_rec, img_cir, mask = None)
plt.imshow(result,cmap='gray')
```

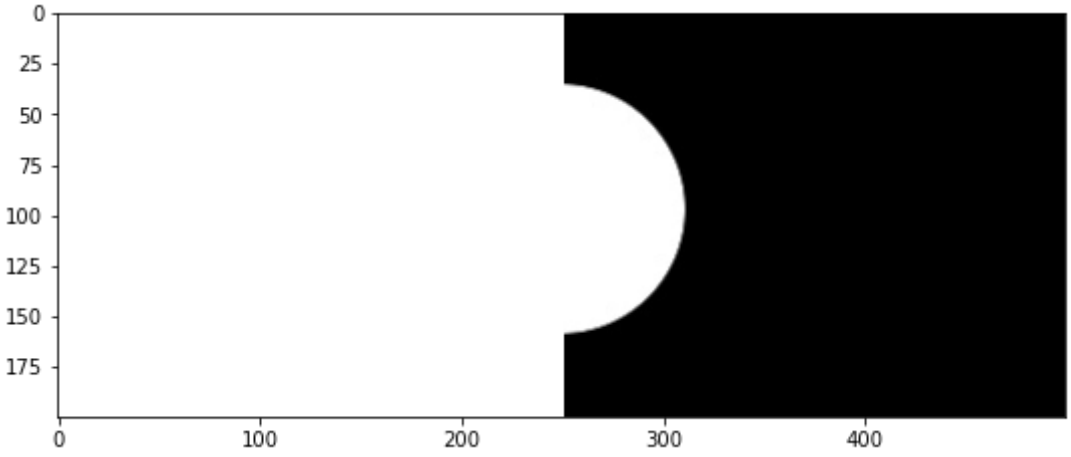
Out[52]: <matplotlib.image.AxesImage at 0x7fbe9929f850>



Bitwise OR Operator

In [53]:
```
result = cv2.bitwise_or(img_rec, img_cir, mask = None)
plt.imshow(result,cmap='gray')
```

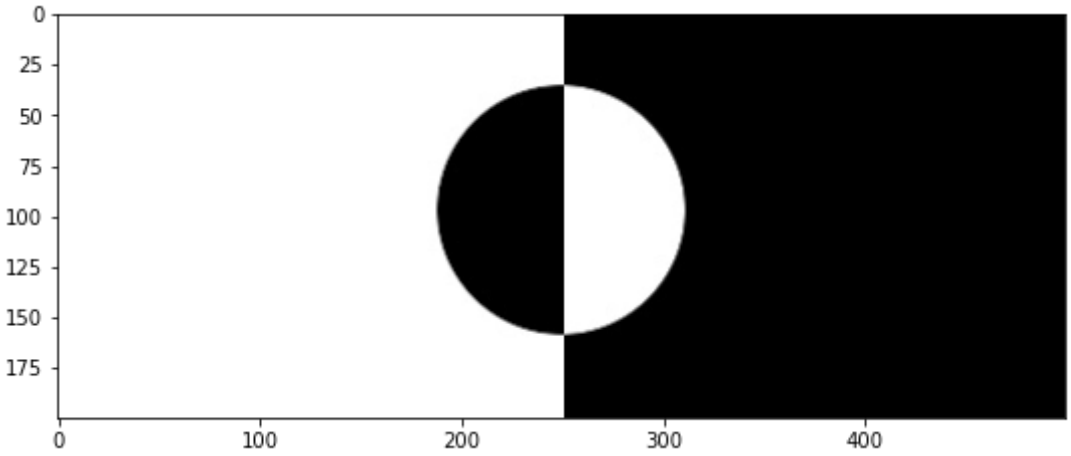
Out[53]: <matplotlib.image.AxesImage at 0x7fbe28fc0f70>



Bitwise XOR Operator

In [54]:
```
result = cv2.bitwise_xor(img_rec, img_cir, mask = None)
plt.imshow(result,cmap='gray')
```

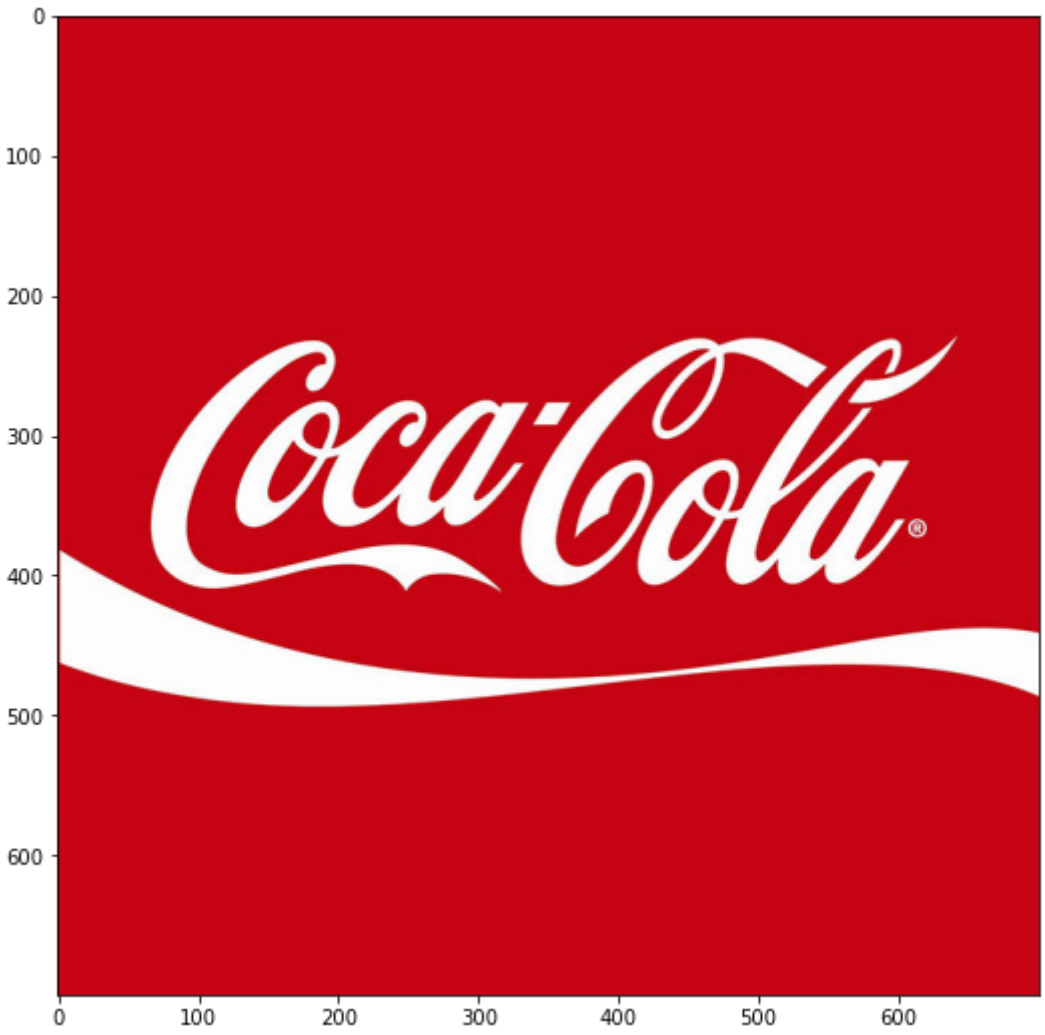
Out[54]: <matplotlib.image.AxesImage at 0x7fbe996a2370>



Read Foreground image

In [56]:
```
1 img_bgr = cv2.imread("imgs/coca-cola-logo.png")
2 img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
3 plt.imshow(img_rgb)
4 print(img_rgb.shape)
5 logo_w = img_rgb.shape[0]
6 logo_h = img_rgb.shape[1]
```

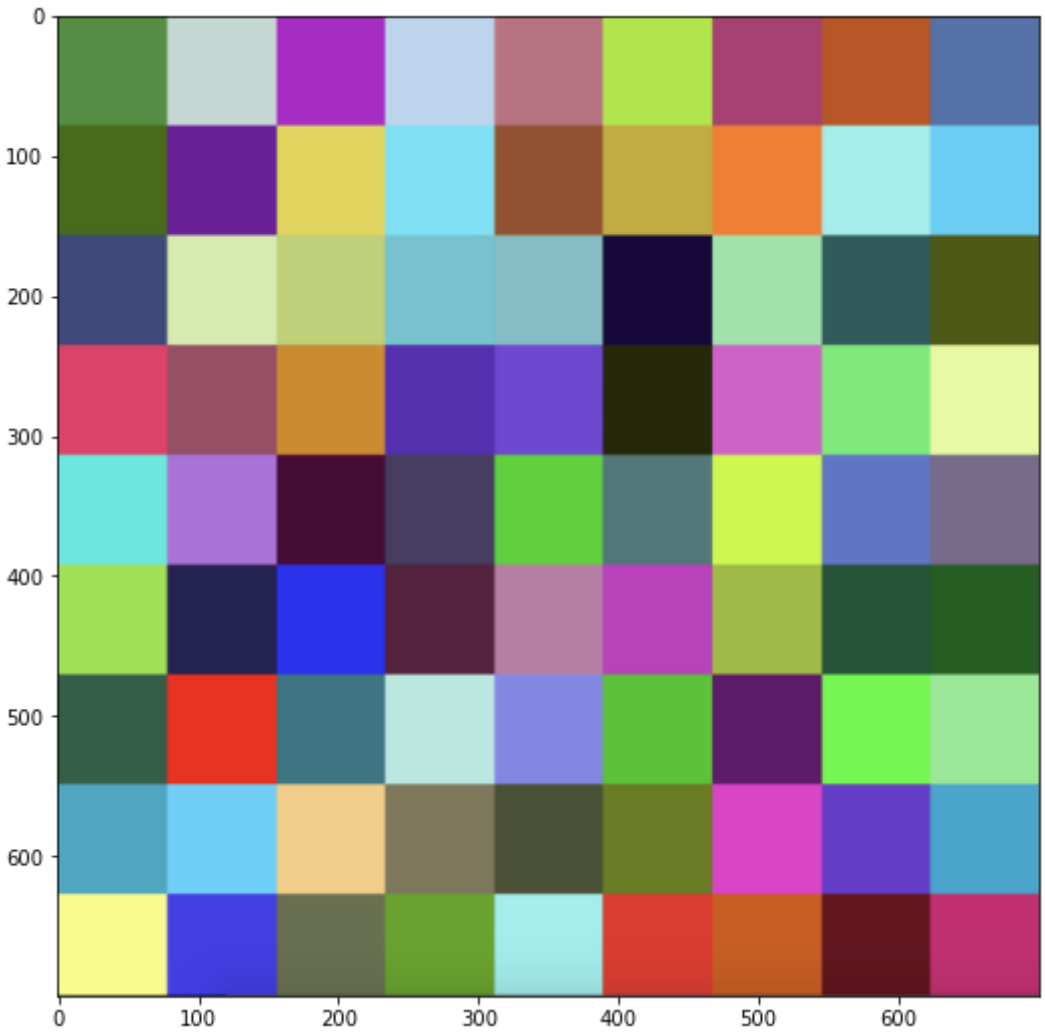
```
(700, 700, 3)
```



Read Background image

In [58]:
```
# Read in image of color cheackerboad background
img_background_bgr = cv2.imread("imgs/checkerboard_color.png")
img_background_rgb = cv2.cvtColor(img_background_bgr, cv2.COLOR_BGR2RGB

