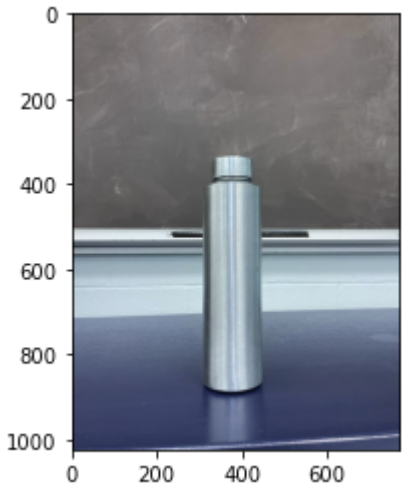
Importing some important module to complete my first software assignment.

In [1]:
```
import cv2
import numpy as np
import os
import glob
from matplotlib import pyplot as plt
import plotly.express as plx
import imutils
```

Import images of interest for completing first task.

In [22]:
```
image_directory= "C:/Users/kaisar/Software Assignment 1/water_pot/"
file_list=glob.glob(image_directory)

#now let us load each file at a time

my_list=[] #Empty list to store images from the folder

path= "C:/Users/kaisar/Software Assignment 1/water_pot/*.jpeg"
for file in glob.glob(path):
    print(file)
    image=cv2.imread(file)
    my_list.append(image)
plt.imshow(my_list[4])

print("Original Image Size:",my_list[2].shape)
print("Number of images: ",len(my_list))
```

```
C:/Users/kaisar/Software Assignment 1/water_pot\pot1.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot10.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot11.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot12.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot13.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot14.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot15.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot16.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot17.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot18.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot19.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot2.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot20.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot3.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot4.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot5.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot6.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot7.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot8.jpeg
C:/Users/kaisar/Software Assignment 1/water_pot\pot9.jpeg
Original Image Size: (1024, 768, 3)
Number of images:  20
```



Double Exposure, SNR Calculation & Corresponding Graph, Noise Variance Calculation & Corresponding Graph:

In this portion of the assignment, we have an instruction to take multiple images of the same object under the same conditions and add them up to create a new image.

Images used: 20 images of my water pot were taken continuously within a fixed distance.

Steps that I have followed for this section:

Steps1 We have an instruction to perform actions on images are color images

Steps2 Images were added one after the other to get 20 images resulting from a simple matrix addition that is the first image is a summation of 20 images.

Steps3 Signal to Noise Ratio (SNR) was calculated after each image addition. SNR was defined as the ratio of image average over its standard deviation.

Steps4 Noise variance was calculated after each addition. To get this value, I set the first image as a reference and subtracted the image from each addition to get the noise matrix which was used to get the variance.

Outcomes from the Graphs:

By plotting SNR obtained after each addition, it first increases after adding couple of images and then reduces as more images were added.

Noise variance decreased as more images were added.

Observation:

Higher SNR means better and less noise signal.

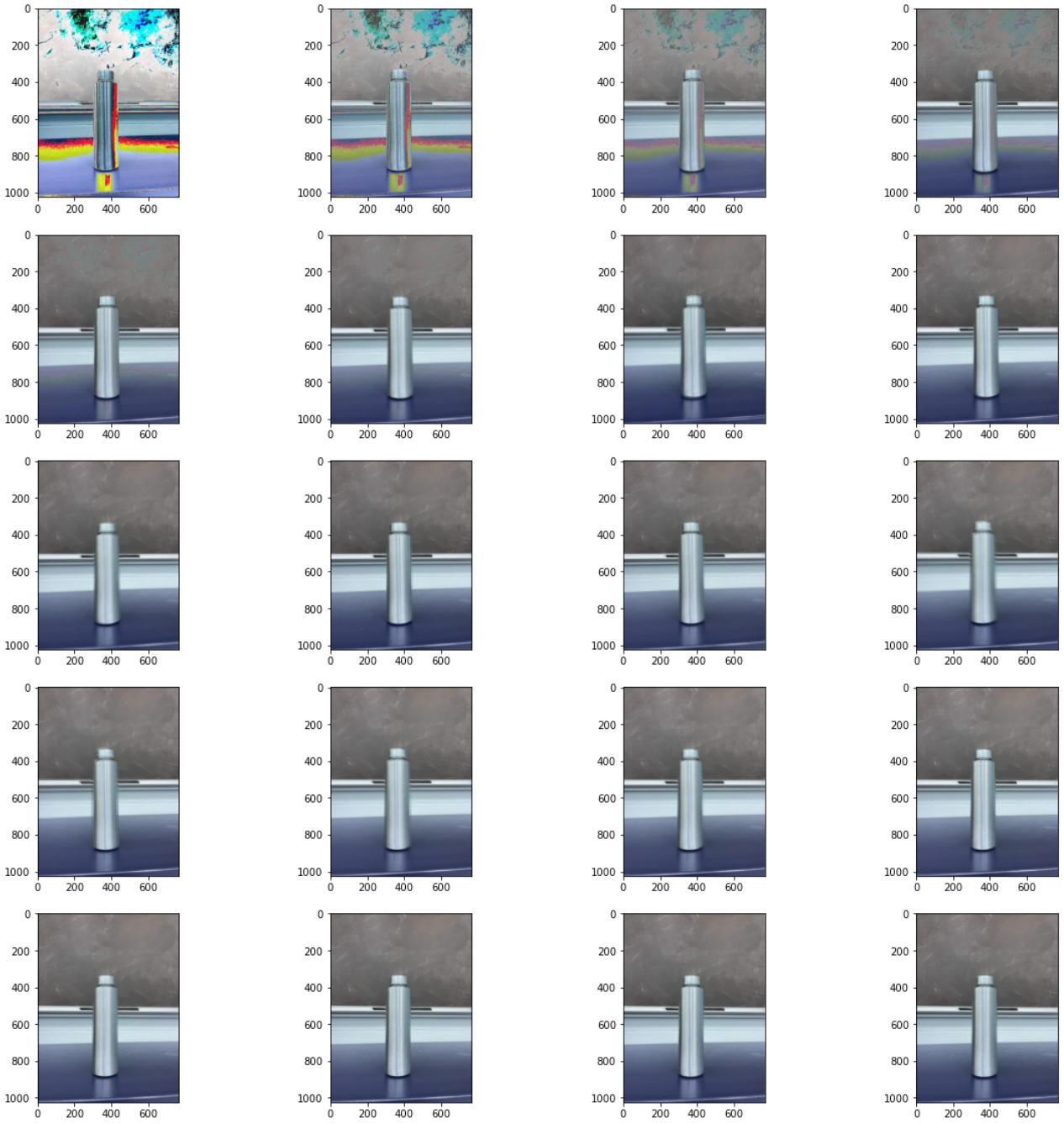
My results show that added images added more noise to our image since there was an overall decrease in SNR.

However, noise variance decreased which might imply that the resulting image became smoother.

In [23]:

```
image_addition=[my_list[0]]
image_addition[-1]

noise_variance=[]
for i in range(len(my_list)):
    p=image_addition[-1]+my_list[i]
    p=((p-p.min())/(p.max()-p.min()))*255
    p=p.astype(dtype=int)
    image_noise=my_list[0]-p
    #plt.imshow(image_noise.astype('uint8'))
    image_addition.append(p)
    noise_variance.append(np.var(image_noise))
print('Noise Variane of 20 Images:', noise_variance)

fig=plt.figure(figsize=(20,20))
for i in range(1,len(image_addition)):
    fig.add_subplot(5,4,i)
    plt.imshow(image_addition[i])
plt.show()
```

```
Noise Variane of 20 Images: [7907.158327575285, 2110.694430502199, 708.171629512473, 42
5.0496770259407, 313.816539010451, 415.6284239306567, 326.5143853067418, 286.72436445631
934, 242.61976400582566, 191.1067839727246, 156.74517672507207, 296.3468481150377, 152.9
1715306690216, 204.25465005456556, 216.557376132838, 216.2961711619409, 236.928373754573
53, 270.8294224002667, 270.32675256519735, 304.63201428520307]
```

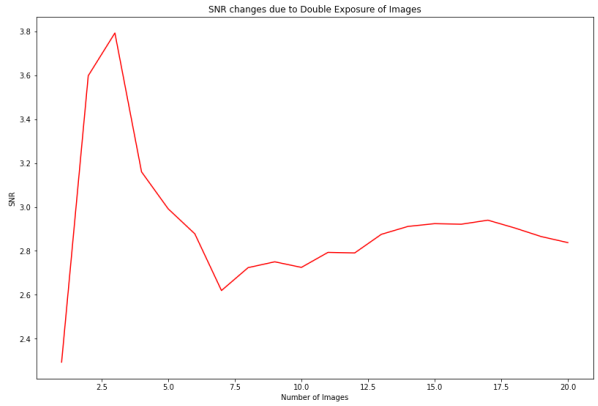


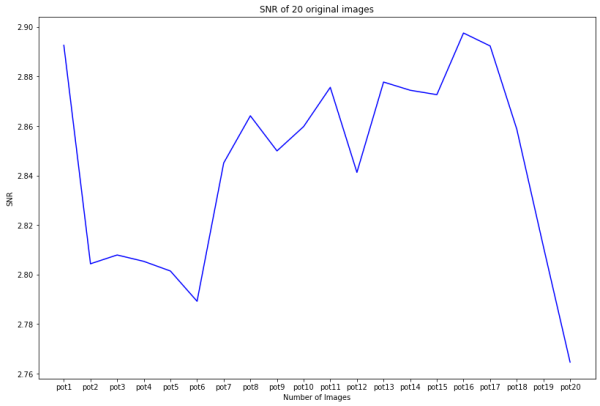
In [24]:

