Technological Institute of the Philippines	Quezon City - Computer Engineering
Course Code:	CPE 313
Code Title:	Advanced Machine Learning and Deep Learning Discussion
1st Semester	AY 2024-2025

ACTIVITY NO. 4	**Edge and Contour Detection**
Name	Montojo, Lance M.
Section	CPE31S3
Date Performed:	02/21/2025
Date Submitted:	02/21/2025
Instructor:	Engr. Roman M. Richard

1. Objectives

This activity aims tointroduce students to the use of OpenCV for edge detection and contour detection.

2. Intended Learning Outcomes (ILOs)

After this activity, the students should be able to:

- Explain fundamental idea of convolution and the kernel's importance.
- Use different image manipulation methods by using openCV functions.
- Perform contour and edge line drawing on their own images.

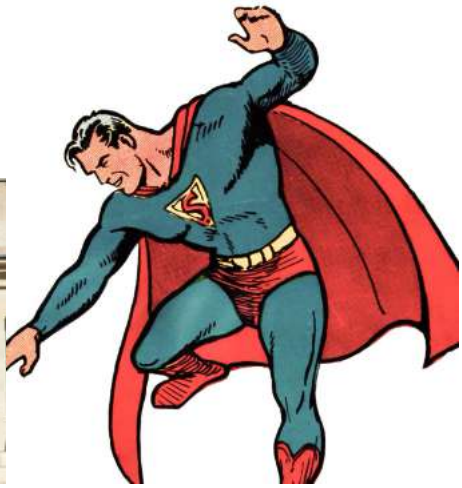
3. Procedures and Outputs

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
import numpy as np
import cv2

# This is the file URL: https://media.wired.com/photos/5cdefb92b86e041493d389df/4:3/w_1330,h_998,c_limit/Culture-Grumpy-Cat-487386121.jpg
src = cv2.imread('/content/drive/MyDrive/CPE 313/Images/Grumpy Cat.png')
dst = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)
```

Edges play a major role in both human and computer vision. We, as humans, can easily recognize many object types and their pose just by seeing a backlit silhouette or a rough sketch. Indeed, when art emphasizes edges and poses, it often seems to convey the idea of an archetype, such as Rodin's The Thinker or Joe Shuster's Superman.



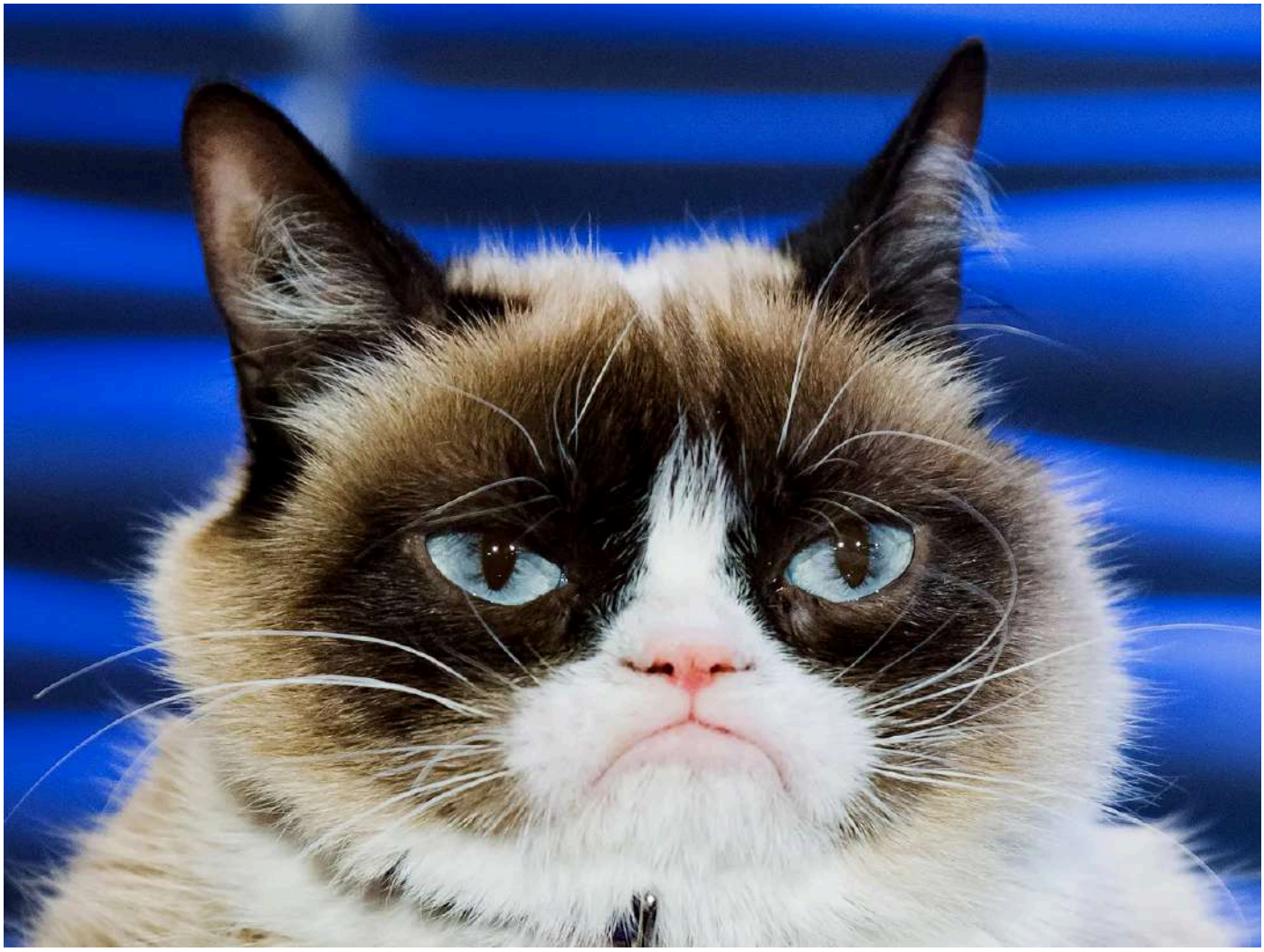
Software, too, can reason about edges, poses, and archetypes.

OpenCV provides many edge-finding filters, including Laplacian(), Sobel(), and Scharr(). These filters are supposed to turn non-edge regions to black while turning edge regions to white or saturated colors. However, they are prone to misidentifying noise as edges. This flaw can be mitigated by blurring an image before trying to find its edges. OpenCV also provides many blurring filters, including blur() (simple average), medianBlur(), and GaussianBlur(). The arguments for the edge-finding and blurring filters vary but always include ksize, an odd whole number that represents the width and height (in pixels) of a filter's kernel.

For blurring, let's use medianBlur(), which is effective in removing digital video noise, especially in color images. For edge-finding, let's use Laplacian(), which produces bold edge lines, especially in grayscale images. After applying medianBlur(), but before applying Laplacian(), we should convert the image from BGR to grayscale.

Once we have the result of Laplacian(), we can invert it to get black edges on a white background. Then, we can normalize it (so that its values range from 0 to 1) and multiply it with the source image to darken the edges. Let's implement this approach:

```
from google.colab.patches import cv2_imshow

cv2_imshow(src)
```



```
cv2_imshow(dst)
```

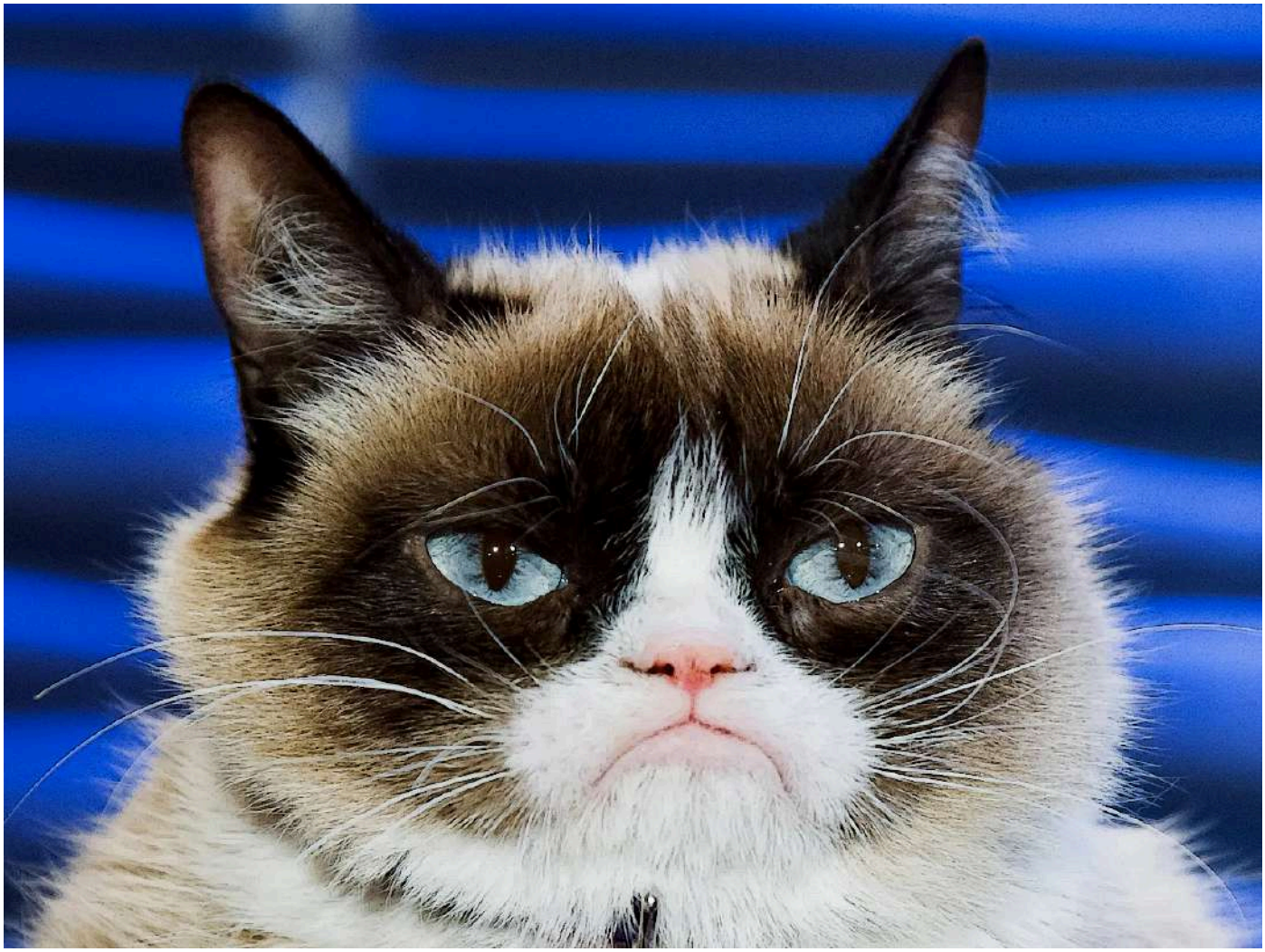
```
def strokeEdges(src, dst, blurKsize = 3, edgeKsize = 3):
  if blurKsize >= 3:
    blurredSrc = cv2.medianBlur(src, blurKsize)
    graySrc = cv2.cvtColor(blurredSrc, cv2.COLOR_BGR2GRAY)
  else:
    graySrc = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)