# Set desired width (logo_w) and maintain image aspect ratio
aspect_ratio = logo_w / img_background_rgb.shape[1]
dim = (logo_w, int(img_background_rgb.shape[0] * aspect_ratio))

# Resize background image to sae size as logo image
img_background_rgb = cv2.resize(img_background_rgb, dim, interpolation=

plt.imshow(img_background_rgb)
print(img_background_rgb.shape)
```

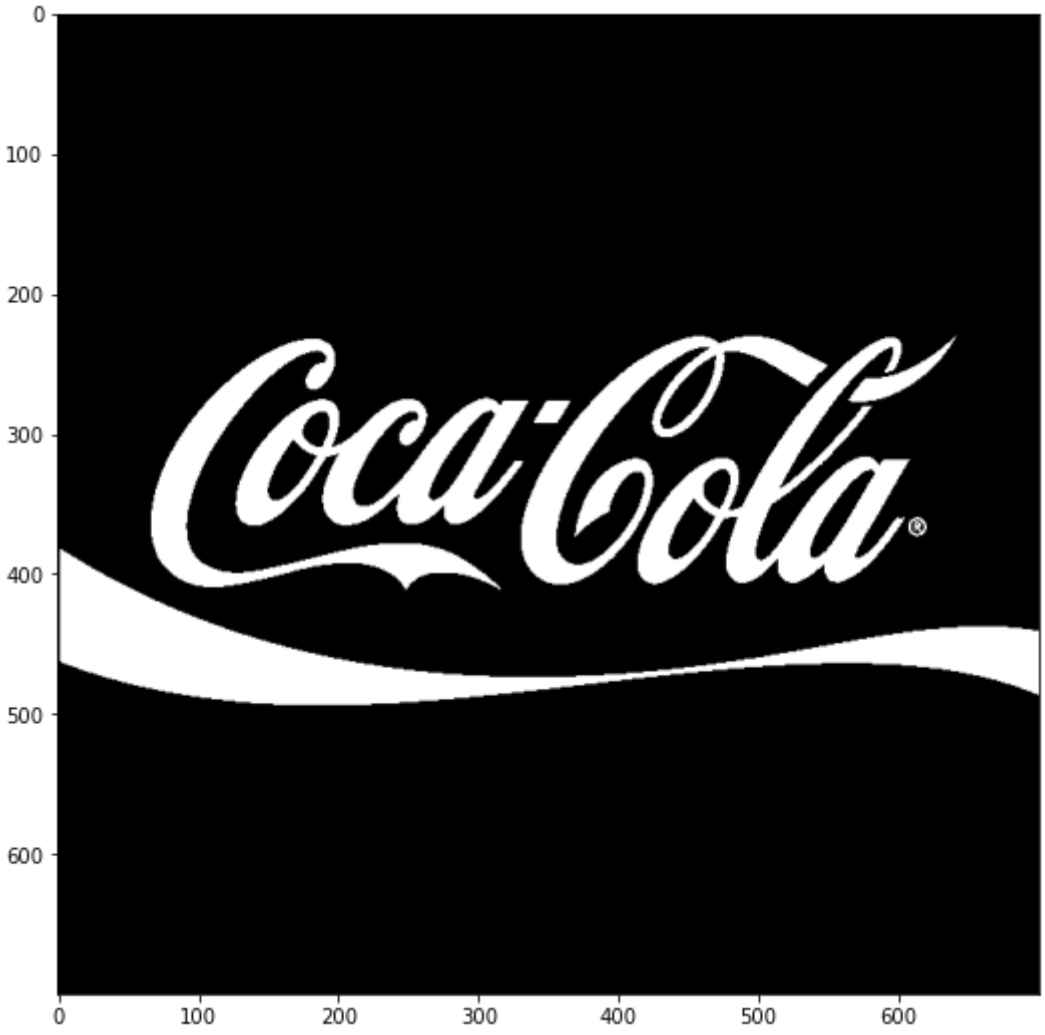
```
(700, 700, 3)
```



Create Mask for original Image

In [59]:
```
1  img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
2
3  # Apply global thresholding to creat a binary mask of the logo
