Technological Institute of the Philippines	**Quezon City - Computer Engineering**
Course Code:	CPE 313
Code Title:	Advanced Machine Learning and Deep Learning
2nd Semester	AY 2024-2025
ACTIVITY NO. 4	**Edge and Contour Detection**
Name	Apuyan, Viktor Angelo
Section	CPE32S3
Date Performed:	February 20, 2025
Date Submitted:	February 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

In [ ]:
```
import numpy as np
import cv2

# This is the file URL: https://media.wired.com/photos/5cdefb92b86e041493d389df/4:3
src = cv2.imread('/content/grumpycat.webp')
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:

In [ ]:
```
from google.colab.patches import cv2_imshow

cv2_imshow(src)
```

In [ ]:
```
cv2_imshow(dst)
```

In [ ]:
```
def strokeEdges(src, dst, blurKsize = 7, edgeKsize = 5):
  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

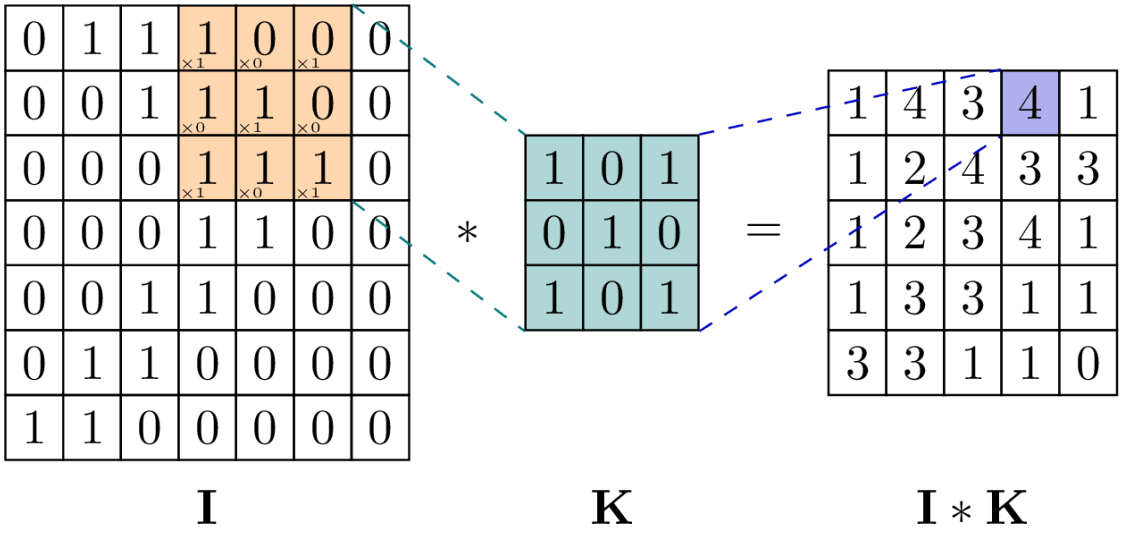
In [ ]:
```
new_img = strokeEdges(src, dst)
cv2_imshow(new_img)
```

When you make the blurKsize higher, the image will appear more blurred, which is good for erasing noise but it happens at the expense of the sharpness of the edges. On the flip side, decreasing the blurKsize gives you a chance to keep some of the details but at the expense of noise increasing. For edgeKsize, increasing it means that the edge detection is more sensitive, so it captures more details and edges while decreasing it results in not finding all edges and thereby the case being that, not all finer details are being detected. The kernel sizes are the main parts in order to control the noise of the picture, the sharpness of the edges, and the sensitivity of the edges. Thus, image processing can be designed to meet

your particular requirements based on these kernel sizes and choosing an appropriate balance among noise reduction, edge sharpness, and edge sensitivity.

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.