  cv2.Laplacian(graySrc, cv2.CV_8U, graySrc, ksize = edgeKsize)
  normalizedInverseAlpha = (1.0 / 255) * (255 - graySrc)
  channels = cv2.split(src)

  for channel in channels:
    channel[:] = channel * normalizedInverseAlpha

  return cv2.merge(channels,dst)
```

For the function above, provide an analysis:

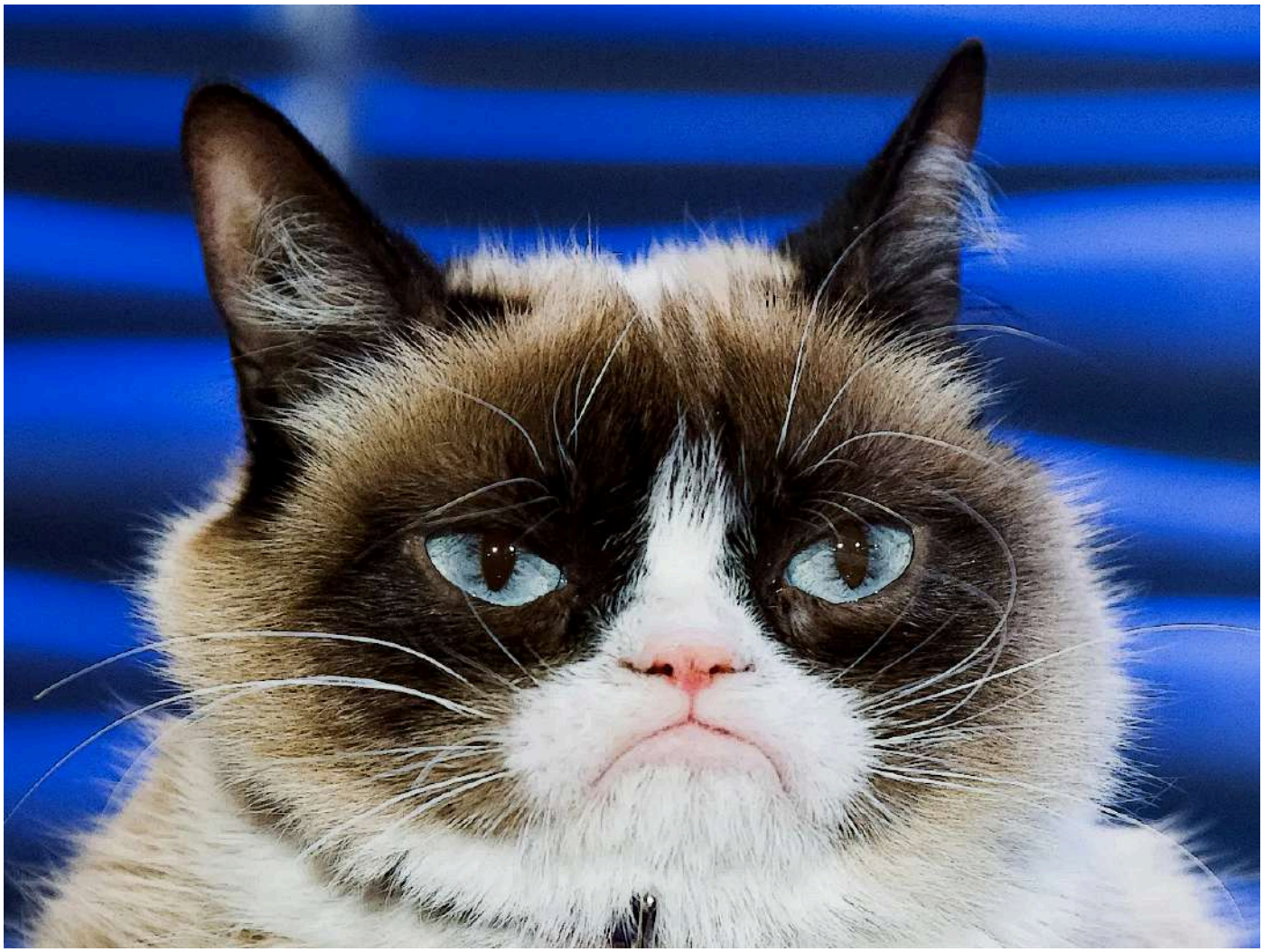
- Try to run the function and pass values for its parameters.
- Change the values in the kSize variables. What do you notice?

Make sure to add codeblocks underneath this section to ensure that your demonstration of the procedure and answers are easily identifiable

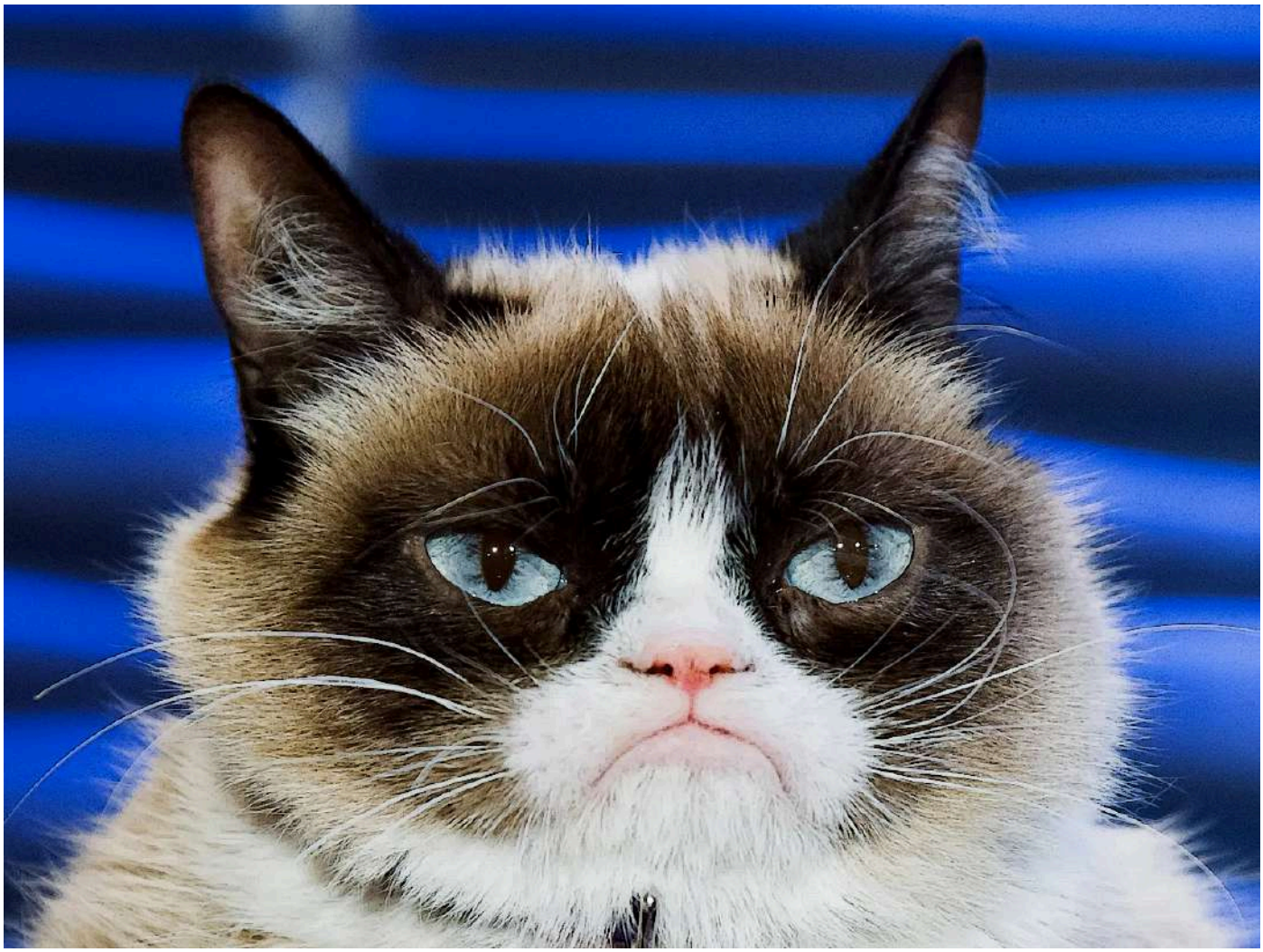
```
# Sample

