```python
"""
CITATIONS: Worked through problems on whiteboard with Athena & Matthew.
Brainstormed, worked through equations, and pseudocoded with both of them.
Also went to more than one office hours sessions where we discussed problems
 with TAs.
Used ChatGPT to explain concepts for me, along with to help me debug.

CS131 - Computer Vision: Foundations and Applications
Project 2 Option B
Author: Donsuk Lee (donlee90@stanford.edu)
Date created: 07/2017
Last modified: 10/25/2022
Python Version: 3.5+
"""

import numpy as np

def conv(image, kernel):
    """ An implementation of convolution filter.

    This function uses element-wise multiplication and np.sum()
    to efficiently compute weighted sum of neighborhood at each
    pixel.

    Args:
        image: numpy array of shape (Hi, Wi).
        kernel: numpy array of shape (Hk, Wk).

    Returns:
        out: numpy array of shape (Hi, Wi).
    """
    Hi, Wi = image.shape
    Hk, Wk = kernel.shape
    out = np.zeros((Hi, Wi))

    # For this assignment, we will use edge values to pad the images.
    # Zero padding will make derivatives at the image boundary very big,
    # whereas we want to ignore the edges at the boundary.
    pad_width0 = Hk // 2
    pad_width1 = Wk // 2
    pad_width = ((pad_width0,pad_width0),(pad_width1,pad_width1))
    padded = np.pad(image, pad_width, mode='edge')

    for i in range(Hi):
        for j in range(Wi):
            region_of_interest = padded[i:i+Hk, j:j+Wk]
            out[i, j] = np.sum(region_of_interest * kernel)

    return out

def gaussian_kernel(size, sigma):
```

```
    """ Implementation of Gaussian Kernel.

    This function follows the gaussian kernel formula,
    and creates a kernel matrix.

    Hints:
    - Use np.pi and np.exp to compute pi and exp.

    Args:
        size: int of the size of output matrix.
        sigma: float of sigma to calculate kernel.

    Returns:
        kernel: numpy array of shape (size, size).
    """

    kernel = np.zeros((size, size))

    k = (size - 1) / 2
    coeff = 1 / (2 * np.pi * sigma**2)
    for i in range(size):
        for j in range(size):
            num = -((i - k)**2 + (j - k)**2)
            denom = 2 * sigma**2
            gauss = coeff * np.exp(num / denom)
            kernel[i][j] = gauss

    return kernel

def partial_x(img):
    """ Computes partial x-derivative of input img.

    Hints:
        - You may use the conv function in defined in this file.

    Args:
        img: numpy array of shape (H, W).
    Returns:
        out: x-derivative image.
    """

    out = None

    kernel_x = np.array([[-1, 0, 1]])
    out = conv(img, kernel_x)/2
    #print(out)

    return out

def partial_y(img):
    """ Computes partial y-derivative of input img.
```

```
    Hints:
        - You may use the conv function in defined in this file.

    Args:
        img: numpy array of shape (H, W).
    Returns:
        out: y-derivative image.
    """

    out = None

    kernel_y = np.array([[-1], [0], [1]])
    out = conv(img, kernel_y)/2
    #print(out)

    return out

def gradient(img):
    """ Returns gradient magnitude and direction of input img.

    Args:
        img: Grayscale image. Numpy array of shape (H, W).

    Returns:
        G: Magnitude of gradient at each pixel in img.
            Numpy array of shape (H, W).
        theta: Direction(in degrees, 0 <= theta < 360) of gradient
            at each pixel in img. Numpy array of shape (H, W).

    Hints:
        - Use np.sqrt and np.arctan2 to calculate square root and arctan
    """
    G = np.zeros(img.shape)
    theta = np.zeros(img.shape)

    inner_part = ((partial_x(img))**2) + ((partial_y(img))**2)
    G = np.sqrt(inner_part)

    np.arctan2(partial_x(img), partial_y(img))

    return G, theta


def non_maximum_suppression(G, theta):
    """ Performs non-maximum suppression.

    This function performs non-maximum suppression along the direction
    of gradient (theta) on the gradient magnitude image (G).