In [ ]:
```
# 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.

In [ ]:
```
# Applying the Kernel

# Reloading src and dst
src = cv2.imread('grumpycat.webp')
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.

The Convolution function filter2D() applies a kernel to an input image such as the source (src) using the kernel provided. In the current case, the kernel is developed to sharpen the image. The kernel weights place the central pixel (with a weight of 9) and remove the values of its neighbors (with -1 weights). This operation transforms more details into the image and improves edges, which is why the image looks sharper. The -1 in cv2.filter2D(src, -1, kernel, dst) indicates that the source image and the output image will have the same depth. The dst parameter here is not used correctly; it should not be used when specifying the depth of the output image.

In [ ]:
```
cv2_imshow(new_img)
cv2_imshow(new_img2)
```

Analysis:

The sharpened image (new_img2) will show more defined edges and details compared to the original or previous image (new_img). The sharpening kernel increases the contrast around edges, making them appear more prominent

---

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.

In [ ]:
```
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)
```

In [ ]:
```
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.

In [ ]:
```
# Load the image
src = cv2.imread('grumpycat.webp')
if src is None:
    print("Error: Could not open or read the image.")
else:
    # Create a copy of the source image for the sharpened image
    dst = src.copy()

    # Create instances of the filters
    sharpen_filter = SharpenFilter()

    # Apply the sharpen filter
    sharpen_filter.apply(src, dst)


    # Display the original and sharpened images
    cv2_imshow(src)
    cv2_imshow(dst)
```

Analysis:

- The SharpenFilter sharpens edges and details of the image by maximizing contrast between adjacent pixels. The resulting image (dst) will be sharper than the source image (src), with more defined edges and finer details. The contrast between the two will be most visible in regions of gradual color change or intensity.

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.

In [ ]:
```
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

In [ ]:
```
# Load the image
src = cv2.imread('grumpycat.webp')
if src is None:
    print("Error: Could not open or read the image.")
else:
    # Create copies of the source image for the filtered images
    dst_edges = src.copy()

    # Create instances of the filters
    edges_filter = FindEdgesFilter()

    # Apply the filters
    edges_filter.apply(src, dst_edges)

    # Display the original and filtered images
    cv2_imshow(src)
    cv2_imshow(dst_edges)
```

Anaylysis:

- The FindEdgesFilter employs a kernel that emphasizes edges in the image. The kernel weights are tuned to identify pixel intensity changes. The center pixel is assigned a weight of 8, with its neighbors being assigned a weight of -1. When convolved over the image, regions of high intensity changes (edges) will result in high values in the output image, and they will appear bright. On the other hand, regions of constant intensity will produce closer-to-zero values, which are darker. Overall, the result is to boost and accentuate the edges so that they stand out more in comparison to smoother areas. This output versus the original image will be most noticeably different in the areas with color or brightness abrupt changes, for these will equate to light areas in the edges-detected image.

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

In [ ]:
```
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

In [ ]:
```
# Load the image
src = cv2.imread('grumpycat.webp')
if src is None:
    print("Error: Could not open or read the image.")
else:
    # Create copies of the source image for the filtered images
    blur_pic = src.copy()

    # Create instances of the filters
    blurfilter = Blurfilter()

    # Apply the filters
    blurfilter.apply(src, blur_pic)

    # Display the original and filtered images
    cv2_imshow(src)
    cv2_imshow(blur_pic)
```

Analysis:

- The Blurfilter is averaging the pixel values inside a 5x5 neighborhood of each pixel. This leads to a loss of detail and noise in the image. The resulting image (dst_blur) will be smoother compared to the original image (src). Fine details and sharp edges will be less vivid, while areas of the same color will look more uniform. The blur effect is determined by the size of the kernel as well as the weights. Here, the kernel is a 5x5 matrix with equal positive weights adding up to 1. Bigger kernels tend to create a stronger blurring effect.

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.

In [ ]:
```
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