new_img = strokeEdges(src, dst)
cv2_imshow(new_img)
```



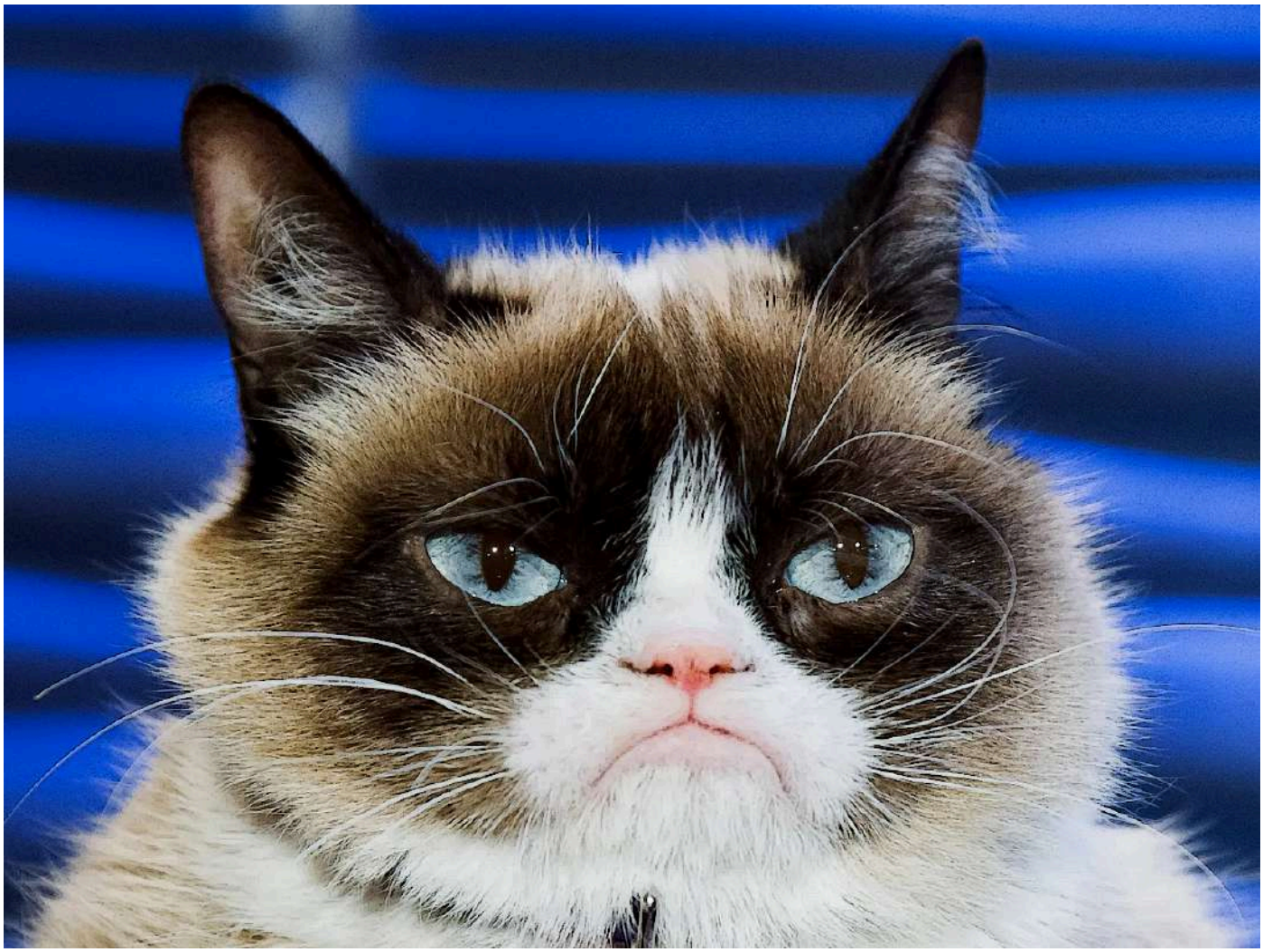
```
Ksizemodified_imgto_15 = strokeEdges(src, dst)
cv2_imshow(Ksizemodified_imgto_15)
```



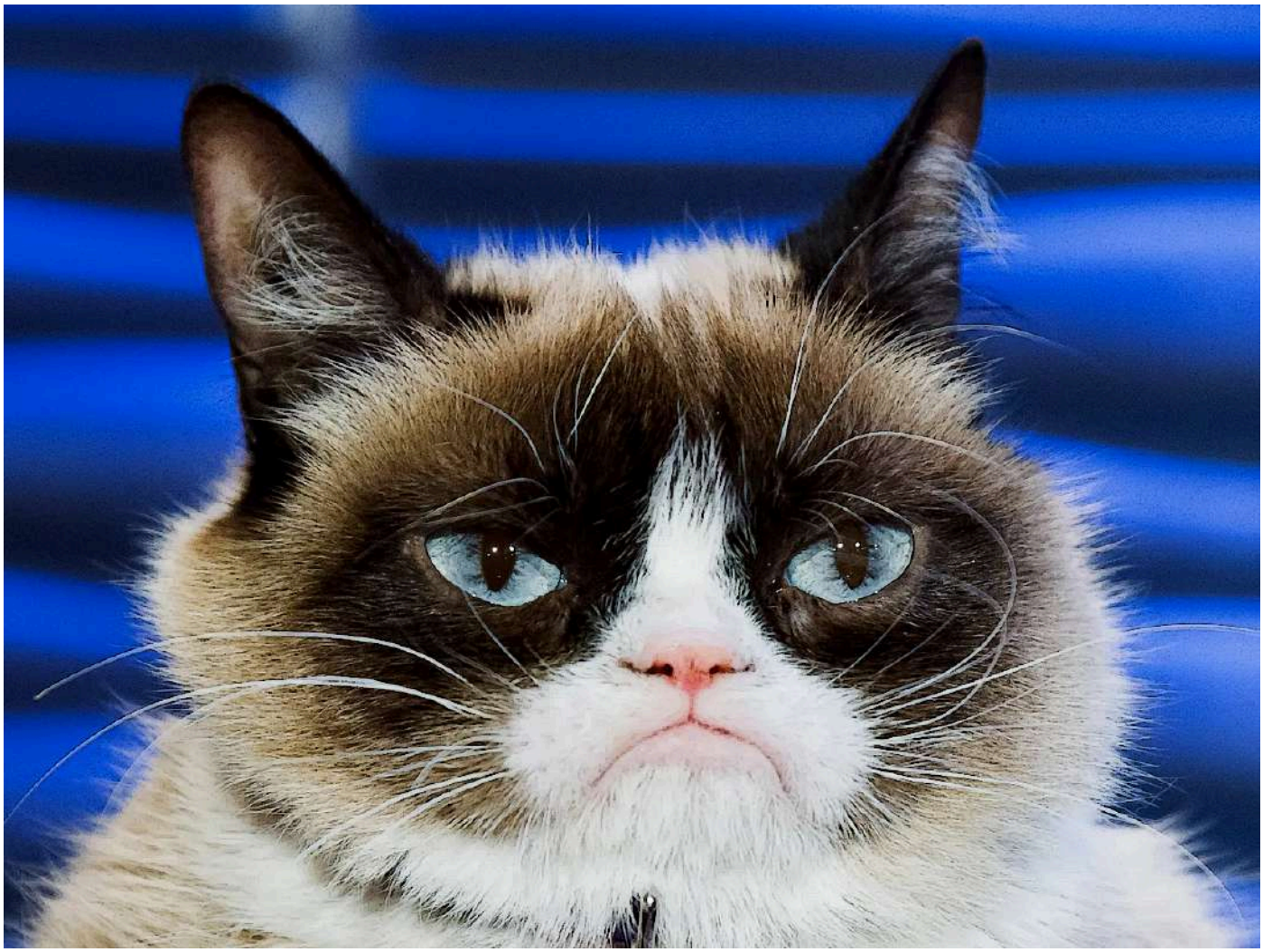
```
Ksizemodified_imgto_3 = strokeEdges(src, dst)
cv2_imshow(Ksizemodified_imgto_3)
```



```
Ksizemodified_imgto_15_3 = strokeEdges(src, dst)
cv2_imshow(Ksizemodified_imgto_15_3)
```