4  retval, img_mask = cv2.threshold(img_gray,127,255,cv2.THRESH_BINARY)
5
6  plt.imshow(img_mask,cmap="gray")
7  print(img_mask.shape)
```

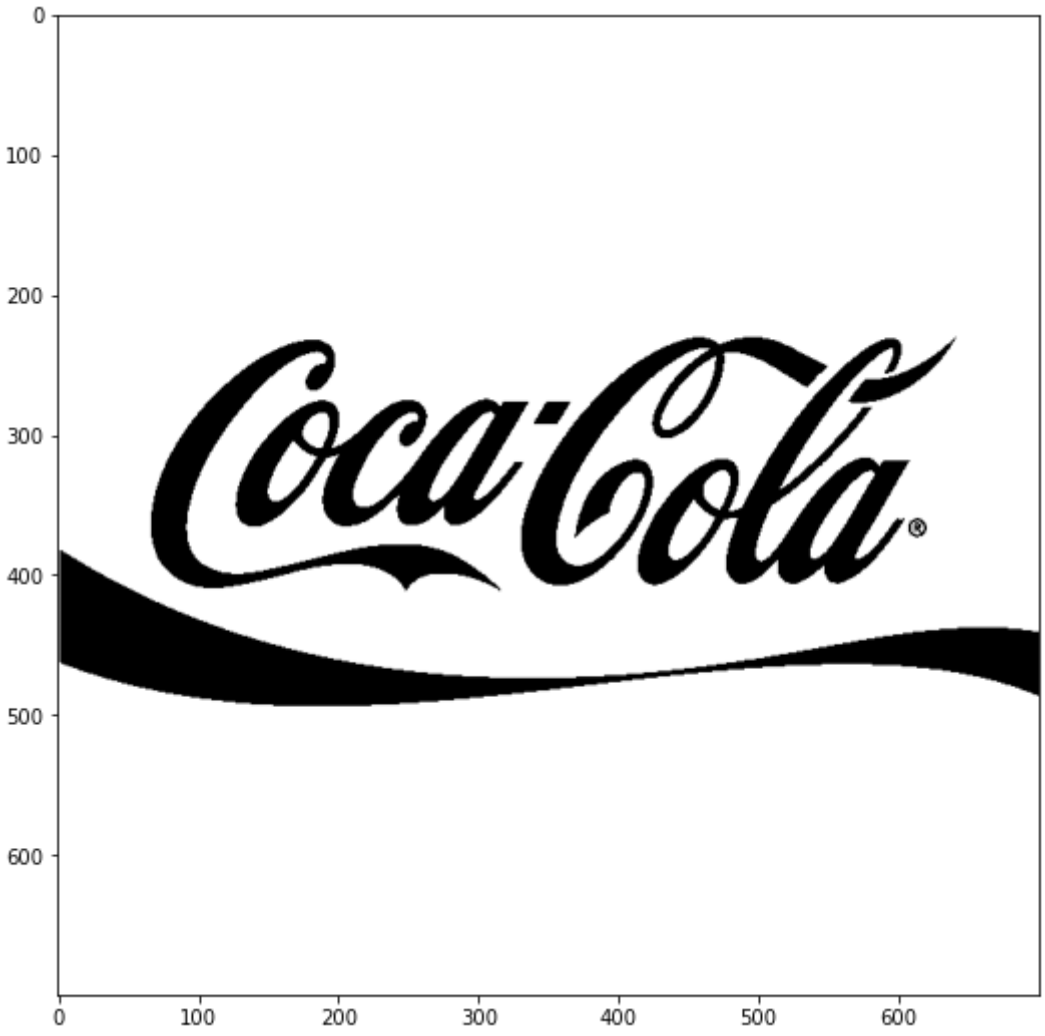
```
(700, 700)
```



Invert the Mask

In [60]:
```
# Create an inverse mask
img_mask_inv = cv2.bitwise_not(img_mask)
plt.imshow(img_mask_inv,cmap="gray")
```

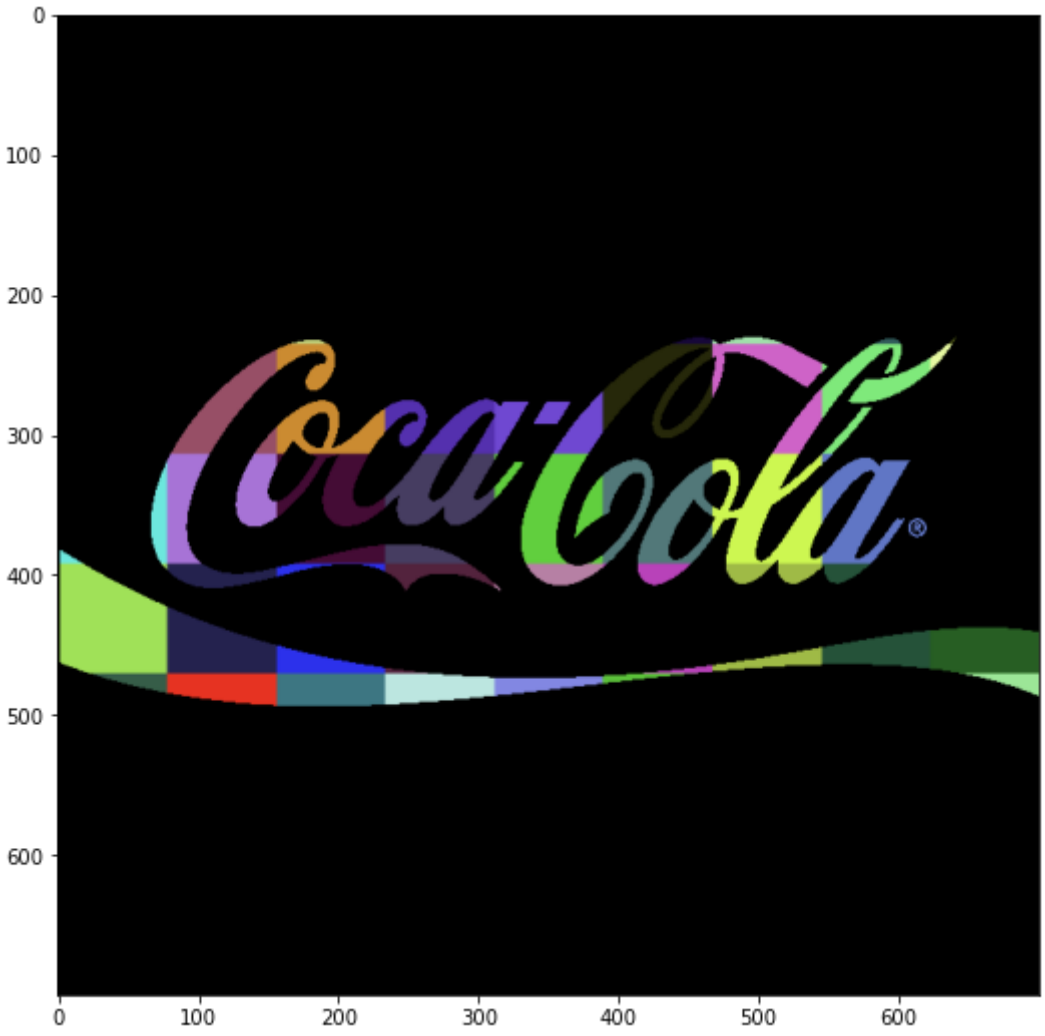
Out[60]: <matplotlib.image.AxesImage at 0x7fbe794f53d0>



Apply background on the Mask

In [61]:
```
# Create colorful background "behind" the logo lettering
img_background = cv2.bitwise_and(img_background_rgb, img_background_rgb
plt.imshow(img_background)
```

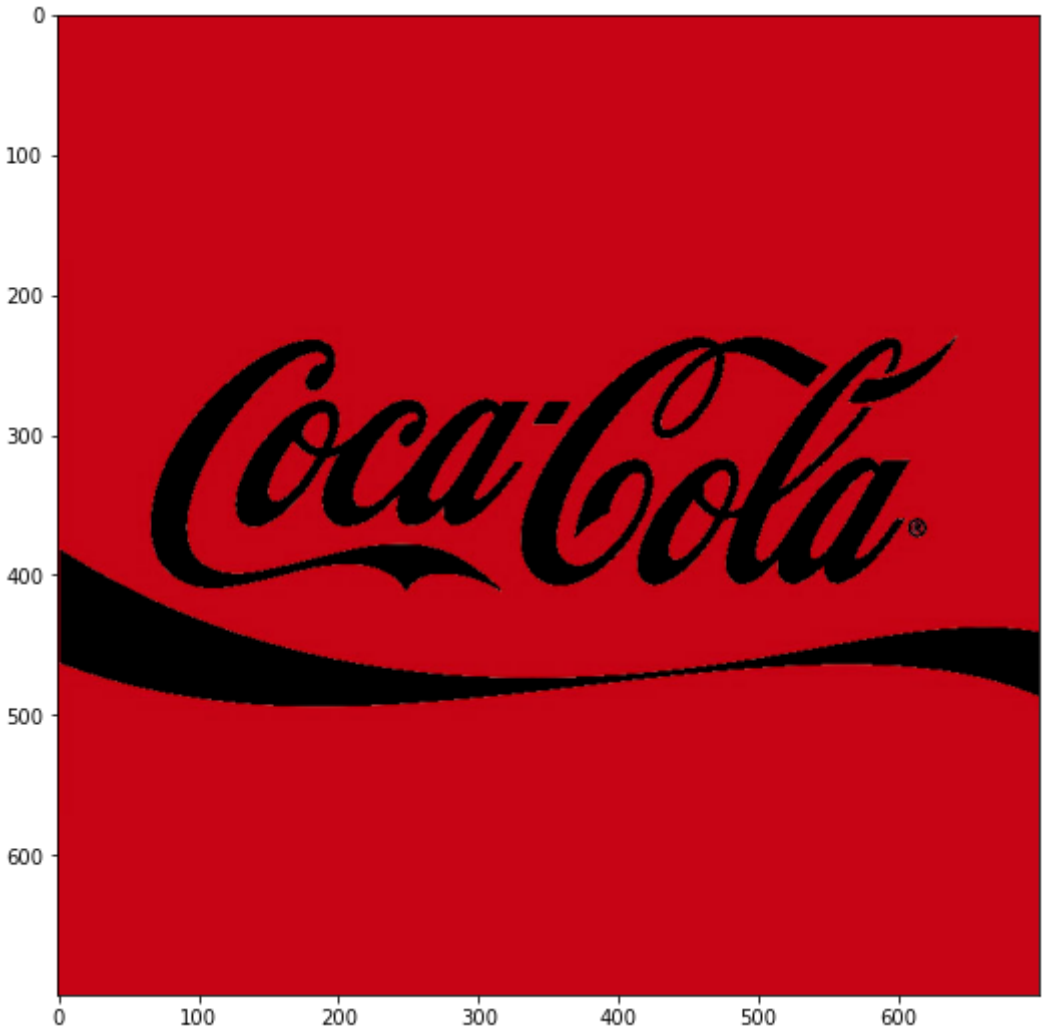