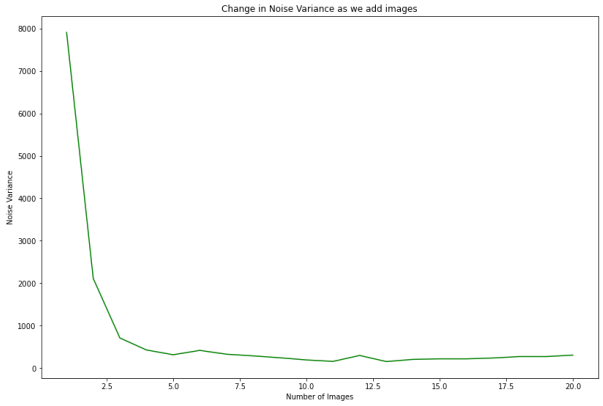
```
def snr_calculation(image):
    mean= np.mean(image)
    std=np.std(image)
    snr=mean/std
    return snr

snr_addition_images=[]
snr_original_images=[]

## SNR calculation from double exposure

for i in image_addition:
    image_snr=snr_calculation(i)
    snr_addition_images.append(image_snr)

##SNR calculation from original Images
```

```
for k in my_list:
    snr_org=snr_calculation(k)
    snr_original_images.append(snr_org)

#remove SNR of original image
del snr_addition_images[0]

x=["pot1","pot2","pot3","pot4","pot5","pot6","pot7","pot8","pot9","pot10","pot11","pot1
x1=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

fig = plt.figure(figsize=(30,20))
fig.add_subplot(221)
plt.plot(x1,snr_addition_images,'r')
plt.title('SNR changes due to Double Exposure of Images')
plt.xlabel('Number of Images')
plt.ylabel('SNR')

fig.add_subplot(222)
plt.plot(x, snr_original_images,'b')
plt.title('SNR of 20 original images')
plt.xlabel('Number of Images')
plt.ylabel('SNR')

fig.add_subplot(223)
plt.plot(x1, noise_variance,"g")
plt.title('Change in Noise Variance as we add images')
plt.xlabel('Number of Images')
plt.ylabel('Noise Variance')
```

Out[24]: Text(0, 0.5, 'Noise Variance')





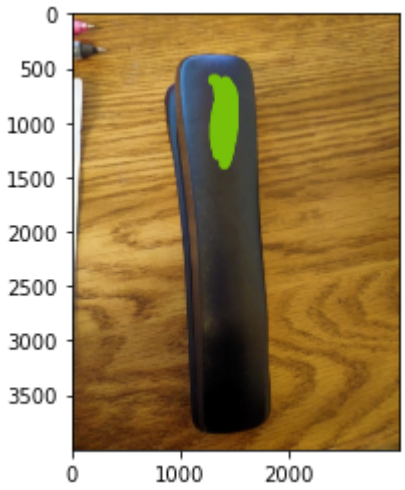


Change Detection

I have two images(one is Stapler that the brand name is covered by color and another is the original image of Stapler) and would like to make it obvious where the differences are. I want to add color to the two images such that a user can clearly spot all the differences.

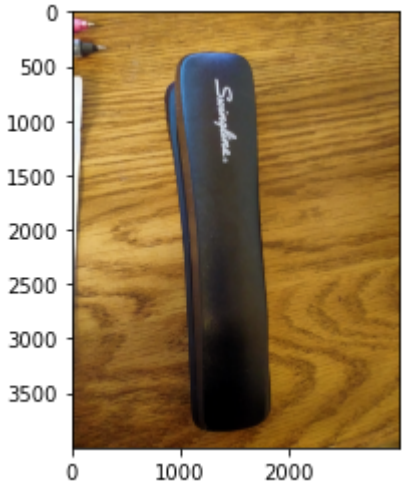
In [40]:
```
# Load images
image1 = cv2.imread("Stapler3.jpg")
image_rgb1=cv2.cvtColor(image1,cv2.COLOR_BGR2RGB)
plt.imshow(image_rgb1)
```

Out[40]:
```
<matplotlib.image.AxesImage at 0x24401620cd0>
```



In [41]:
```
image2 = cv2.imread("Stapler2.jpg")
image_rgb2=cv2.cvtColor(image2,cv2.COLOR_BGR2RGB)
plt.imshow(image_rgb2)
```

Out[41]:
```
<matplotlib.image.AxesImage at 0x2440168c850>
```



In [42]:
```
# compute difference
difference = cv2.subtract(image_rgb1, image_rgb2)
```

I threshold my difference image using both cv2.THRESH_BINARY_INV and cv2.THRESH_OTSU — both of these settings are applied at the same time using the vertical bar 'or' symbol,' |' .

How a Simple thresholding work:

If pixel value is greater than a threshold value, it is assigned one value (may be white), else it is assigned another value (may be black).

The function used is cv2.threshold.

First argument is the source image, which should be a grayscale image.

Second argument is the threshold value which is used to classify the pixel values.

Third argument is the maxium Value which represents the value to be given if pixel value is more than (sometimes less than) the threshold value.

OpenCV provides different styles of thresholding:

The below two methods are two styles of thresholding:

1. cv2.THRESH_BINARY_INV

2. cv2.THRESH_OTSU

In [43]:
```
#Threshold difference images
Stapler_Gray = cv2.cvtColor(difference, cv2.COLOR_RGB2GRAY)
thresh = cv2.threshold(Stapler_Gray, 0, 255,cv2.THRESH_BINARY_INV |cv2.THRESH_OTSU)
```

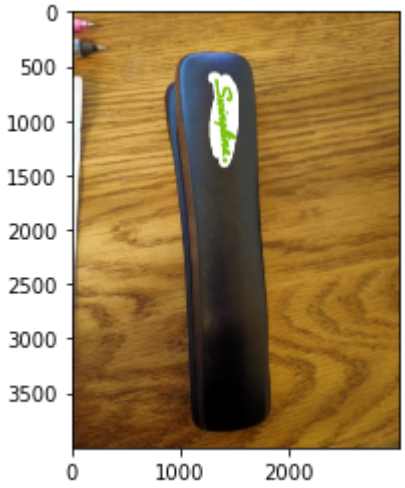
In Python != is defined as not equal to operator. It returns True if operands on either side are not equal to each other, and returns False if they are equal. Why I use != operator?

Let say in the image1 the point image1 [x,y] = [10,10,200]. In the different matrix, the different[x,y] = [0,0,255]. After "+" computing, the new value is [10,10,455], this will not work because of the R value is over 255.

In [44]:
```
# add the white mask to the images to make the differences obvious
difference[mask != 255] = [255,255,255]
image_rgb1[mask != 255] = [255,255,255]
image_rgb2[mask != 255] = [255,255,255]
```

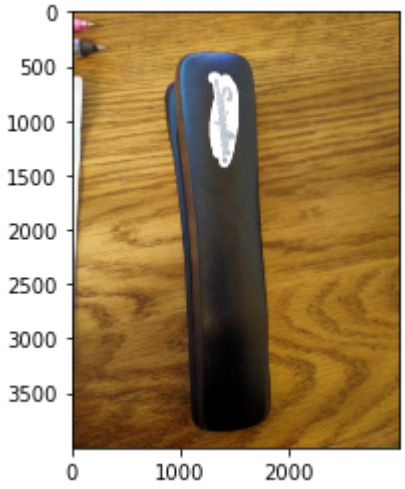
In [45]:
```
#difference Over Image1
plt.imshow(image_rgb1)
```

Out[45]:
```
<matplotlib.image.AxesImage at 0x244016fd160>
```



In [46]:
```
#difference Over Image2
plt.imshow(image_rgb2)
```

Out[46]: <matplotlib.image.AxesImage at 0x24401758ca0>



In [48]:
```
#Shows the difference between two mask
plt.imshow(difference)
```

Out[48]: <matplotlib.image.AxesImage at 0x24401832250>

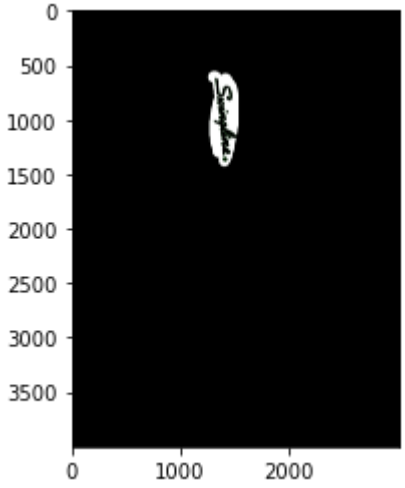


Image Multiplication

In this part, I took seven images of the same table lamp and selected the four best images to be used from image multiplication.