```
Ksizemodified_imgto_3_15 = strokeEdges(src, dst)
cv2_imshow(Ksizemodified_imgto_3_15)
```

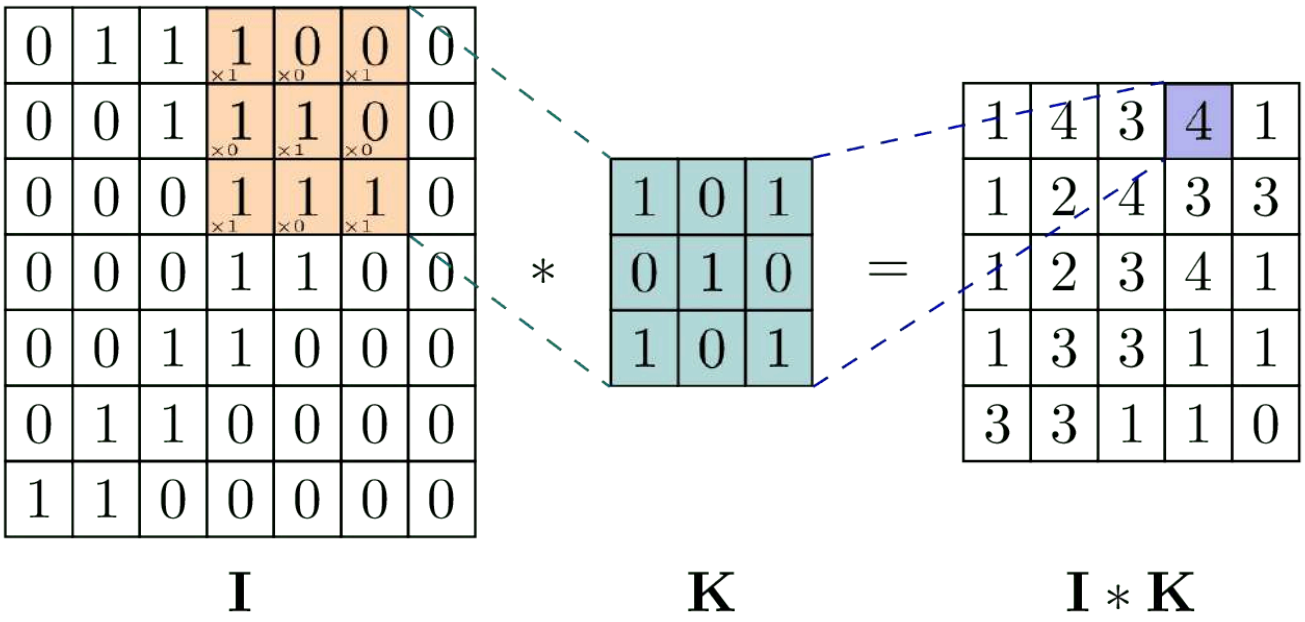


```
"""OBSERVATION HERE"""
"""I notice that the higher the edgeKsize is, the rougher the image looks.
If the blurkSize is higher, then the image would look smoother.
Also, if their parameters are set to be the same, like the one in Ksizemodified_imgto_15, it seems like edgeKsize is more prominent than blu
but it compensates by enhancing color and intensity, so it does not look more 'cartoony' than the one with both parameters set to 15."""
```

```
'I notice that the higher the edgeKsize is, the rougher the image looks.\nIf the blurkSize is higher, then the image would look smoothe
r.\nAlso, if their parameters are set to be the same, like the one in Ksizemodified_imgto_15, it seems like edgeKsize is more prominent
than blurkSize.\nbut it compensates by enhancing color and intensity, so it does not look more 'cartoony' than the one with both parame
```

Custom Kernels -- Getting Convoluted

As we have just seen, many of OpenCV's predefined filters use a kernel. Remember that a kernel is a set of weights, which determine how each output pixel is calculated from a neighborhood of input pixels. Another term for a kernel is a convolution matrix. It mixes up or convolves the pixels in a region. Similarly, a kernel-based filter may be called a convolution filter.

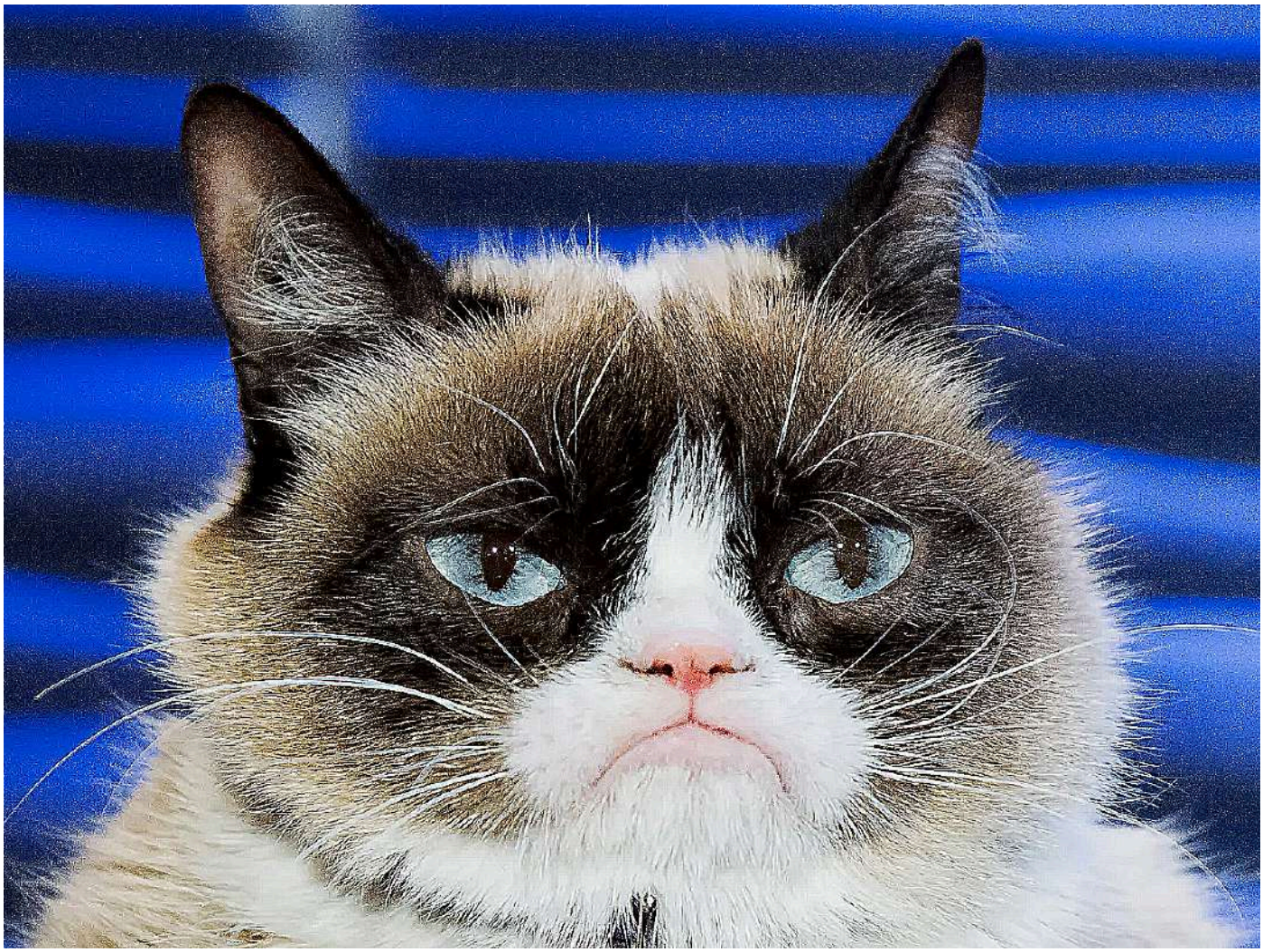


OpenCV provides a very versatile filter2D() function, which applies any kernel or convolution matrix that we specify. To understand how to use this function, let's first learn the format of a convolution matrix. It is a 2D array with an odd number of rows and columns. The central element corresponds to a pixel of interest and the other elements correspond to the neighbors of this pixel. Each element contains an integer or floating point value, which is a weight that gets applied to an input pixel's value.

```
# Example

kernel = np.array([[-1, -1, -1],
                   [-1, 9, -1],
                   [-1, -1, -1]])
```

Here, the pixel of interest has a weight of 9 and its immediate neighbors each have a weight of -1. For the pixel of interest, the output color will be nine times its input color minus the input colors of all eight adjacent pixels. If the pixel of interest is already a bit different from its neighbors, this difference becomes intensified. The effect is that the image looks sharper as the contrast between the neighbors is increased.

```
# Applying the Kernel

# Reloading src and dst
src = cv2.imread('/content/drive/MyDrive/CPE 313/Images/Grumpy Cat.png')
dst = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)