Out[61]: <matplotlib.image.AxesImage at 0x7fbe582806d0>



Isolate foreground from image

In [62]:
```
# Isolate foreground (red from original image) using the inverse mask
img_foreground = cv2.bitwise_and(img_rgb, img_rgb, mask=img_mask_inv)
plt.imshow(img_foreground)
```

Out[62]: <matplotlib.image.AxesImage at 0x7fbe69369100>



Result: Merge Foreground and Background

In [63]:
```
# Add the two previous results obtain the final result
result = cv2.add(img_background,img_foreground)
plt.imshow(result)
cv2.imwrite("logo_final.png", result[:,:,::-1])
```

Out[63]: True

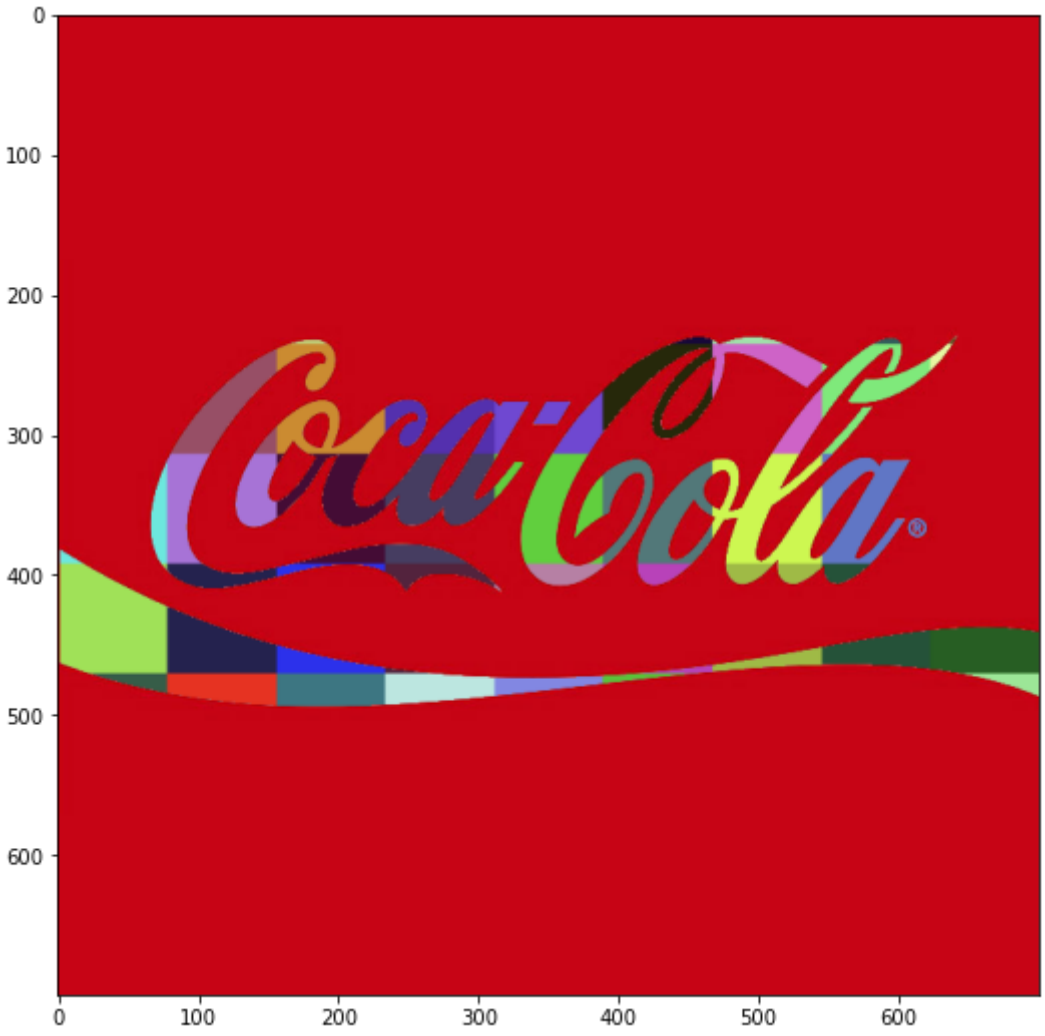