In [ ]:
```
# Load the image
src = cv2.imread('grumpycat.webp')
if src is None:
    print("Error: Could not open or read the image.")
```

```
else:
    # Create copies of the source image for the filtered images
    emboss_pic = src.copy()

    # Create instances of the filters
    embossfilter = EmbossFilter()

    # Apply the filters
    embossfilter.apply(src, emboss_pic)

    # Display the original and filtered images
    cv2_imshow(src)
    cv2_imshow(emboss_pic)
```

Analysis:

- The EmbossFilter uses a kernel that produces an embossed look simulation. The kernel utilized makes neighboring pixels compare intensity differences between them, causing an illusion of texture or depth. Negative weights on half of the kernel and positive on the other reproduce light and shade, making the image appear in 3D. Regions of the image where intensity varies gradually will be comparatively smooth, but steep changes (edges) will manifest as highlights or darks, adding to the embossed appearance. What is produced is an image that resembles being physically embossed or sculpted. The effect will be most evident along edges and at brightness or color variations.

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:

In [ ]:
```
# import cv2
# import numpy as np
# from google.colab.patches import cv2_imshow

img = cv2.imread('/content/spidey.jpg', 0)
cv2.imwrite("canny.jpg", cv2.Canny(img, 200, 300))
```

In [ ]:
```
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

In [ ]:
```
#Demonstration
img = cv2.imread('/content/swinging.jpg', 0)
cv2.imwrite("canny.jpg", cv2.Canny(img, 200, 300))

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

In [ ]:
```
img = cv2.imread('/content/finalswing.jpg', 0)
cv2.imwrite("canny.jpg", cv2.Canny(img, 200, 300))

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

Conclusion:

Yes, the Canny edge detection algorithm, as used in the code provided, is by far the most commonly used, and for a very good reason. It can produce a broad range of associated edges in pictures while virtually canceling out the noise and fakes. Nevertheless, can be the fact the first picture failed to get all the edges accurately visible and maybe its resolution resulted in that, in contrast to the bottom picture where I used the 1920x1080 image, the reason. I am not sure the image resolution will help to get better detection of the edge or not, but it is reasonable to think that the high-quality images like the 1920X1080 image used in the bottom picture might provide the algorithm with better information and thus better edge detection results could be possible.

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.

In [ ]:
```
# 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
```

In [ ]:
```
cv2_imshow(img)
```

In [ ]:
```
ret, thresh = cv2.threshold(img, 127, 255, 0)
print(ret)
print(thresh)
```

In [ ]:
```
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMP
color = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
```

In [ ]:
```
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 bounding box in the center of the image marks the bounding rectangle that is drawn around a detected contour by using cv2.rectangle() function. The box marks the region containing a particular contour with coordinates of the box being defined by the location and size of the contour. The reason for the appearance of only one green box is probably because of the contour filtering process, where the code picks the most significant contour—usually the largest or most important—to visualize. This pick might be due to the edge detection or thresholding process that might have detected only one dominant contour in the image. Also, there may be criteria in the loop that exclude smaller or less significant contours, resulting in the lack of other green lines. Other contours may also be plotted with other colors, leaving the green box to only target the main feature the code is trying to emphasize.

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:

In [ ]:
```
# import cv2
# import numpy as np

img = cv2.pyrDown(cv2.imread("/content/hammer.png", cv2.IMREAD_UNCHANGED))

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

In [ ]:
```
contours, hier = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPL
```

In [ ]:
```
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)
```

In [ ]:
```
cv2.drawContours(img, contours, -1, (255, 0, 0), 1)
cv2_imshow(img)
```

Explain what has happened so far.

In [ ]:
```
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:

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

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

In [ ]:
```
rect = cv2.minAreaRect(c)
box = cv2.boxPoints(rect)
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:

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

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:

In [ ]:
```
(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?

In [ ]:
```
cv2_imshow(img)
```

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?

Use edge detection and contour detection on your fave character. Do they indicate the same?