# Using the filter with our Kernel
new_img2 = cv2.filter2D(src, -1, kernel, dst)

# Display
cv2_imshow(new_img2)
```



Answer the following with analysis:

- What does filter2d() function do that resulted to this image above?
- Provide a comparison between this new image and the previous image that we were able to generate.

Filter2d() made the image more refined compared to what we generated earlier using strokeEdges. This is because the pixel of interest is nine times more pronounced while its eight neighbors in the kernel each have a weight of -1. This sharpens high-contrast areas like the cat's eyes and fur`

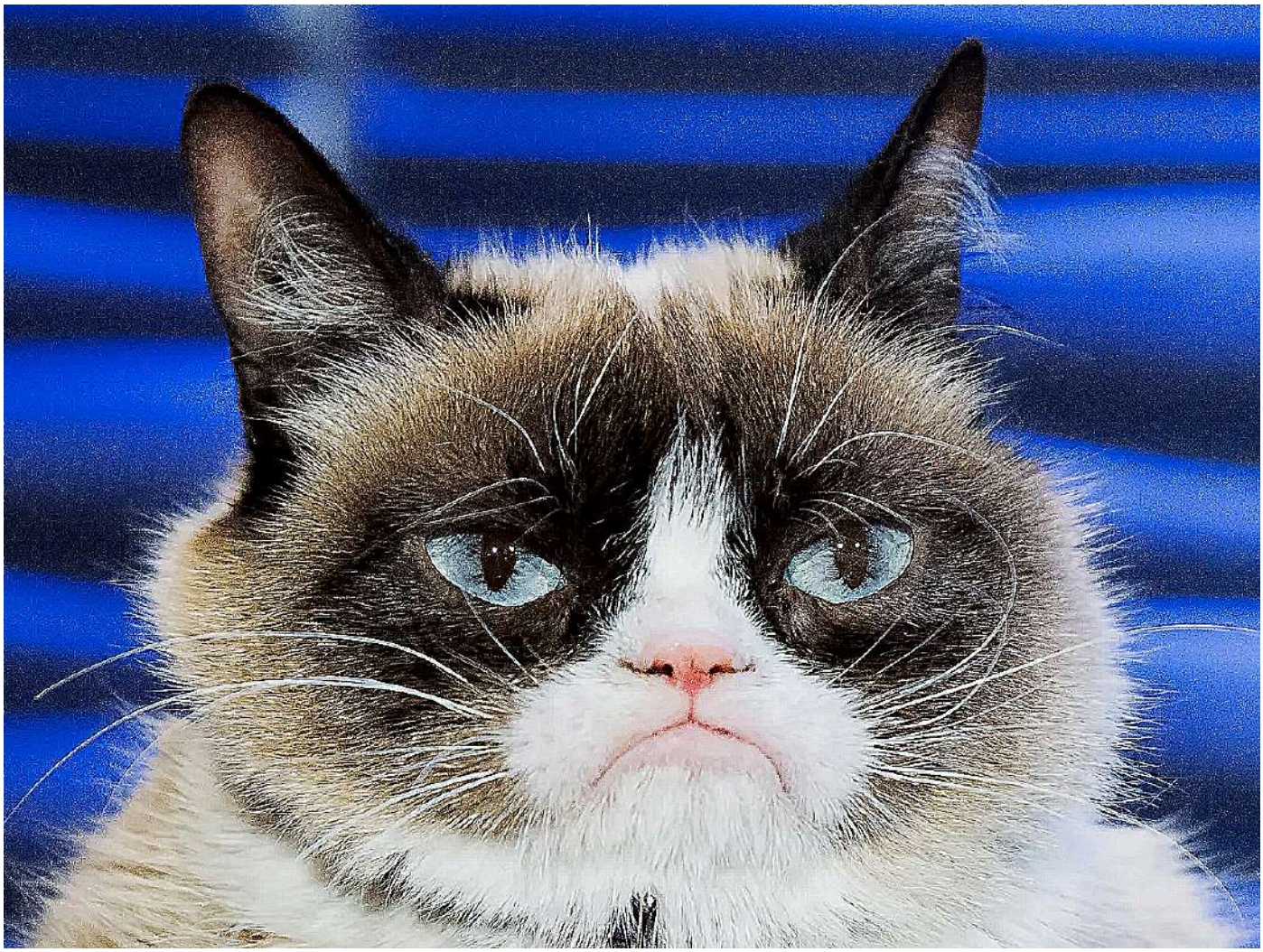
Based on this simple example, let's add two classes. One class, VConvolutionFilter, will represent a convolution filter in general. A subclass, SharpenFilter, will represent our sharpening filter specifically.

```
class VConvolutionFilter(object):
  """a filter that applies a convolution to V (or all of BGR)."""

  def __init__(self, kernel):
```

```
    self._kernel = kernel

  def apply(self, src, dst):
    """ Apply the filter with a BGR or gray source/destination """
    cv2.filter2D(src, -1, self._kernel, dst)


class SharpenFilter(VConvolutionFilter):
  """a sharpen filter with a 1-pixel radius."""

  def __init__(self):
    kernel = np.array([[-1, -1, -1],
                       [-1, 9, -1],
                       [-1, -1, -1]])
    VConvolutionFilter.__init__(self, kernel)
```

Run the classes above, create objects and aim to show the output of the two classes. Afterwards, make sure to make an analysis of the output.

```
src = cv2.imread('/content/drive/MyDrive/CPE 313/Images/Grumpy Cat.png')
src_copy = src.copy()

sharpen_filter = SharpenFilter()
sharpened_filter = sharpen_filter.apply(src, src_copy)

cv2_imshow(src_copy)
```



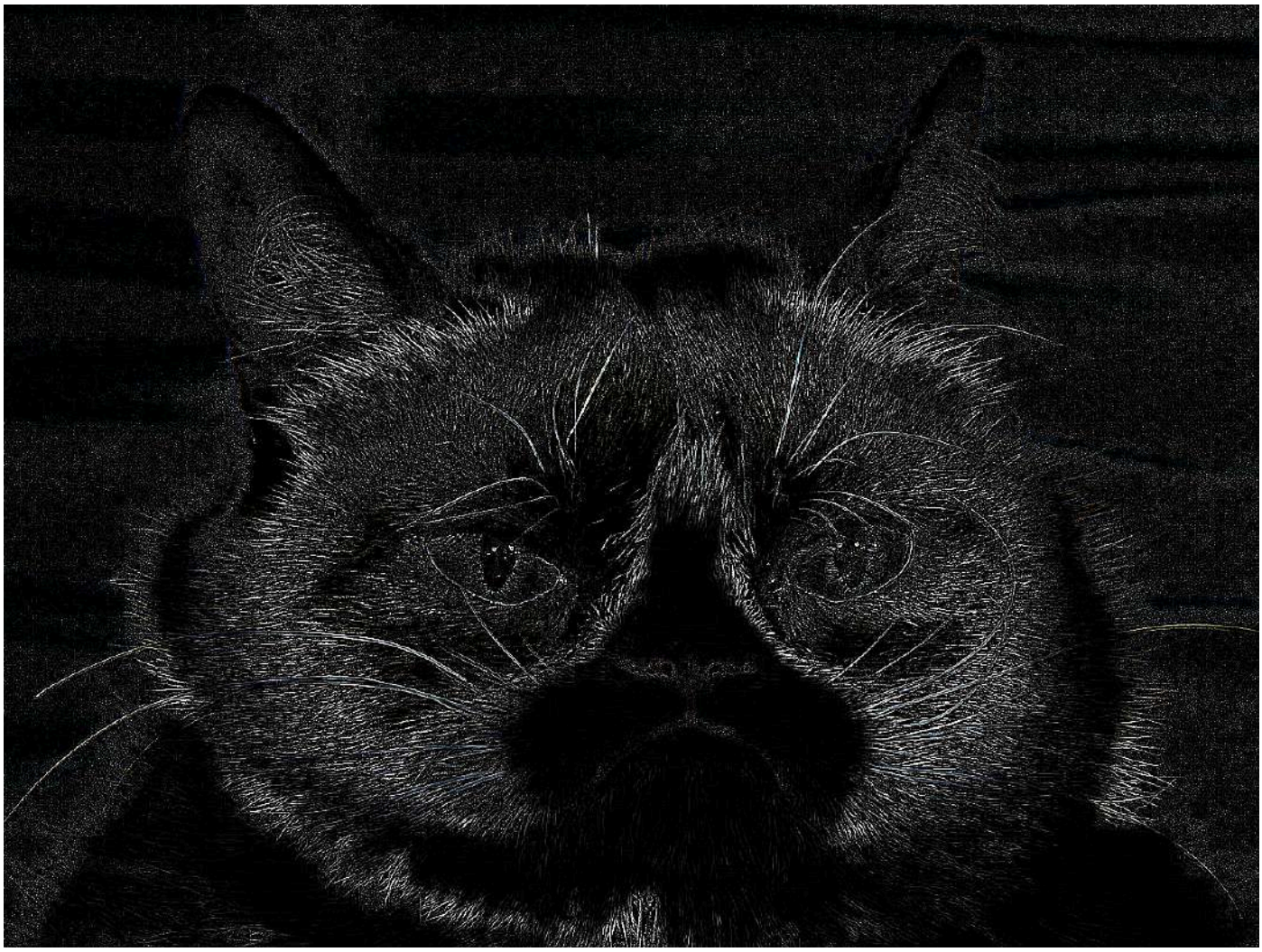
It looks the same as when we used the filter2D() where the features look more enhanced and you can see the details clearly. The sharpening effect makes edges and textures more distinct, improving clarity while preserving the overall structure of the image.

Note that the weights sum up to 1. This should be the case whenever we want to leave the image's overall brightness unchanged. If we modify a sharpening kernel slightly so that its weights sum up to 0 instead, we have an edge detection kernel that turns edges white and non-edges black.