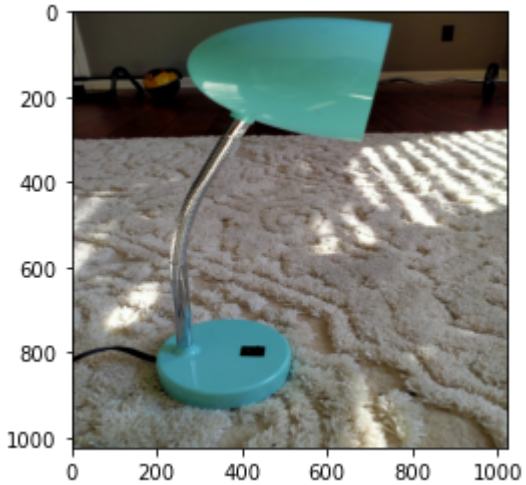
I created four different rectangular masks and used them to extract a given area in the image.

One mask was multiplied by one image to get one portion of the image, other with different images to cut off the left, middle, and right sides of three remaining images.

Each multiplication produced one matrix, and all matrices were added to obtain a brand-new image from different images.

In [25]:
```
#We first load our image of interest.
#Table_lamp_image_directory="C:/Users/kaisar/Software Assignment 1/Table Lamp"
table_lamp=[]
lamp_image_path="C:/Users/kaisar/Software Assignment 1/Table Lamp/*.jpg"
for lamp in glob.glob(lamp_image_path):
    print(lamp)
    l=cv2.imread(lamp)
    l.astype(dtype=int)
    l=cv2.cvtColor(l,cv2.COLOR_BGR2RGB)
    l=cv2.resize(l,(1024,1024))
    table_lamp.append(l)
plt.imshow(table_lamp[6])
print("Table Lamp Image Size:",table_lamp[2].shape)
print("Number of Table Lamp Images: ",len(table_lamp))
```

```
C:/Users/kaisar/Software Assignment 1/Table Lamp\IMG20220205102821.jpg
C:/Users/kaisar/Software Assignment 1/Table Lamp\IMG20220205102824.jpg
C:/Users/kaisar/Software Assignment 1/Table Lamp\IMG20220205102825.jpg
C:/Users/kaisar/Software Assignment 1/Table Lamp\IMG20220205102826.jpg
C:/Users/kaisar/Software Assignment 1/Table Lamp\IMG20220205102828.jpg
C:/Users/kaisar/Software Assignment 1/Table Lamp\IMG20220205102830.jpg
C:/Users/kaisar/Software Assignment 1/Table Lamp\IMG20220205102832.jpg
Table Lamp Image Size: (1024, 1024, 3)
Number of Table Lamp Images:  7
```



Syntax: cv2.rectangle(image, start_point, end_point, color, thickness)

Parameters:

image: It is the image on which rectangle is to be drawn.

start_point: It is the starting coordinates of rectangle. The coordinates are represented as tuples of two values i.e. (X coordinate value, Y coordinate value).

end_point: It is the ending coordinates of rectangle. The coordinates are represented as tuples of two values i.e. (X coordinate value, Y coordinate value).

color: It is the color of border line of rectangle to be drawn. For BGR, we pass a tuple. eg: (255, 0, 0) for blue color.

thickness: It is the thickness of the rectangle border line. Thickness of -1 will fill the rectangle shape by the specified color.

In [26]:

```
lamp_head_mask= np.zeros_like(table_lamp[6])
cv2.rectangle(lamp_head_mask, (0,0), (1024, 420),(255,255,255), thickness=-1)

# Creatng different masks
lamp_left_mask=np.zeros_like(table_lamp[5])
cv2.rectangle(lamp_left_mask, (0,420), (200,1024),(255,255,255), thickness=-1)
lamp_right_mask=np.zeros_like(table_lamp[4])
cv2.rectangle(lamp_right_mask,(800,420),(1024,1024),(255,255,255), thickness=-1)
middle_mask=np.zeros_like(table_lamp[3])
cv2.rectangle(middle_mask, (200,420), (800,1024),(255,255,255),thickness=-1)

# Multiplication of masks and different images
head=(table_lamp[6]/255)*(lamp_head_mask/255)
right=(table_lamp[4]/255)*(lamp_right_mask/255)
left=(lamp_left_mask/255)*(table_lamp[2]/255)
body= (table_lamp[3]/255)*(middle_mask/255)

## Addition to get original image
Whole_lamp= head+right+left+body
masked=[left,body,head,right,Whole_lamp]
f = plt.figure(figsize=(20,15))
f.add_subplot(231)
plt.imshow(left)
f.add_subplot(232)
plt.imshow(body )
f.add_subplot(233)
plt.imshow(head)
f.add_subplot(234)
plt.imshow(right)
f.add_subplot(235)
#plt.imshow(Whole_lamp)
plt.imshow((Whole_lamp * 255).astype(np.uint8))
```

Out[26]:

```
<matplotlib.image.AxesImage at 0x2ab822c3580>
```

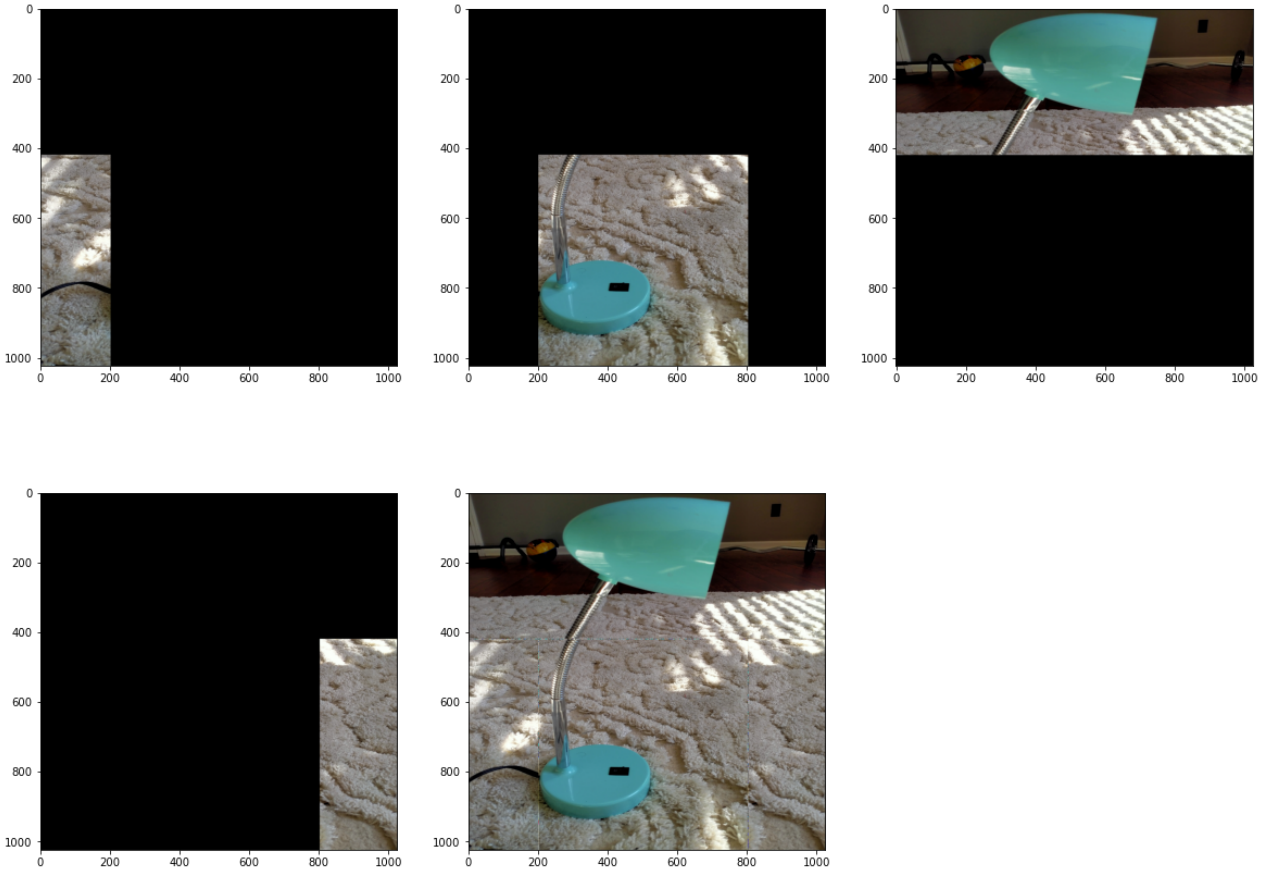


Image Registration

I have tried to understand, what is the origin of ORB by using the below tasks and how can I use it for Image Registration

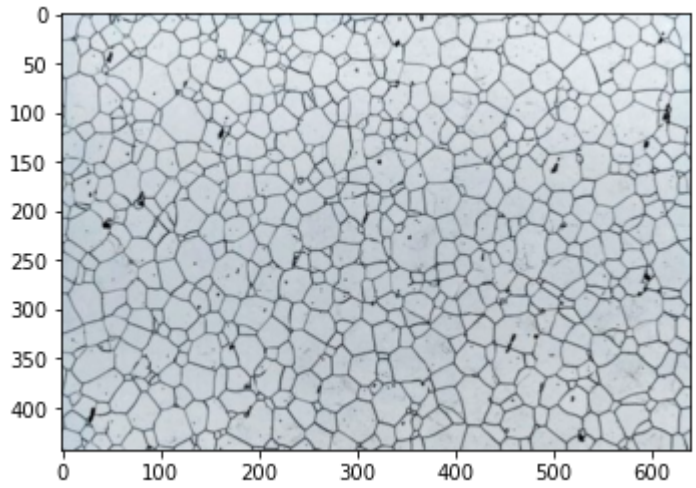
Key points, detectors and descriptors-If we are collecting a whole bunch of images and we are trying to do for example on a Focused Ion Beam Scanning Electron Microscopy (FIB-SEM) (It can automatically generate 3D images with superior z-axis resolution, yielding data that needs minimal image registration and related post-processing). If we are collecting images and want to register these images it's nice if we can get some key points and work on that we can actually bring them back and we can register them in a nice way.