Edge Detection

In [ ]:
```
# Load the image
src = cv2.imread('/content/spiderman.png')
if src is None:
    print("Error: Could not open or read the image.")
else:
    # Create copies of the source image for the filtered images
    emboss_pic = src.copy()
    sharpen_pic = src.copy()
    blur_pic = src.copy()
    findedge_pic = src.copy()

    # Create instances of the filters
    embossfilter = EmbossFilter()
    sharpenfilter = SharpenFilter()
    blurfilter = Blurfilter()
    findedges = FindEdgesFilter()

    # Apply the filters
    embossfilter.apply(src, emboss_pic)
    sharpenfilter.apply(src, sharpen_pic)
    blurfilter.apply(src, blur_pic)
    findedges.apply(src, findedge_pic)

    # Display the original and filtered images
    cv2_imshow(src)
    cv2_imshow(sharpen_pic)
    cv2_imshow(findedge_pic)
    cv2_imshow(blur_pic)
    cv2_imshow(emboss_pic)
```

Contour Detection

In [ ]:
```
img = cv2.imread("spiderman.png")

# Convert the image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Apply Canny edge detection
edges = cv2.Canny(gray, 50, 150)
cv2_imshow(edges)

# Find contours
contours, hierarchy = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_S

# Draw contours on a copy of the original image
img_contours = img.copy()
cv2.drawContours(img_contours, contours, -1, (0, 255, 0), 2)
cv2_imshow(img_contours)
```

In [ ]:
```
# Comparison and bounding boxes (as before)
img_boxes = img.copy()
for cnt in contours:
    x, y, w, h = cv2.boundingRect(cnt)
    cv2.rectangle(img_boxes, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

```
cv2_imshow(img_boxes)
```

Answer:

Spiderman's pictures plainly suggest how closely connected edge detection is to contour detection, although it is not exactly the same features they point out. For instance, edge detection represents his outlines and fine details, such as the borders of his garment, web pattern, and the features of his face, to more accurately detect sharp changes in intensity using the Canny algorithm. While contour detection, in contrast, is tailored to finding and outlining whole closed shapes through grouping connected pixels that are the boundaries of objects or regions. This includes, therefore, labeling the body outline of the character, his mask, and the individual parts of the clothes that are much larger and potentially more meaningful. As well, however, contour detection lies in the proper grouping of edges together with meaningful shapes in addition to the edge detection. Of the two methods, both of them bring some useful information such as structure and shapes of objects in the photo, but they only differ slightly in how to perceive them, where the most emphasis is placed on fining the least intricate shapes and delineation complete lines for contour detection.

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?

Canny serve as an extremely useful tool in edge and contour detection. Canny allows for the clear and precise identification of an object's edge, smoothing noise with a Gaussian filter, increasing strong edges by lengthening the gradient while reducing here-and-there noise with a dual threshold. Following edge detection, the picture is reduced to a black and white representation whereby the edges are white and the background black, picking out areas where there are very sharp changes in intensity while filtering out trivial details. So, after the edges had been found, contour detection traces these edges to create a continuous curve encompassing the objects, giving effective contours of shapes that segment the picture. This provides a means by which an image is simplified in object detection, shape analysis, and feature extraction, making it easier to locate and analyze objects in an image.

5. Summary, Conclusions and Lessons Learned

From this activity, I learned how to make use of the basic ideas behind edge and contour detection with OpenCV in Python. I learned how convolution is utilized and how kernels play a critical part in edge detection in images. From hands-on implementation, I understood how to apply functions from OpenCV for image processing to detect contours and edges in example images. In addition, I was able to utilize these methods in a real-life situation by conducting edge detection on the given image of the Grumpy Cat. This experiential learning cemented my comprehension of how computer vision methods are practically utilized in detecting and emphasizing significant features in images. Through the completion of this

exercise, I gained a solid appreciation of how OpenCV may be utilized for edge and contour detection, opening doors for more in-depth investigation in the areas of computer vision and image processing.