```
class FindEdgesFilter(VConvolutionFilter):
  """An edge-finding filter with a 1-pixel radius."""

  def __init__(self):
    kernel = np.array([[-1, -1, -1],
                       [-1, 8, -1],
                       [-1, -1, -1]])
    VConvolutionFilter.__init__(self, kernel)
```

Run this class and demonstrate the output. Provide an analysis

```
FE_filter = FindEdgesFilter()
FE_copy = src.copy()
FE_filter.apply(src, FE_copy)

cv2_imshow(FE_copy)
```



The image lost its color and it enhanced the cat's features, such as its whiskers, eyes, and fur pattern, making them sharper and more pronounced. This highlights key structural elements while suppressing unnecessary details in low-contrast areas.

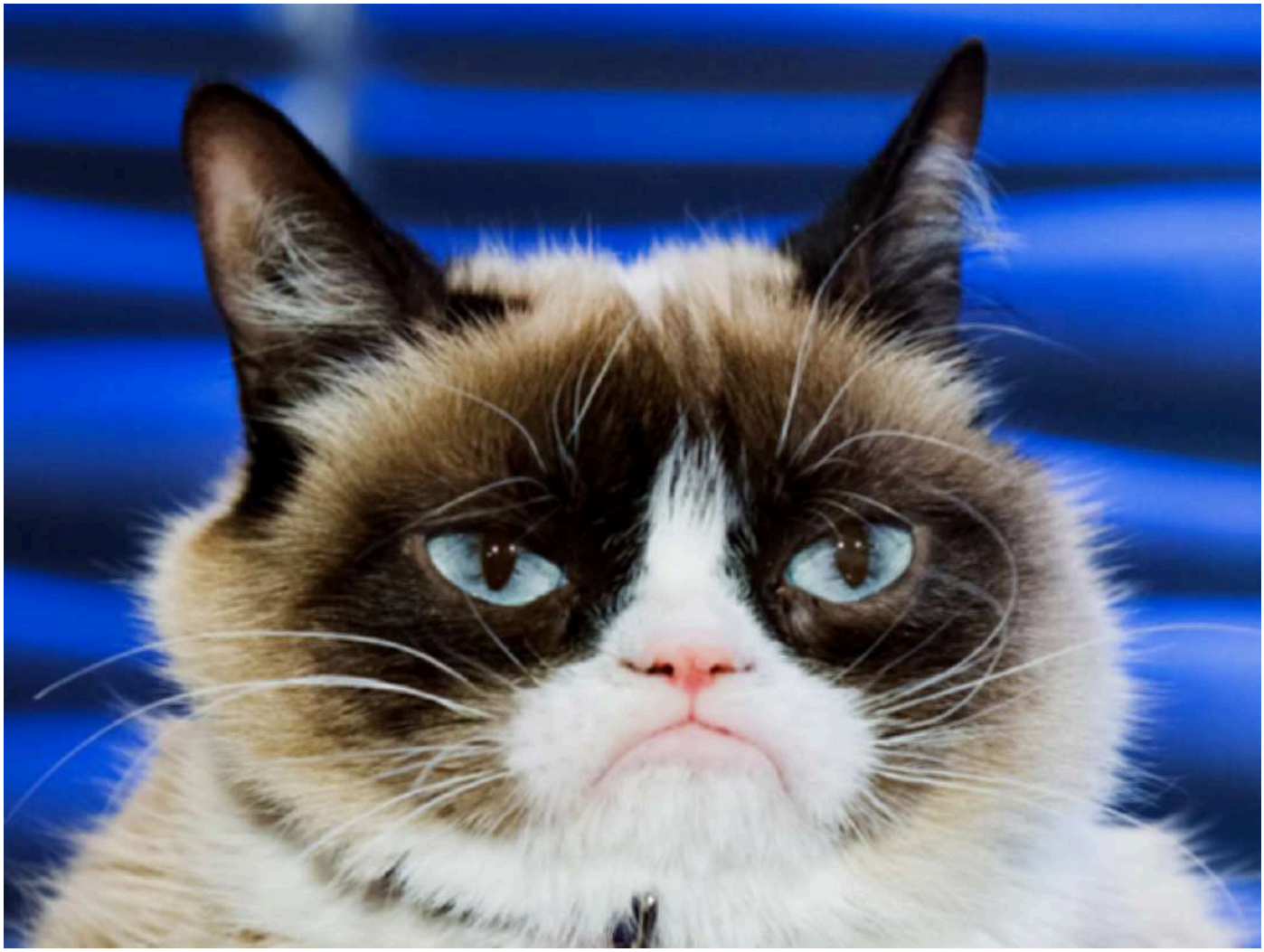
Next, let's make a blur filter. Generally, for a blur effect, the weights should sum up to 1 and should be positive throughout the neighborhood. For example, we can take a simple average of the neighborhood as follows

```
class Blurfilter(VConvolutionFilter):
  """A blur filter with a 2-pixel radius"""

  def __init__(self):
    kernel = np.array([[0.04, 0.04, 0.04, 0.04, 0.04],
                       [0.04, 0.04, 0.04, 0.04, 0.04],
                       [0.04, 0.04, 0.04, 0.04, 0.04],
                       [0.04, 0.04, 0.04, 0.04, 0.04],
```

```
                      [0.04, 0.04, 0.04, 0.04, 0.04]])
    VConvolutionFilter.__init__(self, kernel)
```

Run this class and demonstrate the output. Provide an analysis

```
Blur_filter = Blurfilter()
Blur_copy = src.copy()
Blur_filter.apply(src, Blur_copy)

# BlurredFilter image
cv2_imshow(Blur_copy)
```



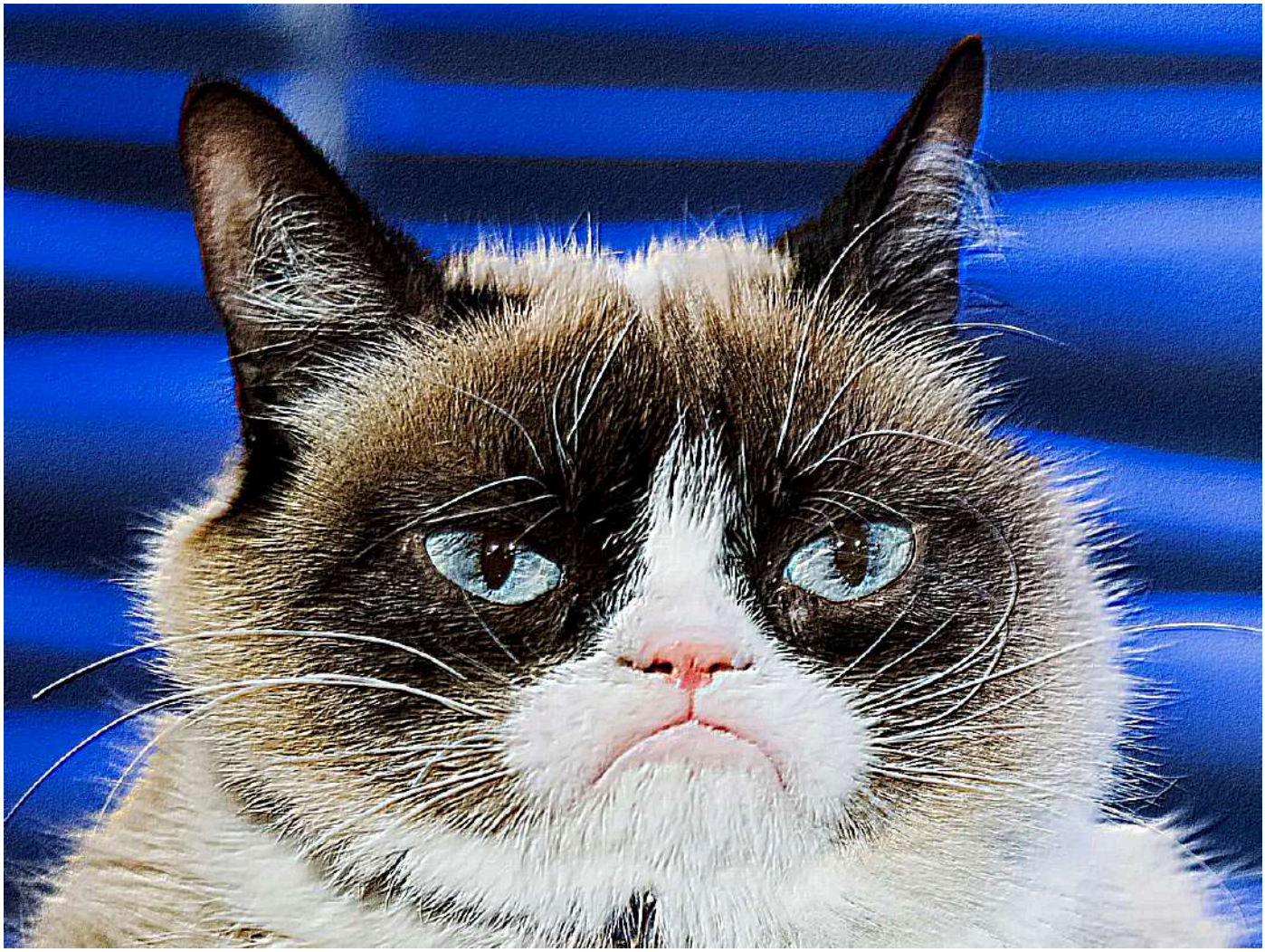
The image is now blurrier compared to the original, softening the details while still focusing on the cat's eyes, whiskers, and fur. This effect creates a smoother appearance, which can be useful for reducing noise or achieving a softer visual aesthetic.

Our sharpening, edge detection, and blur filters use kernels that are highly symmetric. Sometimes, though, kernels with less symmetry produce an interesting effect. Let's consider a kernel that blurs on one side (with positive weights) and sharpens on the other (with negative weights). It will produce a ridged or embossed effect.