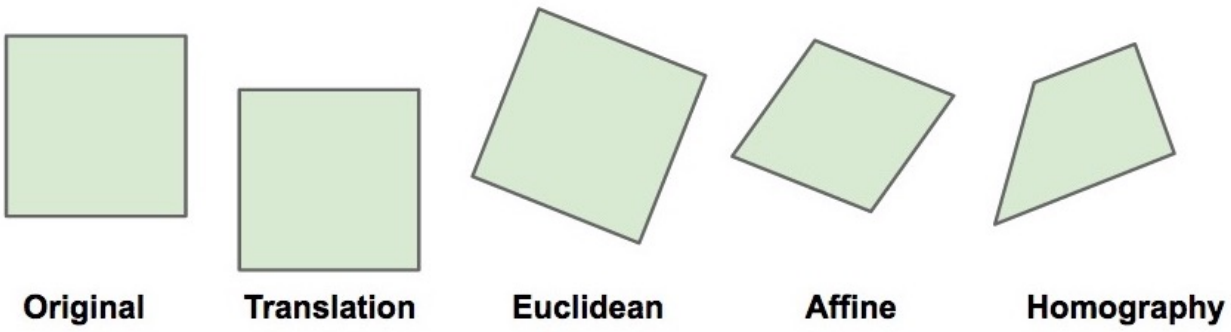


Image Alignment

Align an image to a template.

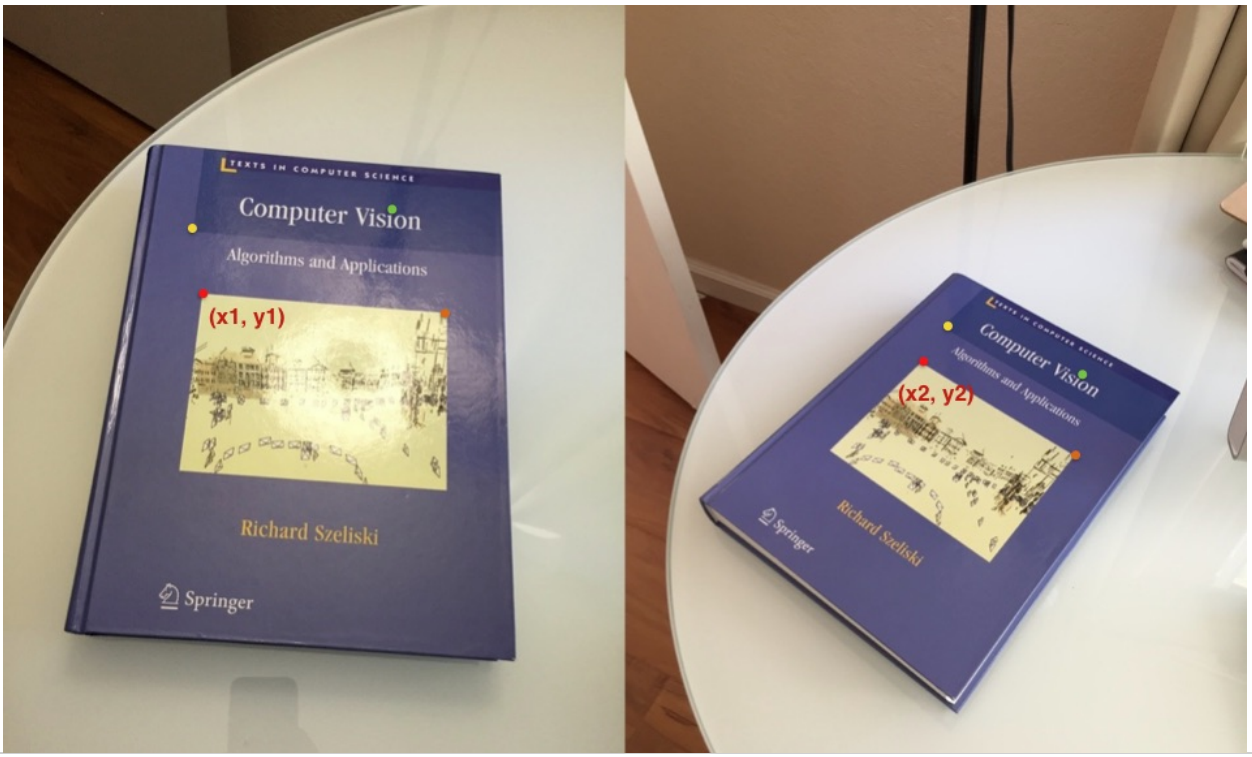
Theory

1. A **Homography** transforms a square to arbitrary quad.



Theory

2. Images of two planes are related by a **Homography**
3. We need **4 corresponding points** to estimate Homography



Step 1: Read Tempalate and Scanned Image

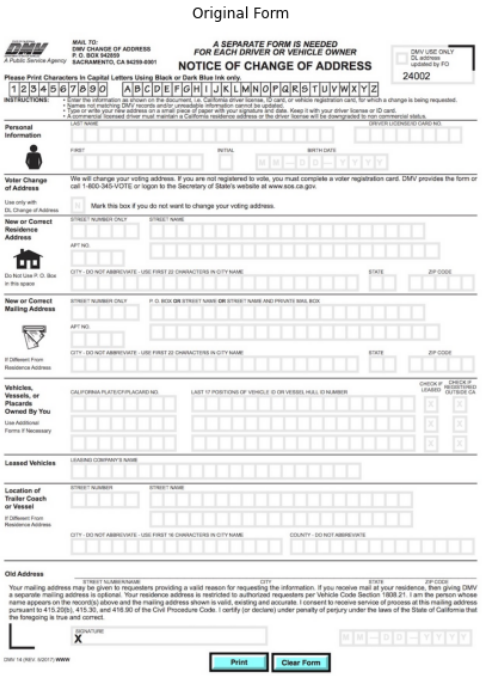
In [68]:

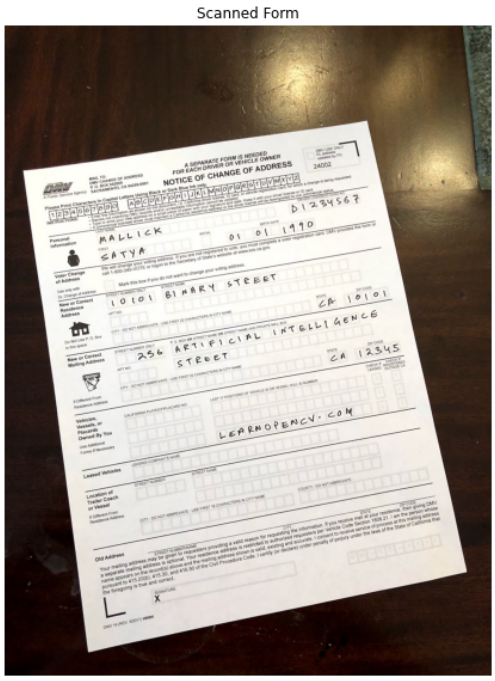
```
# Read reference image
refFilename = "imgs/form.jpg"
print("Reading reference image : ", refFilename)
im1 = cv2.imread(refFilename, cv2.IMREAD_COLOR)
im1 = cv2.cvtColor(im1, cv2.COLOR_BGR2RGB)