So the reason I am showing this example is once we identify these key points we can use those key points to transform the image back to the reference image so that the registration would be easily guided by these key points.

Harris Key Point Detectors

In [47]:
```
image=cv2.imread("grains1.jpg")
plt.imshow(image)
```

Out[47]:
```
<matplotlib.image.AxesImage at 0x1c0aeb0b400>
```



In [49]:
```
gray=cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
#Harris works on gray level images that are float32 so
# i will be converting this image into floor 32.
gray=np.float32(gray)
```

From the documentation, the next one would be the block size which is nothing but the size of the neighborhood considered. Here I used 2 that they used in the documentation and then the next one is the aperture parameter for the Sobel filter that it uses as part of the Harris, so I used it to 3 as it suggests and also the last parameter they call it the K Harris parameter I'm going to leave it 0.04.

In [56]:
```
harris=cv2.cornerHarris(gray,2,3,0.04)
print("grains Image Size:",harris.shape)
```

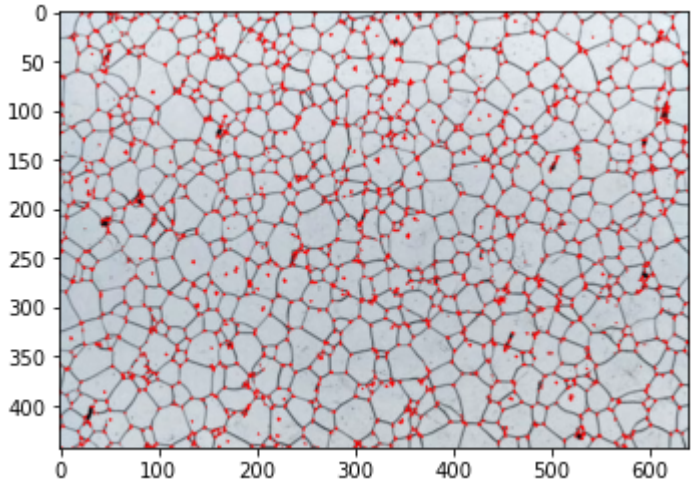
```
grains Image Size: (443, 640)
```

when I run this it returns in here is a whole bunch of points and I do not want all of these points I only want the points that are above a certain number so I'm going to take my original image array, I just want to take only the pixels where the highest value is greater than a certain number so let's say one percent higher is dot. The maximum value is just take everything above the one percent of that the value so all the noise is gone and I'm only taking this thresholded once and convert all of those pixels to blue.

In [68]:
```
image[harris>0.01*harris.max()]=[255,0,0]
```

In [69]:
```
plt.imshow(image)
```

Out[69]:
```
<matplotlib.image.AxesImage at 0x1c0b1e67280>
```



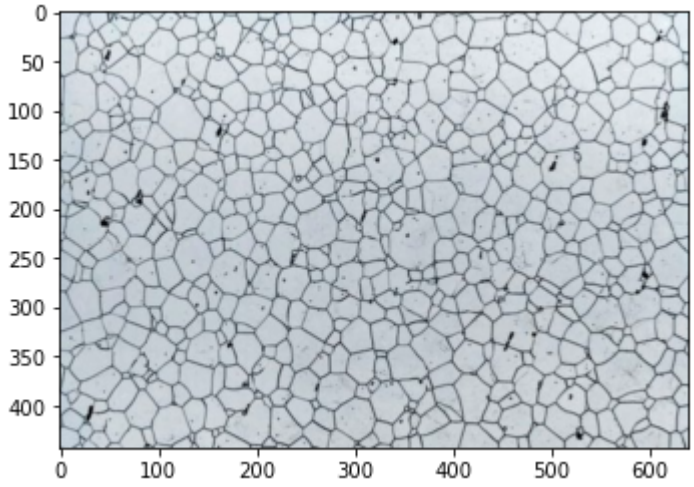
Harris algorithm tells us that red color points are the unique points that it actually thinks are the best points for this specific image to work on. This method and the previous one these are key point detectors not descriptors these are key point detectors meaning they detect where the key points are.So, We are looking for both a detector and a descriptor and it has the information about the key points and it also has the information about the description of those key points.

FAST: Features from Accelerated Segment Test -detect corners.

In [88]:
```
img = cv2.imread('grains1.jpg')
```

In [89]:
```
plt.imshow(img)
```

Out[89]:
```
<matplotlib.image.AxesImage at 0x1c0c1bb4760>
```

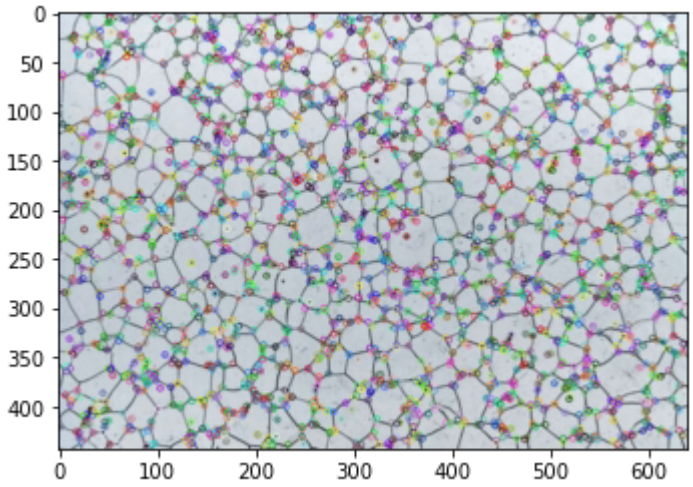


In [90]:
```
# Initiate FAST object with default values
detector = cv2.FastFeatureDetector_create(50)#Detects 50 points
key_points = detector.detect(img, None)
```

In [91]:
```
img2 = cv2.drawKeypoints(img, key_points, None, flags=0)
```

In [95]:
```
print("Circular key point of grains image:", plt.imshow(img2))
```

```
Circular key point of grains image: AxesImage(54,36;334.8x217.44)
```



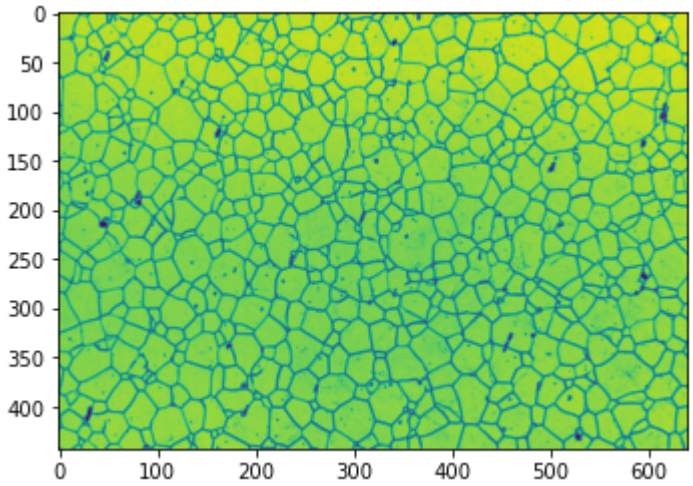
FAST is again just a feature detector it's not a feature descriptor.