```
class EmbossFilter(VConvolutionFilter):
  """An emboss filter with a 1-pixel radius."""

  def __init__(self):
    kernel = np.array([[-2, -1, 0],
                       [-1, 1, 1],
                       [0, 1, 2]])
    VConvolutionFilter.__init__(self, kernel)
```

Run this class and demonstrate the output. Provide an analysis

```
Emboss_filter = EmbossFilter()
Emboss_copy = src.copy()
Emboss_filter.apply(src, Emboss_copy)

cv2_imshow(Emboss_copy)
```



The cat's features now appear raised due to the suppression of certain details, such as the black marks around its eyes. This effect enhances the contours, giving the image a sculpted or engraved appearance.

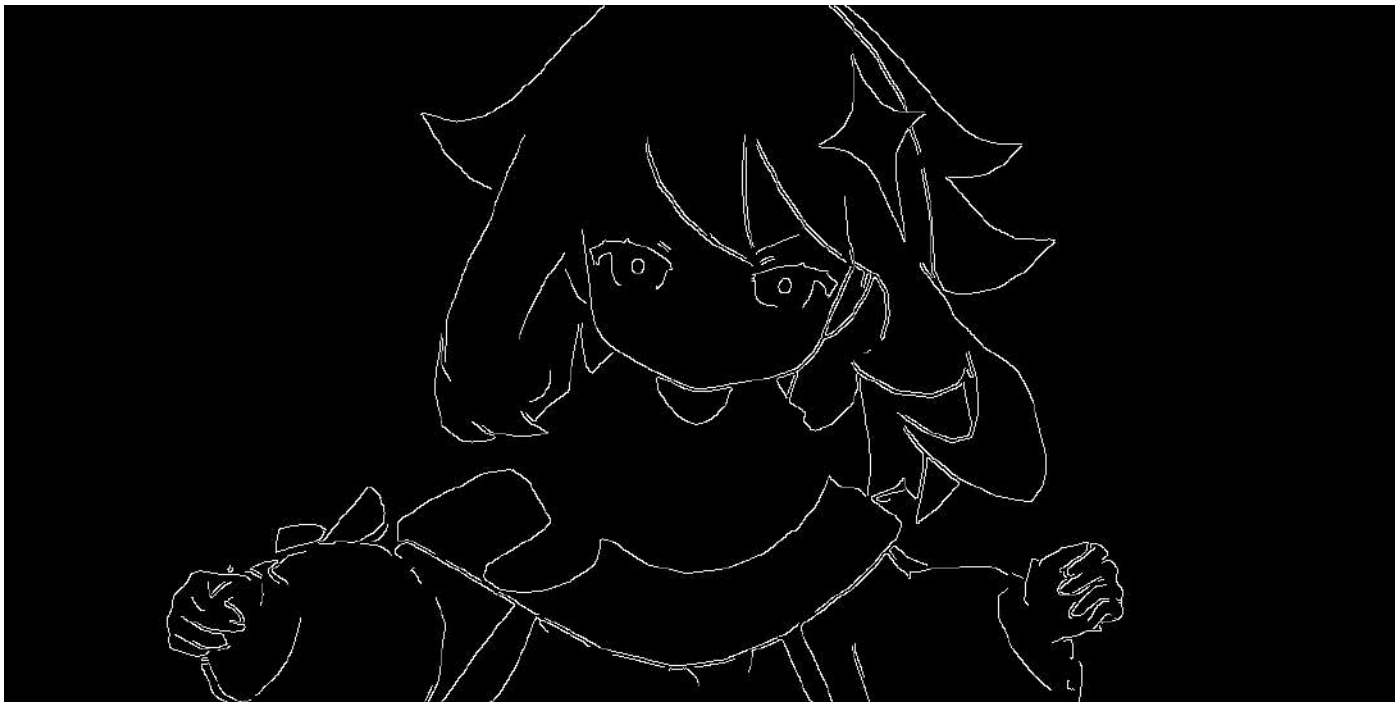
Edge Detection with Canny

OpenCV also offers a very handy function called Canny (after the algorithm's inventor, John F. Canny), which is very popular not only because of its effectiveness, but also the simplicity of its implementation in an OpenCV program, as it is a one-liner:

```
# import cv2
# import numpy as np
# from google.colab.patches import cv2_imshow

img = cv2.imread('/content/drive/MyDrive/CPE 313/Images/Paimon.png', 0)
cv2.imwrite("canny.jpg", cv2.Canny(img, 200, 300))
```

```
True
```

```
img_canny = cv2.imread('/content/canny.jpg')
cv2_imshow(img_canny)
```



The Canny edge detection algorithm is quite complex but also interesting: it's a five-step process that denoises the image with a Gaussian filter, calculates gradients, applies non maximum suppression (NMS) on edges, a double threshold on all the detected edges to eliminate false positives, and, lastly, analyzes all the edges and their connection to each other to keep the real edges and discard the weak ones.

Try it on your own image, do you agree that it's an effective edge detection algorithm? Demonstrate your samples before making a conclusion

```
SS = cv2.imread('/content/drive/MyDrive/CPE 313/Images/sunny n nephis vogue.png', 0)
cv2.imwrite("Vouge-SS_canny.jpg", cv2.Canny(SS, 200, 300))
```

```
True
```

```
SS_canny = cv2.imread('/content/Vouge-SS_canny.jpg')
cv2_imshow(SS_canny)
cv2_imshow(SS)
```

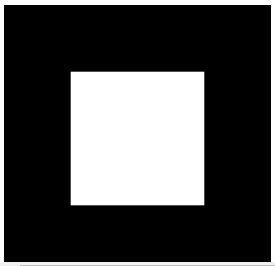
After testing it with my image and comparing the edge detection algorithm with the original image, I can say that it is indeed effective, as it accurately outlined the image I provided.

Contour Detection

Another vital task in computer vision is contour detection, not only because of the obvious aspect of detecting contours of subjects contained in an image or video frame, but because of the derivative operations connected with identifying contours.

These operations are, namely, computing bounding polygons, approximating shapes, and generally calculating regions of interest, which considerably simplify interaction with image data because a rectangular region with NumPy is easily defined with an array slice. We will be using this technique a lot when exploring the concept of object detection (including faces) and object tracking.

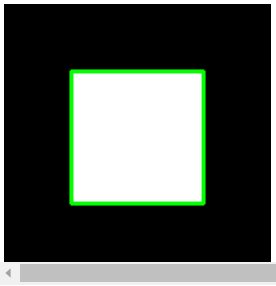
```
# import numpy as np
# import cv2
# from google.colab.patches import cv2_imshow

img = np.zeros((200,200), dtype=np.uint8)
img[50:150, 50:150] = 255


cv2_imshow(img)
```



```
ret, thresh = cv2.threshold(img, 127, 255, 0)
print(ret)
print(thresh)
```

```
127.0
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

```
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
color = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)

img = cv2.drawContours(color, contours, -1, (0, 255, 0), 2)
cv2_imshow(color)
```



What is indicated by the green box in the middle? Why are there no other green lines? Provided an analysis

The green box represents the contour of the square, outlining its external boundaries. There are no other green lines present since there are no additional figures to detect. This demonstrates how contour detection isolates distinct shapes by identifying their edges and contrasts.

Contours - bounding box, minimum area rectangle, and minimum enclosing circle

Finding the contours of a square is a simple task; irregular, skewed, and rotated shapes bring the best out of the cv2.findContours utility function of OpenCV. Consider this sample image:

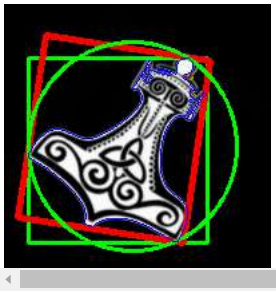
In a real-life application, we would be most interested in determining the bounding box of the subject, its minimum enclosing rectangle, and its circle. The cv2.findContours function in conjunction with a few other OpenCV utilities makes this very easy to accomplish:

```
# import cv2
# import numpy as np

img = cv2.pyrDown(cv2.imread("/content/drive/MyDrive/CPE 313/Images/Hammer contours.jpg", cv2.IMREAD_UNCHANGED))

ret, thresh = cv2.threshold(cv2.cvtColor(img.copy(), cv2.COLOR_BGR2GRAY), 127, 255, cv2.THRESH_BINARY)

contours, hier = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

for c in contours:
  # Find the bounding box coordinates
```

```
  x, y, w, h = cv2.boundingRect(c)
  cv2.rectangle(img, (x,y), (x+w, y+h), (0, 255, 0), 2)

  # Find minimum area
  rect = cv2.minAreaRect(c)

  # Calculate coordinates of the minimum area rectangle
  box = cv2.boxPoints(rect)

  # Normalize coordinates to integers
  box = np.int0(box)

  # Draw contours
  cv2.drawContours(img, [box], 0, (0, 0, 255), 3)

  # Calculate center and radius of minimu enclosing circle
  (x, y), radius = cv2.minEnclosingCircle(c)

  # Cast to integers
  center = (int(x), int(y))
  radius = int(radius)

  # Draw the circle
  img = cv2.circle(img, center, radius, (0, 255, 0), 2)
```

```
<ipython-input-34-935b8ec75673>:13: DeprecationWarning: `np.int0` is a deprecated alias for `np.intp`.  (Deprecated NumPy 1.24)
  box = np.int0(box)
```

```
cv2.drawContours(img, contours, -1, (255, 0, 0), 1)
cv2_imshow(img)
```



Explain what has happened so far.

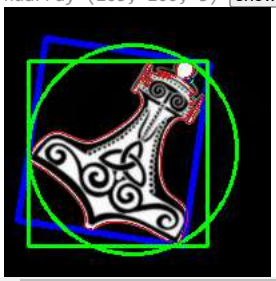
The code detects the hammer's contours, highlighting its edges in blue. A green bounding box encloses its shape, while a green circle represents the smallest enclosing boundary. A slightly tilted red rectangle provides a better-fitted bounding shape. The output effectively demonstrates contour-based shape detection using bounding boxes, minimum area rectangles, and enclosing circles.

```
x, y, w, h = cv2.boundingRect(c)
```

This is a pretty straightforward conversion of contour information to the (x, y) coordinates, plus the height and width of the rectangle. Drawing this rectangle is an easy task and can be done using this code:

```
cv2.rectangle(img, (x,y), (x+w, y+h), (0, 255, 0), 2)
```

```
ndarray (205, 200, 3) show data
```



Secondly, let's calculate the minimum area enclosing the subject:

```
rect = cv2.minAreaRect(c)
box = cv2.boxPoints(rect)
box = np.int0(box)
```

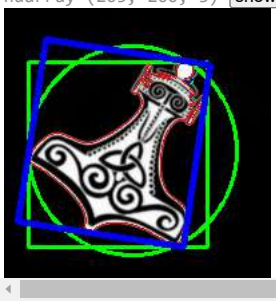
```
<ipython-input-38-ceb1eab3a452>:3: DeprecationWarning: `np.int0` is a deprecated alias for `np.intp`.  (Deprecated NumPy 1.24)
  box = np.int0(box)
```

The mechanism used here is particularly interesting: OpenCV does not have a function to calculate the coordinates of the minimum rectangle vertexes directly from the contour information. Instead, we calculate the minimum rectangle area, and then calculate the vertexes of this rectangle.