# Read image to be aligned
imFilename = "imgs/scanned-form.jpg"
print("Reading image to align : ", imFilename)
im2 = cv2.imread(imFilename, cv2.IMREAD_COLOR)
im2 = cv2.cvtColor(im2, cv2.COLOR_BGR2RGB)
```

```
Reading reference image :  imgs/form.jpg
Reading image to align :  imgs/scanned-form.jpg
```

In [69]:
```
# Display Images

plt.figure(figsize=[20,10]);
plt.subplot(121); plt.axis('off'); plt.imshow(im1); plt.title("Original
plt.subplot(122); plt.axis('off'); plt.imshow(im2); plt.title("Scanned
```

Out[69]:
```
Text(0.5, 1.0, 'Scanned Form')
```



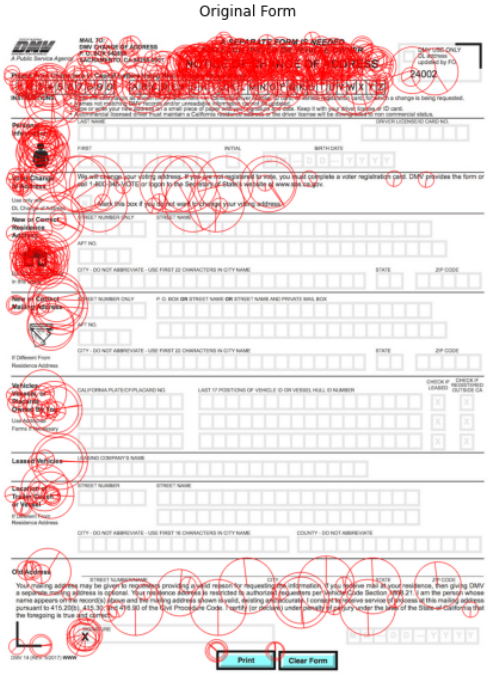


Step 2: Find keypoints in both Images

Think of keypoints as corner points that are stable under image transformations

In [70]:
```
# Convert images to grayscale
im1_gray = cv2.cvtColor(im1, cv2.COLOR_BGR2GRAY)
im2_gray = cv2.cvtColor(im2, cv2.COLOR_BGR2GRAY)


# Detect ORB features and compute descriptors.
MAX_NUM_FEATURES = 500
orb = cv2.ORB_create(MAX_NUM_FEATURES)
keypoints1, descriptors1 = orb.detectAndCompute(im1_gray, None)
keypoints2, descriptors2 = orb.detectAndCompute(im2_gray, None)

# Display
im1_display = cv2.drawKeypoints(im1, keypoints1, outImage=np.array([]),
im2_display = cv2.drawKeypoints(im2, keypoints2, outImage=np.array([]),

```

In [71]:
```
plt.figure(figsize=[20,10])
plt.subplot(121); plt.axis('off'); plt.imshow(im1_display); plt.title("
plt.subplot(122); plt.axis('off'); plt.imshow(im2_display); plt.title("
```

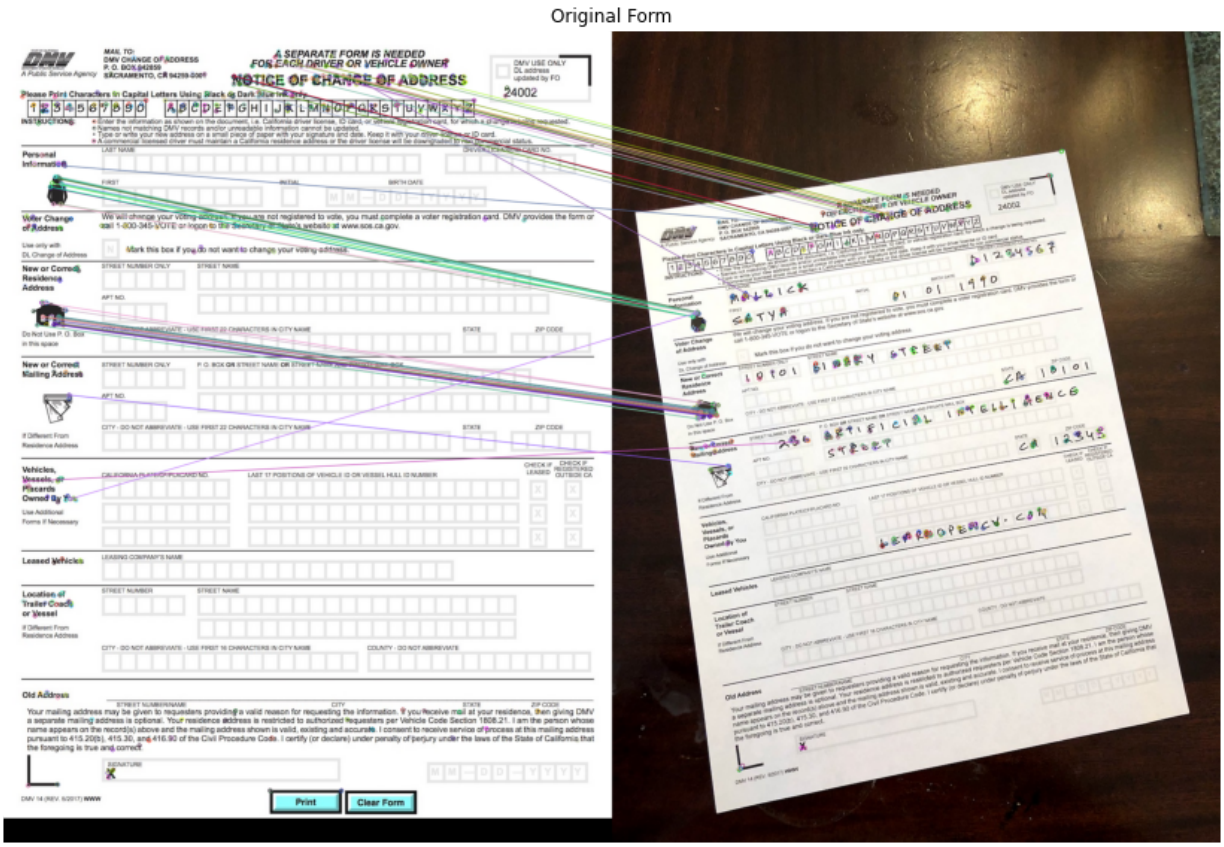


Step 3 : Match keypoints in the two image

In [72]:
```
# Match features.
matcher = cv2.DescriptorMatcher_create(cv2.DESCRIPTOR_MATCHER_BRUTEFORC
matches = matcher.match(descriptors1, descriptors2, None)