We are looking for both feature detector and descriptor. One of the popular both feature detctor and descriptor is ORB- Oriented FAST (that we already described) and Rotated BRIEF(Binary Robust Independent Elementary Features)

In [96]:
```
#Again we use same grains image
img = cv2.imread('grains1.jpg',0)
```

In [98]:
```
plt.imshow(img)
```

Out[98]:
```
<matplotlib.image.AxesImage at 0x1c084340880>
```



In [102…
```
ORB=cv2.ORB_create(100) #100 data points

#Unpack the keypont and descriptor separate
```

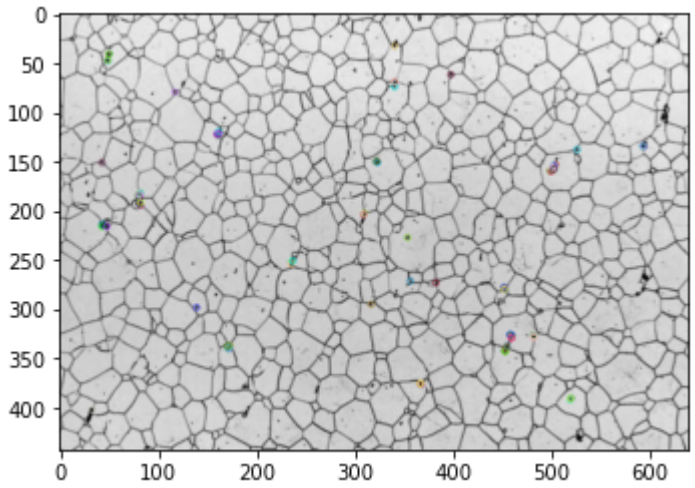
```
key_points,descriptor=ORB.detectAndCompute(img, None)
```

In [107…

```
# draw only keypoints location,not size and orientation
img2 = cv2.drawKeypoints(img, key_points, None, flags=None)
plt.imshow(img2)
```

Out[107…

```
<matplotlib.image.AxesImage at 0x1c0c2103790>
```

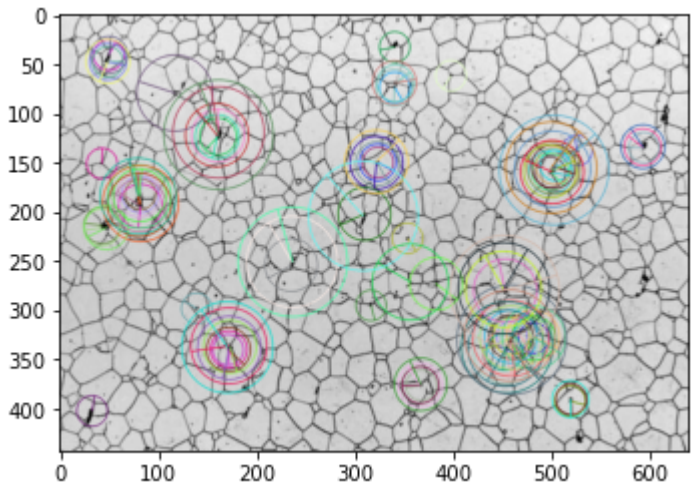


In [108…

```
# Now, let us draw with rich key points, reflecting descriptors.
# Descriptors here show both the scale and the orientation of the keypoint.
img2 = cv2.drawKeypoints(img, key_points, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_
plt.imshow(img2)
```

Out[108…

```
<matplotlib.image.AxesImage at 0x1c0c2176310>
```



Instead of just showing the key points ORB actually showing the circles that have different sizes and if we see there is a line inside these circles which is representing an angle. So there is the circle size reflects the let's say the strength of that key point the angle is in a way reflects the direction of that key point based on the neighboring pixels.

We will see an application where we have two images one reference and one warped or deformedand then take the deformed and change it transforms and stretches it to match or to

register with the reference image and this is probably needed for microscopy because most microscopes drift especially the older ones and if we want to collect images over a long period of time so registration may be something that we can find useful.

Steps that I followed for Image Registration:

Step 1: Import two Images (Original And Wrap Image);(

Step 2:Convert to Grayscale;

Step 3:Initiate ORB Detector;

Step 4:Find Keypoints and describe them;

Step 5: Match key_points between two images by using Brute-Force matcher **(Brute-Force matcher takes the descriptor of one feature in first set(Wrapped_Image) and is matched with all other features in second set(Original_Image) using some distance calculation).**

Step 6:Use RANSAC **(Random sample consensus, or RANSAC, is an iterative method for estimating a mathematical model from a data set that contains outliers, overall, it can be considered as outlier rejection method for keypoints)** to extract bad key_points;

Step 7:At last, Register two images(Use Homography- It looks at a 2D plane and a 2D plane that's warped and then tries to match these are based on the key points that are defined in both of these images).

In [49]:
```
im1 = cv2.imread('Mnkey_Wrapped.jpg') # Image that needs to be registered.
im2 = cv2.imread('Monkey_Original.jpg') # Reference Image
```

In [50]:
```
img1 = cv2.cvtColor(im1, cv2.COLOR_BGR2GRAY)
img2 = cv2.cvtColor(im2, cv2.COLOR_BGR2GRAY)
```

In [51]:
```
#Initiate ORB
orb=cv2.ORB_create(50) #Registration works with at least 50 points
```

In [52]:
```
key_points1,descriptor1=orb.detectAndCompute(img1, None)
key_points2,descriptor2=orb.detectAndCompute(img2, None)
```

In [53]:
```
# img3=cv2.drawKeypoints(img1,key_points1,None,flags=None)
# img4=cv2.drawKeypoints(img2,key_points2,None,flags=None)
```

In [54]:
```
# cv2.imshow("img3",img3)
# cv2.imshow("img4",img4)
# cv2.waitKey(0)
```

In [55]:
```
# Create Matcher
matcher = cv2.DescriptorMatcher_create(cv2.DESCRIPTOR_MATCHER_BRUTEFORCE_HAMMING)
```

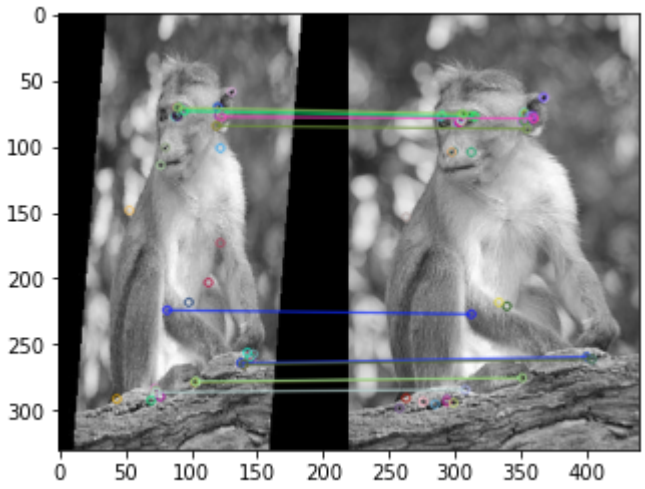
In [56]:
```
#Match the Descriptor
matches = matcher.match(descriptor1, descriptor2, None) #Creates a list of all matches,
```

In [57]:
```
#Now Sorting the Distance
matches=sorted(matches, key= lambda x:x.distance)
```

In [58]:
```
img3=cv2.drawMatches(img1,key_points1,img2,key_points2, matches[:10],None)
print("Give me top 10 matches:")
plt.imshow(img3)
```

```
Give me top 10 matches:
```

Out[58]:
```
<matplotlib.image.AxesImage at 0x244019b2160>
```



In [59]:
```
#Extract location of good matches.For this we will use RANSAC.
#Here,I am going to do is start with a empty array that's filled with zeros and then re
#keypoints that are indexed how long do I want these this this list to be it is as big
points1=np.zeros((len(matches),2),dtype=np.float32)
points2=np.zeros((len(matches),2),dtype=np.float32)
```

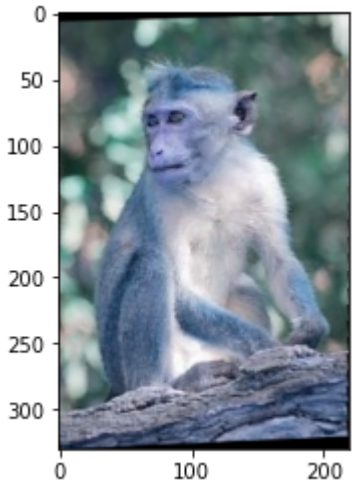
In [60]:
```
#For filling this zeros I use Enumerate function for unwrapping the matches.
for i, match in enumerate(matches):
    points1[i, :] = key_points1[match.queryIdx].pt #gives index of the descriptor in th
    points2[i, :] = key_points2[match.trainIdx].pt #gives index of the descriptor in th
print("The coordinates of our key points in the first image and in the second image:",p
```

```
The coordinates of our key points in the first image and in the second image: (47, 2)
```

In [61]:
```
#Now we have all good keypoints so we are ready for homography.
h, mask = cv2.findHomography(points1, points2, cv2.RANSAC)
# Use homography
height, width, channels = im2.shape #im2 is reference image
#Applies a perspective transformation to an image.
im1Registered=cv2.warpPerspective(im1, h, (width, height))
```

In [84]:
```
plt.imshow(im1Registered)
```

Out[84]:
```
<matplotlib.image.AxesImage at 0x170dac263a0>
```



Summary: Original image one, image two which is our reference image, key points that are matched and image one transformed by using the key points and using the homography matrix that we just calculated and unmarked it and the above image it looks like the original image.