Note that the calculated vertexes are floats, but pixels are accessed with integers (you can't access a "portion" of a pixel), so we need to operate this conversion. Next, we draw the box, which gives us the perfect opportunity to introduce the cv2.drawContours function:

```
cv2.drawContours(img, [box], 0, (0, 0, 255), 3)
```

```
ndarray (205, 200, 3) show data
```



Firstly, this function—like all drawing functions—modifies the original image. Secondly, it takes an array of contours in its second parameter, so you can draw a number of contours in a single operation. Therefore, if you have a single set of points representing a contour polygon, you need to wrap these points into an array, exactly like we did with our box in the preceding example. The third parameter of this function specifies the index of the contours array that we want to draw: a value of -1 will draw all contours; otherwise, a contour at the specified index in the contours array (the second parameter) will be drawn.

Most drawing functions take the color of the drawing and its thickness as the last two parameters.

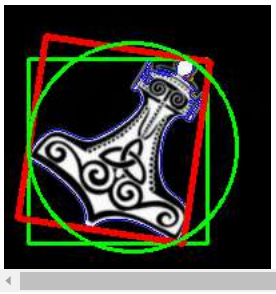
The last bounding contour we're going to examine is the minimum enclosing circle:

```
(x, y), radius = cv2.minEnclosingCircle(c)
center = (int(x), int(y))
radius = int(radius)
img = cv2.circle(img, center, radius, (0, 255, 0), 2)
```

The only peculiarity of the cv2.minEnclosingCircle function is that it returns a two-element tuple, of which the first element is a tuple itself, representing the coordinates of the circle's center, and the second element is the radius of this circle. After converting all these values to integers, drawing the circle is quite a trivial operation.

Show the final image output and provide an analysis. Does it look similar to the image shown below?

```
cv2_imshow(img)
```



Yes, the output is similar to the image shown.

4. Supplementary Activity

For this section of the activity, you must have your favorite fictional character's picture ready.

Perform/Answer the following:

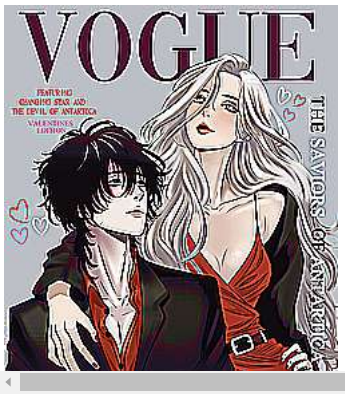
- Run all classes above meant to filter an image to your favorite character.
- Use edge detection and contour detection on your fave character. Do they indicate the same?
- Modify your character's picture such that bounding boxes similar to what happens in the last procedure will be written on the image.
- Research on the benefits of using canny and contour detection. What happens to the image after edge detection? What happens when you apply contour straight after?

```
import cv2
from google.colab.patches import cv2_imshow

SS = cv2.imread('/content/drive/MyDrive/CPE 313/Images/sunny n nephis vogue.png')
cv2_imshow(SS)
```

Running all the filtering of an image

Sharpen Filter

```
SS_copy_sharp = SS.copy()

SS_sharpen_filter = SharpenFilter()
SS_sharpened_filter = sharpen_filter.apply(SS, SS_copy_sharp)

cv2_imshow(SS_copy_sharp)
```



Find Edges Filter

```
SS_filter = FindEdgesFilter()
SS_copy = SS.copy()
SS_filter.apply(SS, SS_copy)

cv2_imshow(SS_copy)
```

Blur Filter

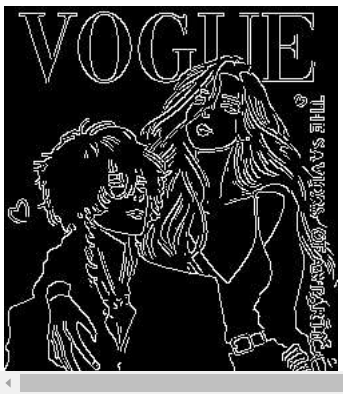
```
SS_copy_blur = SS.copy()

SS_blur_filter = Blurfilter()
SS_blur_filter.apply(SS, SS_copy_blur)

cv2_imshow(SS_copy_blur)
```

Emboss Filter

```
SS_filter = EmbossFilter()
SS_copy = SS.copy()
SS_filter.apply(SS, SS_copy)

cv2_imshow(SS_copy)
```

Using Edge Detection and Contour Detection

```
# Edge Detection With Canny
SS = cv2.imread('/content/drive/MyDrive/CPE 313/Images/sunny n nephis vogue.png', 0)
cv2.imwrite("Vouge-SS_canny.jpg", cv2.Canny(SS, 200, 300))

SS_canny = cv2.imread('/content/Vouge-SS_canny.jpg')
cv2_imshow(SS_canny)
```

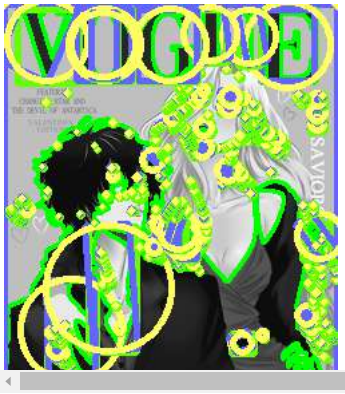



```
## Convert to grayscale
SS_gray = cv2.cvtColor(SS, cv2.COLOR_BGR2GRAY) if len(SS.shape) == 3 else SS
ret, thresh = cv2.threshold(SS_gray, 127, 255, 0)
SS_contours, SS_hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
SS_color = cv2.cvtColor(SS_gray, cv2.COLOR_GRAY2BGR)
cv2.drawContours(SS_color, SS_contours, -1, (0, 255, 0), 2)


# Display the result
cv2_imshow(SS_color)
```

No, they do not indicate the same. Edge detection captures fine details and highlights all areas of contrast, including textures and gradients, while contour detection focuses on distinct shapes with closed boundaries. In my case, edge detection provided a more detailed outline of the characters, including smaller features like hair strands and text edges. Contour detection, however, simplified the image by emphasizing the main structures and ignoring some finer details

Modifying character's picture

```
for c in SS_contours:
    # Bounding Box (Green, thin line)
    x, y, w, h = cv2.boundingRect(c)
    cv2.rectangle(SS_color, (x, y), (x + w, y + h), (0, 255, 100), 3)

    # Minimum Area Rectangle (Red, thin line)
    rect = cv2.minAreaRect(c)
    box = cv2.boxPoints(rect)

    # Normalize coordinates to integers
    box = np.intp(box)

    # Draw contours
    cv2.drawContours(SS_color, [box], 0, (255, 100, 100), 3)

    # Minimum Enclosing Circle (Blue, thin line)
    (x, y), radius = cv2.minEnclosingCircle(c)

    # Cast to integers
    center = (int(x), int(y))
    radius = int(radius)
    cv2.circle(SS_color, center, radius, (100, 255, 255), 3)

# Draw the circle
cv2_imshow(SS_color)
```



Research on the benefits of using canny and contour detection. What happens to the image after edge detection? What happens when you apply contour straight after?

Using Canny edge detection and contour detection together improves image analysis by first detecting strong edges and then refining them into meaningful shapes. Canny edge detection highlights significant brightness transitions, isolating key structures while removing unnecessary textures. After edge detection, contour detection simplifies these outlines by forming distinct, connected boundaries around objects. In my case, the Canny edge-detected image preserved fine details like hair and text, while contour detection refined these edges into clearer, enclosed shapes, making the image more structured and less noisy.

Using Canny edge detection benefits image analysis as it reduces noise and detects strong edges with high accuracy. Contour detection, on the other hand, detects borders and easily localizes them in an image. After edge detection, contour detection simplifies these outlines by forming distinct, connected boundaries around objects. In my case, the Canny edge-detected image preserved fine details like hair and text, while contour detection refined these edges into clearer, enclosed shapes, making the image more structured and less noisy.

References:

https://www.geeksforgeeks.org/sobel-edge-detection-vs-canny-edge-detection-in-computer-vision/

https://learnopencv.com/contour-detection-using-opencv-python-c/#:~:text=Using%20contour%20detection%2C%20we%20can,image%20segmentation%2C%20detection%20and%20recognition.

5. Summary, Conclusions and Lessons Learned

In this activity, I learned to manipulate images using OpenCV, applying Canny edge detection and contour detection to highlight object boundaries and shapes. Using the Canny edge detector and findContours(), I outlined key features in an image of Grumpy Cat, demonstrating how edge detection isolates strong transitions while contour detection refines them into structured shapes. In the supplementary activity, I applied the same techniques to an image of SS, further improving edge clarity and contour visualization. This activity reinforced how these methods work together to enhance object recognition and image analysis.