# Sort matches by score
matches.sort(key=lambda x: x.distance, reverse=False)

# Remove not so good matches
numGoodMatches = int(len(matches) * 0.1)
matches = matches[:numGoodMatches]
```

In [73]:

```
# Draw top matches
im_matches = cv2.drawMatches(im1, keypoints1, im2, keypoints2, matches,

plt.figure(figsize=[40,10])
plt.imshow(im_matches); plt.axis('off'); plt.title("Original Form");
```



Step 4: Find Homography

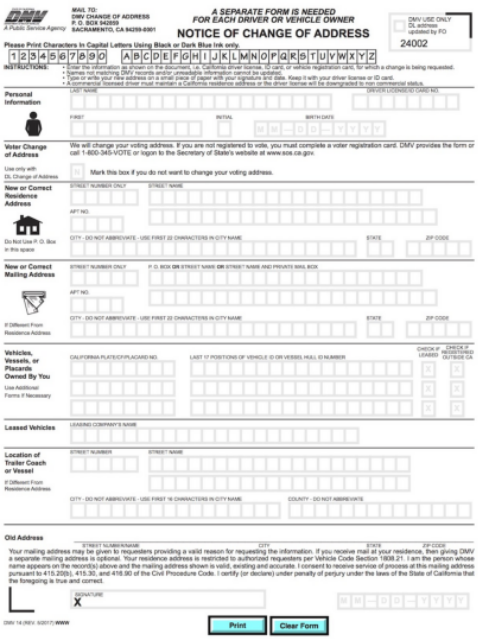
In [74]:

```
# Extract location of good matches
points1 = np.zeros((len(matches), 2), dtype=np.float32)
points2 = np.zeros((len(matches), 2), dtype=np.float32)

for i, match in enumerate(matches):
  points1[i, :] = keypoints1[match.queryIdx].pt
  points2[i, :] = keypoints2[match.trainIdx].pt

# Find homography
h, mask = cv2.findHomography(points2, points1, cv2.RANSAC)
```

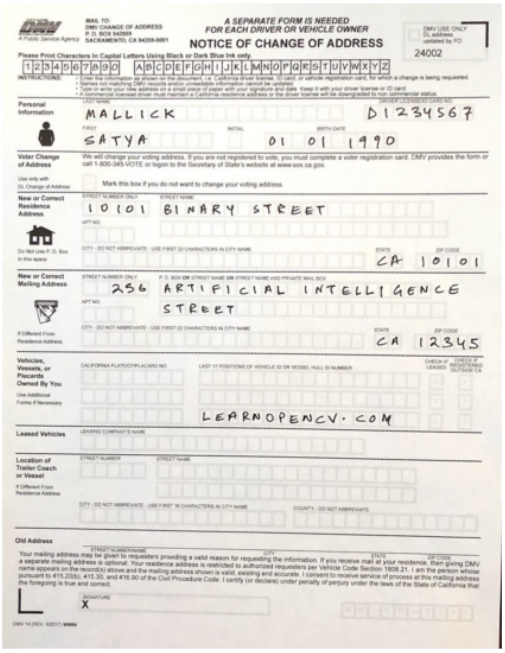
Step 5: Warp image

In [75]:
```
# Use homography to warp image
height, width, channels = im1.shape
im2_reg = cv2.warpPerspective(im2, h, (width, height))
# Display results
plt.figure(figsize=[20,10]);
plt.subplot(121); plt.imshow(im1); plt.axis('off'); plt.title("Original
plt.subplot(122); plt.imshow(im2_reg); plt.axis('off'); plt.title("Scan
```

Original Form



Scanned Form



Creating Panoramas using OpenCV

Steps for Creating Panoramas

1. Find keypoints in all images
2. Find pairwise correspondences
3. Estimate pairwise Homographies
4. Refine Homographies
5. Stitch with Blending

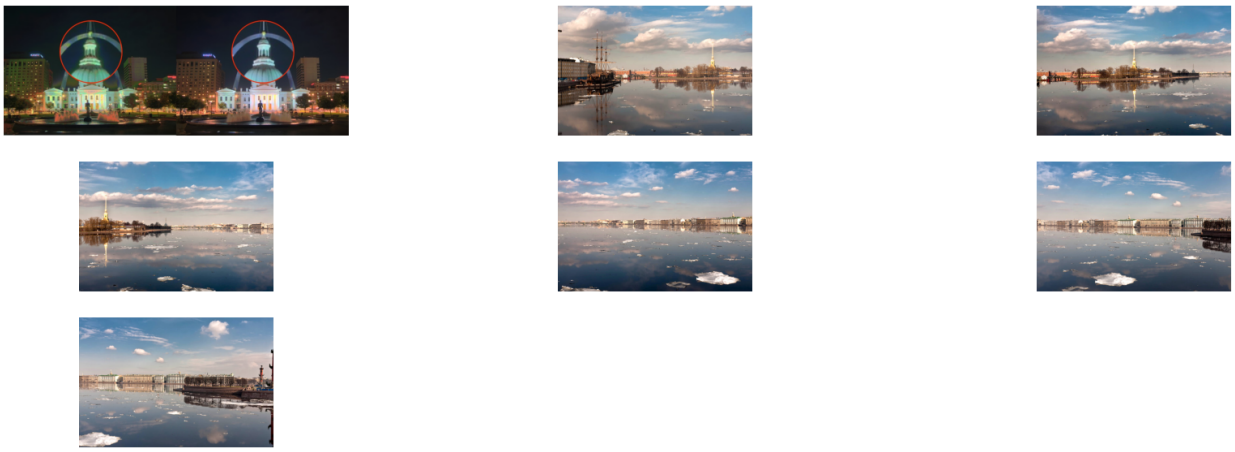
Steps for Creating Panoramas using OpenCV

1. Use the **sticher** class

In [80]:
```
import glob
import math
```

In [81]:
```
# Read Images