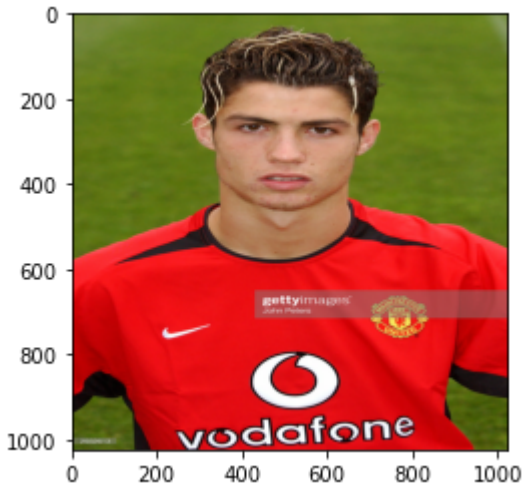
Image Resolution

Here, I have tried to reduce the size of an image by a factor of 2 from (1024,1024) to (32,32). The image was resized to the original size after changing the resolution. The results show that the image got blurrier as we move from higher to lower resolution.

In [13]:
```
ronaldo_image=cv2.imread("ronaldo.jpg")
#ronaldo_image.astype(dtype=int)
ronaldo_image_rgb=cv2.cvtColor(ronaldo_image, cv2.COLOR_BGR2RGB)
ronaldo_image_resize=cv2.resize(ronaldo_image_rgb,(1024,1024))
```

In [14]:
```
plt.imshow(ronaldo_image_resize)
print("The dimension of reiszed image:",ronaldo_image_resize.shape)
```

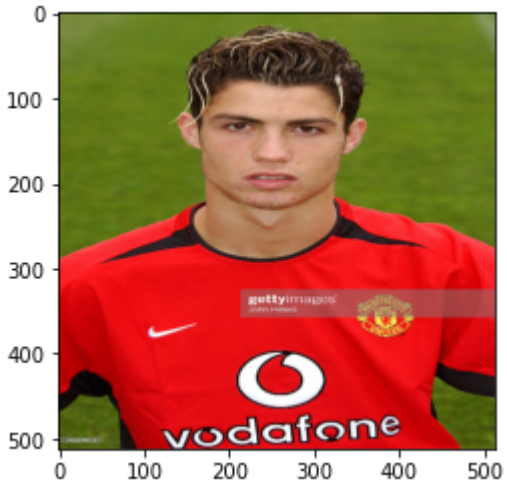
```
The dimension of reiszed image: (1024, 1024, 3)
```



In [15]:

```
ronaldo_image_resize_1=cv2.resize(ronaldo_image_rgb,(512,512))
plt.imshow(ronaldo_image_resize_1)
print("The dimension of reiszed image:",ronaldo_image_resize_1.shape)
```

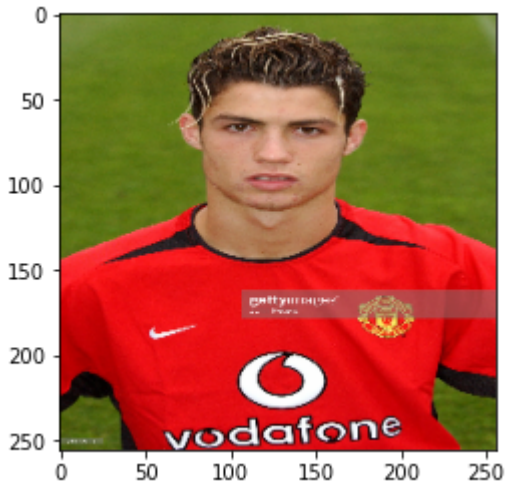
```
The dimension of reiszed image: (512, 512, 3)
```



In [16]:

```
ronaldo_image_resize_2=cv2.resize(ronaldo_image_rgb,(256,256))
plt.imshow(ronaldo_image_resize_2)
print("The dimension of reiszed image:",ronaldo_image_resize_2.shape)
```

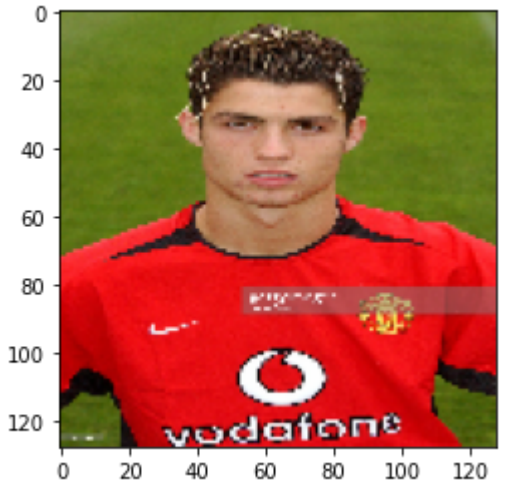
```
The dimension of reiszed image: (256, 256, 3)
```



In [17]:

```
ronaldo_image_resize_3=cv2.resize(ronaldo_image_rgb,(128,128))
plt.imshow(ronaldo_image_resize_3)
print("The dimension of reiszed image:",ronaldo_image_resize_3.shape)
```

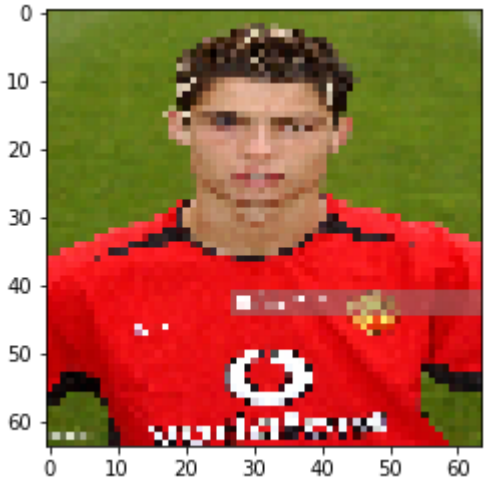
```
The dimension of reiszed image: (128, 128, 3)
```



In [18]:

```
ronaldo_image_resize_4=cv2.resize(ronaldo_image_rgb,(64,64))
plt.imshow(ronaldo_image_resize_4)
print("The dimension of reiszed image:",ronaldo_image_resize_4.shape)
```

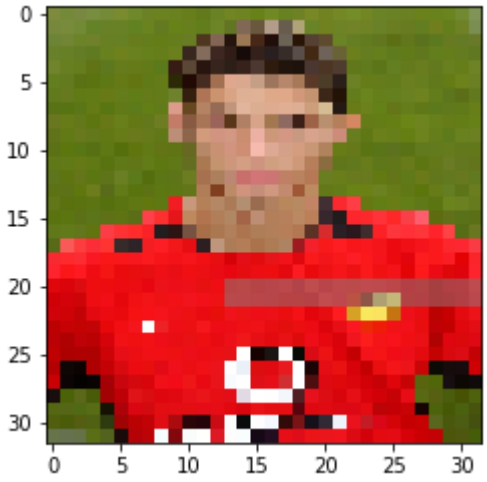
```
The dimension of reiszed image: (64, 64, 3)
```



In [19]:

```
ronaldo_image_resize_5=cv2.resize(ronaldo_image_rgb,(32,32))
plt.imshow(ronaldo_image_resize_5)
print("The dimension of reiszed image:",ronaldo_image_resize_5.shape)
```

```
The dimension of reiszed image: (32, 32, 3)
```



Quantization

Image quantization involves changing the range of pixel values. I achieved this by applying the following equation to the original image.

$$Quantization = ((image - imagemin)/(imagemax - imagemin)) * (Rmax - Rmin)$$

where, image= original image

imagemin=Munimum pixel of on original Image

imagemax=Maximum pixel of an Image

Rmax, Rmin : Represents maximum and minimum value of the target range

For example: Consider of an given image as

[[20 40 20 20]

[80 90 150 0]

[40 90 200 100]

[100 10 20 30]]

matrix. How can we quantize the given image to 8 gray levels?

A simple solution: Find the smallest and largest value (0 & 200). Divide the range length by 8 to get 25. Divide each pixel by 25, throw away the remainder. The new values are now quantized to 8 gray levels.

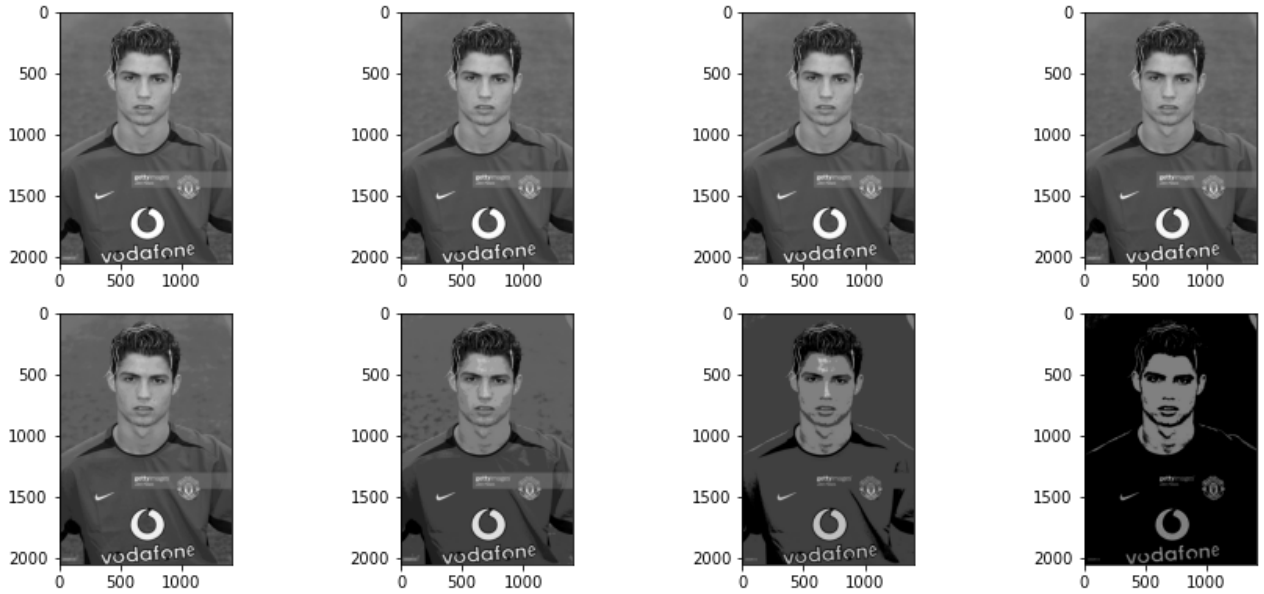
By applying this quantization, I obtained 8 quantization ranges and plotted the corresponding images. As quantization range reduced image(gray) became darker and darker as we approach the range of (0-2).