imagefiles = glob.glob("imgs/boat/*")
imagefiles.sort()

images = []
for filename in imagefiles:
  img = cv2.imread(filename)
  img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
  images.append(img)

num_images = len(images)
```

In [82]:
```
# Display Images
plt.figure(figsize=[30,10])
num_cols = 3
num_rows = math.ceil(num_images / num_cols)
for i in range(0, num_images):
  plt.subplot(num_rows, num_cols, i+1)
  plt.axis('off')
  plt.imshow(images[i])

```



In [83]:

```
# Stitch Images
stitcher = cv2.Stitcher_create()
status, result = stitcher.stitch(images)
if status == 0:
  plt.figure(figsize=[30,10])
  plt.imshow(result)
```

High Dynamic Range (HDR) Imaging

Basic Idea

1. The **dynamic range** of images is limited to 8-bits (0 - 255) per channel
2. Very bright pixels saturate to 255
3. Very dark pixels clip to 0

Step 1: Capture Multiple Exposures

In [84]:

```
def readImagesAndTimes():
  # List of file names
  filenames = ["imgs/img_0.033.jpg", "imgs/img_0.25.jpg", "imgs/img_2.5

  # List of exposure times
  times = np.array([ 1/30.0, 0.25, 2.5, 15.0 ], dtype=np.float32)

  # Read images
  images = []
  for filename in filenames:
    im = cv2.imread(filename)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
    images.append(im)

  return images, times
```

Step 2: Align Images

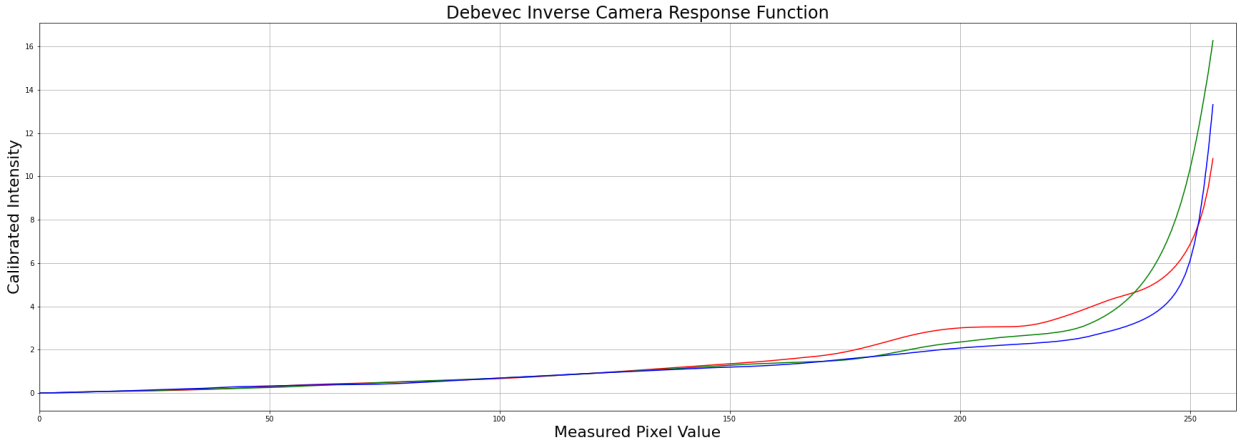
In [85]:

```
# Read images and exposure times
images, times = readImagesAndTimes()

# Align Images
alignMTB = cv2.createAlignMTB()
alignMTB.process(images, images)
```

Step 3: Estimate Camera Response Function

In [86]:

```
# Find Camera Response Function (CRF)
calibrateDebevec = cv2.createCalibrateDebevec()
responseDebevec = calibrateDebevec.process(images, times)

# Plot CRF
x = np.arange(256, dtype=np.uint8)
y = np.squeeze(responseDebevec)

ax = plt.figure(figsize=(30,10))
plt.title("Debevec Inverse Camera Response Function", fontsize=24)
plt.xlabel("Measured Pixel Value", fontsize=22)
plt.ylabel("Calibrated Intensity", fontsize=22)
plt.xlim([0,260])
plt.grid()
plt.plot(x, y[:,0],'r'  , x, y[:,1],'g', x, y[:,2],'b');
```



Step 4: Merge Exposure into an HDR Image

In [87]:

```
# Merge images into an HDR linear image
mergeDebevec = cv2.createMergeDebevec()
hdrDebevec = mergeDebevec.process(images, times, responseDebevec)
```

Step 5: Tonemapping

Many Tonemapping algorithms are available in OpenCV. We chose Durand as it has more controls.

In [91]:

```
# Tonemap using Drago's method to obtain 24-bit color image
tonemapDrago = cv2.createTonemapDrago(1.0, 0.7)
ldrDrago = tonemapDrago.process(hdrDebevec)
ldrDrago = 3 * ldrDrago
plt.figure(figsize=(20,10)); plt.imshow(np.clip(ldrDrago,0,1)); plt.axi
cv2.imwrite("imgs/ldr-Drago.jpg", ldrDrago * 255)
print("imgs/saved ldr-Drago.jpg")

```

```
imgs/saved ldr-Drago.jpg
```

In [92]:

```
# Tonemap using Reinhard's method to obtain 24-bit color image
print("Tonemaping using Reinhard's method ... ")
tonemapReinhard = cv2.createTonemapReinhard(1.5, 0,0,0)
ldrReinhard = tonemapReinhard.process(hdrDebevec)
plt.figure(figsize=(20,10)); plt.imshow(np.clip(ldrReinhard,0,1)); plt.
cv2.imwrite("ldr-Reinhard.jpg", ldrReinhard * 255)
print("imgs/saved ldr-Reinhard.jpg")
```

```
Tonemaping using Reinhard's method ...
imgs/saved ldr-Reinhard.jpg
```

In [93]:

```
# Tonemap using Mantiuk's method to obtain 24-bit color image
print("Tonemaping using Mantiuk's method ... ")
tonemapMantiuk = cv2.createTonemapMantiuk(2.2,0.85, 1.2)
ldrMantiuk = tonemapMantiuk.process(hdrDebevec)
ldrMantiuk = 3 * ldrMantiuk
plt.figure(figsize=(20,10)); plt.imshow(np.clip(ldrMantiuk,0,1)); plt.a
cv2.imwrite("ldr-Mantiuk.jpg", ldrMantiuk * 255)
print("imgs/saved ldr-Mantiuk.jpg")
```

```
Tonemaping using Mantiuk's method ...
imgs/saved ldr-Mantiuk.jpg
```