SNR calculated above is plotted over the corresponding max range.

In [69]:
```
ronaldo1=cv2.imread("ronaldo.jpg")
```

In [76]:
```
ronaldo=cv2.cvtColor(ronaldo1, cv2.COLOR_BGR2GRAY)
```

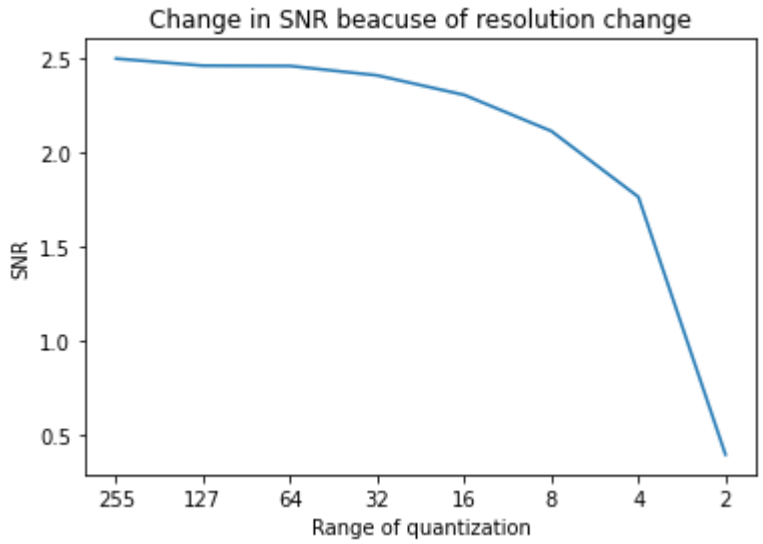
In [77]:
```
## Quantization function takes original image required range of pixel values and it ret
def quantization( ronaldo, rmin,rmax):
    imgmax=ronaldo.max()
    imgmin=ronaldo.min()
    quantized=((ronaldo-imgmin)/(imgmax-imgmin))*(rmax-rmin)
    quantized = quantized.astype(np.uint8)
    #print(quantized)
    return quantized

ronaldo_quantized=[quantization(ronaldo,0,255),quantization(ronaldo,0,127),quantization


snr=[]
fig=plt.figure(figsize=(15,10))
for i in range(1,len(ronaldo_quantized)+1):
    fig.add_subplot(3,4,i)
    snr.append(snr_calculation(ronaldo_quantized[i-1]))
    plt.imshow(ronaldo_quantized[i-1],cmap='gray') #Capmp GrayChoosing Colormaps as gra
plt.show()
```



In [23]:

```
## SNR calculated above is plotted over the corresponding max range
x=["255","127","64","32","16","8","4","2"]
plt.plot(x,snr)
plt.ylabel("SNR")
plt.xlabel("Range of quantization")
plt.title("Change in SNR beacuse of resolution change")
print(len(snr))
```

8

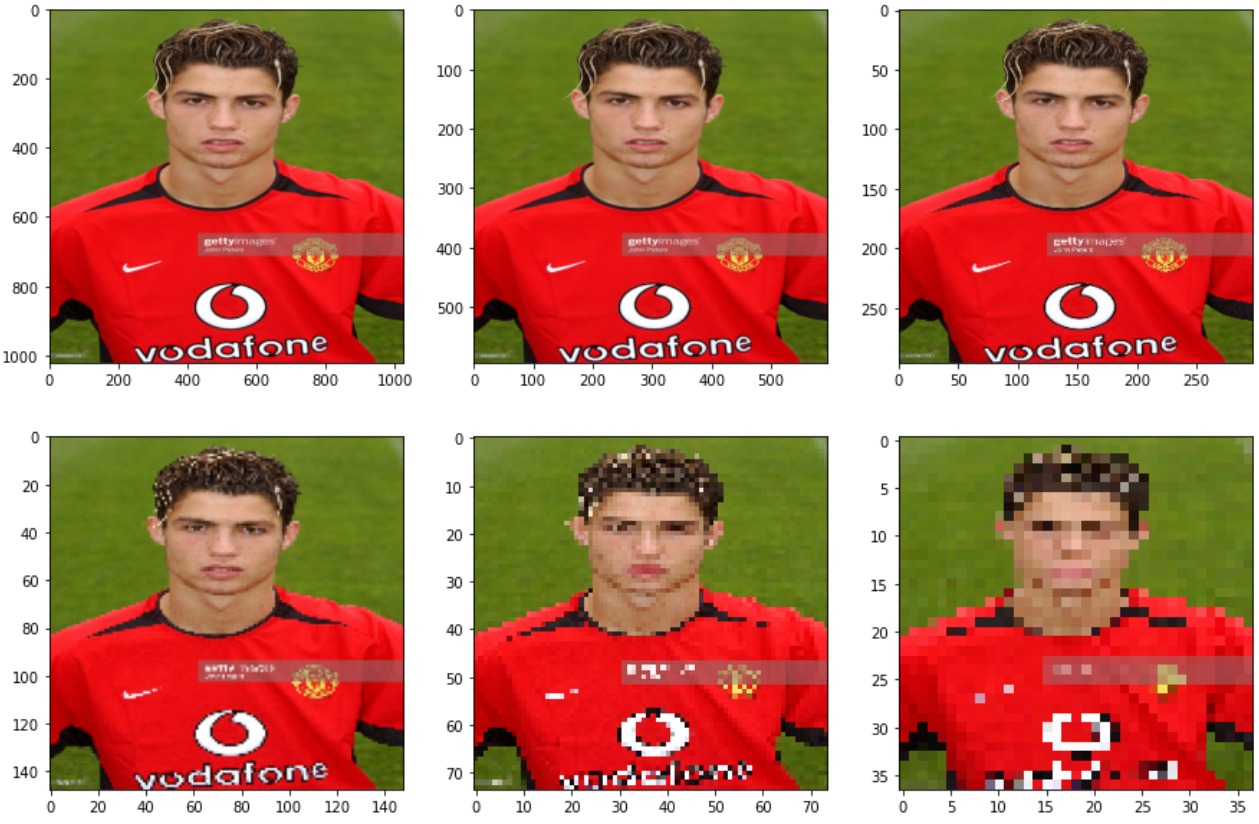


Multi-resolution: Reduce the size the image progressively by a factor of 1.72 and display images.

This part is very similar to image resolution but here I have reduced the resolution of the original image by the factor of 1.72 progressively and the resulting image dimensions were maintained. By plotting all images, images with low resolution became blurrier.

In [24]:
```
ronaldo_image=cv2.imread("ronaldo.jpg")
#ronaldo_image.astype(dtype=int)
ronaldo_image_rgb=cv2.cvtColor(ronaldo_image, cv2.COLOR_BGR2RGB)
ronaldo_image_resize=cv2.resize(ronaldo_image_rgb,(1024,1024))
```

In [28]:
```
# Dimension is reduced by factor of 1.72
# Dimension was converted to integer values since resize function only takes integers.
dimension1=1024
dimension2=int(dimension1/1.72)
dimension3=int(dimension1/3.44)
dimension4=int(dimension1/6.88)
dimension5=int(dimension1/13.76)
dimension6=int(dimension1/27.52)

i1=cv2.resize(ronaldo_image_resize,(dimension2,dimension2))

i2=cv2.resize(ronaldo_image_resize,(dimension3,dimension3))

i3=cv2.resize(ronaldo_image_resize,(dimension4,dimension4))

i4=cv2.resize(ronaldo_image_resize,(dimension5,dimension5))

i5=cv2.resize(ronaldo_image_resize,(dimension6,dimension6))

ronaldo=[ronaldo_image_resize,i1,i2,i3,i4,i5]

fig= plt.figure(figsize=(15,10))

for i in range(1,len(ronaldo)+1):
    fig.add_subplot(2,3,i)
    plt.imshow(ronaldo[i-1])
```



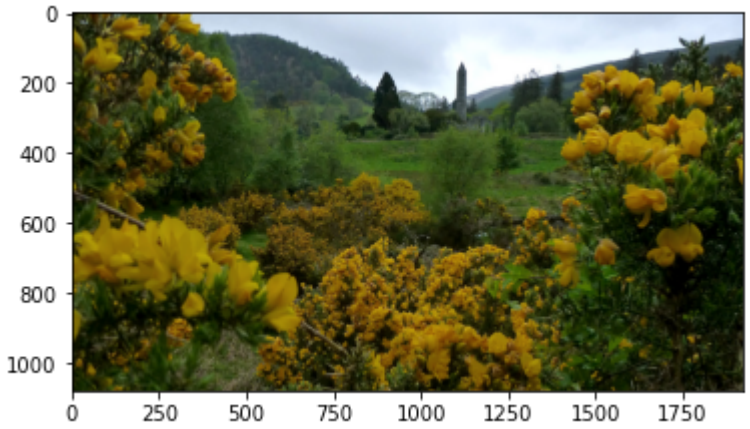
Bonus

Zoom Out

In this program, I have down-sampled an image. Downsampling is decreasing the spatial resolution while keeping the 2D representation of an image. It is typically used for zooming out of an image. I have used the pyrdown() function in the openCV library to complete this task. Then, I have checked of the original and zoomed-out image.

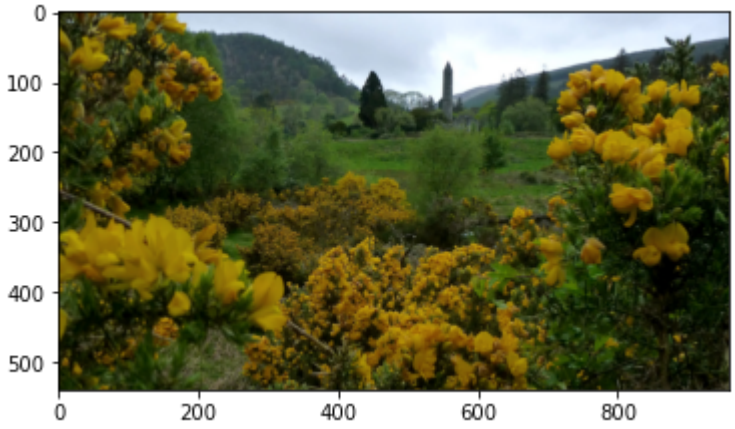
In [19]:
```
img = cv2.imread("Green View.jpg")
img1=cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
print("Size of the original image: ", img1.shape)
plt.imshow(img1)
```

Out[19]:
```
Size of the original image:  (1080, 1920, 3)
<matplotlib.image.AxesImage at 0x1b412734d90>
```



In [20]:
```
image = cv2.pyrDown(img1)
print("Size of image after pyrDown: ", image.shape)
plt.imshow(image)
```

Out[20]:
```
Size of image after pyrDown:  (540, 960, 3)
<matplotlib.image.AxesImage at 0x1b40882f700>
```



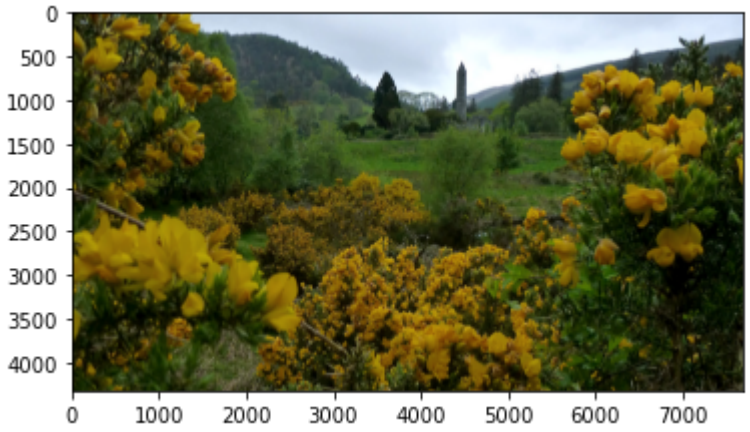
Zoom-In

In this program, I have up sampled an image. Up sampling is increasing the spatial resolution while keeping the 2D representation of an image. It is typically used for zooming in on an image. I have used the pyrup() function in the openCV library to complete this task and checked shape of the both Image..

In [23]:
```
image = cv2.pyrUp(image)
print("Size of image after pyrUp: ", image.shape)
plt.imshow(image)
```

```
Size of image after pyrUp:  (4320, 7680, 3)
```

Out[23]:
```
<matplotlib.image.AxesImage at 0x1b413fa5c70>
```



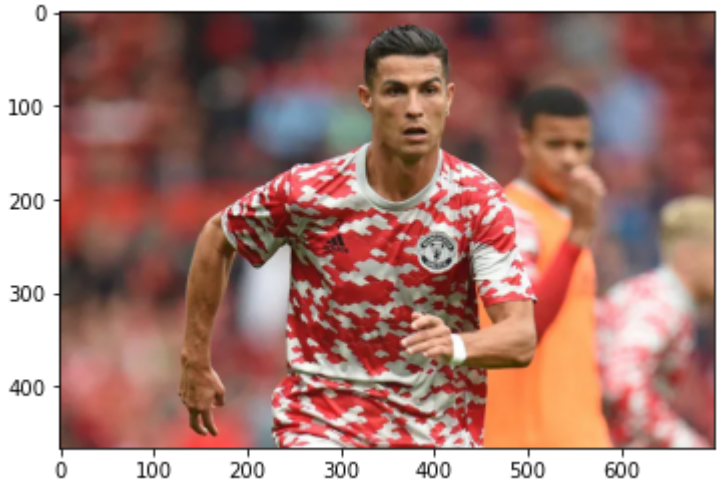
Shrinking

Shrinking involves reduction of pixels and it means lost of irrecoverable information. zooming may be viewed as oversampling, while shrinking may be viewed as undersampling. The key difference between these two operations and sampling and quantizing an original continuous image is that zooming and shrinking are applied to a digital image

In [7]:
```
cr7 = cv2.imread("RonaldoWarmup.jpg")
cr7_col=cv2.cvtColor(cr7, cv2.COLOR_BGR2RGB)
print('Original Image Shape:', cr7_col.shape)
plt.imshow(cr7_col)
```

```
Original Image Shape: (466, 700, 3)
```

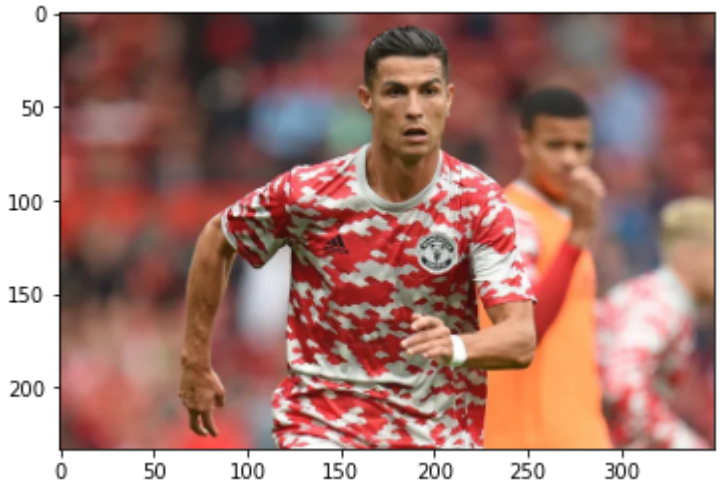
Out[7]:
```
<matplotlib.image.AxesImage at 0x2abed333a00>
```



In [11]:
```
#percent by which the image is resized
scale_percent = 50

#calculate the 50 percent of original dimensions
width = int(cr7_col.shape[1] * scale_percent / 100)
height = int(cr7_col.shape[0] * scale_percent / 100)

# dsize
dsize = (width, height)
```

In [13]:
```
# resize image
output = cv2.resize(cr7_col, dsize)
print("Output Image Shape:",output.shape)

plt.imshow(output)
```

```
Output Image Shape: (233, 350, 3)
```
Out[13]:
```
<matplotlib.image.AxesImage at 0x2abed446610>
```



Multi-Resolutions by Using Pyramid

Normally, we used to work with an image of constant size. But on some occasions, we need to work with the same images in different resolution. For example, while searching for something in an

image, like face, we are not sure at what size the object will be present inside image. In that case, we will need to create a set of the same image with different resolutions and search for object in all of them. These set of images with different resolutions are called Image Pyramid.

There are two kinds of Image Pyramids.

1) Gaussian Pyramid and 2) Laplacian Pyramid

I have used Gaussian Pyramid here. That is used pyrDown() function.

In [19]:
```
i = cv2.imread("RonaldoWarmup.jpg")
layer = img.copy()
gaussian_pyramid_list = [layer]
```

In [21]:
```
for i in range(6):
    layer = cv2.pyrDown(layer)
    gaussian_pyramid_list.append(layer)
    cv2.imshow(str(i), layer)
cv2.imshow("Original Image",img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```