    Args:
```

```
        G: gradient magnitude image with shape of (H, W).
        theta: direction of gradients with shape of (H, W).

    Returns:
        out: non-maxima suppressed image.
    """
    H, W = G.shape
    out = np.zeros((H, W))


    theta = np.floor((theta + 22.5) / 45) * 45
    theta = (theta % 360.0).astype(np.int32)

    direction_offsets = {
        0: ((0, 1), (0, -1)),
        45: ((-1, 1), (1, -1)),
        90: ((-1, 0), (1, 0)),
        135: ((-1, -1), (1, 1)),
        180: ((0, 1), (0, -1)),
        225: ((-1, 1), (1, -1)),
        270: ((-1, 0), (1, 0)),
        315: ((-1, -1), (1, 1))
    }

    for i in range(H):
        for j in range(W):
            direction = theta[i, j] % 360
            offsets = direction_offsets.get(direction, ((0, 0), (0, 0)))

            neigh1_val = G[i + offsets[0][0], j + offsets[0][1]] if 0 <= i +
             offsets[0][0] < H and 0 <= j + offsets[0][1] < W else 0
            neigh2_val = G[i + offsets[1][0], j + offsets[1][1]] if 0 <= i +
             offsets[1][0] < H and 0 <= j + offsets[1][1] < W else 0

            if G[i, j] >= neigh1_val and G[i, j] >= neigh2_val:
                out[i, j] = G[i, j]

    return out


def double_thresholding(img, high, low):
    """
    Args:
        img: numpy array of shape (H, W) representing NMS edge response.
        high: high threshold(float) for strong edges.
        low: low threshold(float) for weak edges.

    Returns:
        strong_edges: Boolean array representing strong edges.
            Strong edeges are the pixels with the values greater than
            the higher threshold.
```

```
        weak_edges: Boolean array representing weak edges.
            Weak edges are the pixels with the values smaller or equal to the
            higher threshold and greater than the lower threshold.
    """

    strong_edges = np.zeros(img.shape, dtype=np.bool_)
    weak_edges = np.zeros(img.shape, dtype=np.bool_)

    height = img.shape[0]
    width = img.shape[1]

    for i in range(height):
        for j in range(width):
            if img[i, j] > high:
                strong_edges[i, j] = True
            if (img[i, j] > low) and (img[i, j] <= high):
                weak_edges[i, j] = True

    return strong_edges, weak_edges


def get_neighbors(y, x, H, W):
    """ Return indices of valid neighbors of (y, x).

    Return indices of all the valid neighbors of (y, x) in an array of
    shape (H, W). An index (i, j) of a valid neighbor should satisfy
    the following:
        1. i >= 0 and i < H
        2. j >= 0 and j < W
        3. (i, j) != (y, x)

    Args:
        y, x: location of the pixel.
        H, W: size of the image.
    Returns:
        neighbors: list of indices of neighboring pixels [(i, j)].
    """
    neighbors = []

    for i in (y-1, y, y+1):
        for j in (x-1, x, x+1):
            if i >= 0 and i < H and j >= 0 and j < W:
                if (i == y and j == x):
                    continue
                neighbors.append((i, j))

    return neighbors

def link_edges(strong_edges, weak_edges):
    """ Find weak edges connected to strong edges and link them.
```

```python
    Iterate over each pixel in strong_edges and perform breadth first
    search across the connected pixels in weak_edges to link them.
    Here we consider a pixel (a, b) is connected to a pixel (c, d)
    if (a, b) is one of the eight neighboring pixels of (c, d).

    Args:
        strong_edges: binary image of shape (H, W).
        weak_edges: binary image of shape (H, W).

    Returns:
        edges: numpy boolean array of shape(H, W).
    """

    H, W = strong_edges.shape
    indices = np.stack(np.nonzero(strong_edges)).T
    edges = np.zeros((H, W), dtype=np.bool_)

    # Make new instances of arguments to leave the original
    # references intact
    weak_edges = np.copy(weak_edges)
    edges = np.copy(strong_edges)

    queue = list(indices)

    while queue:
        y, x = queue.pop(0)

        neighbors = get_neighbors(y, x, H, W)

        for ny, nx in neighbors:
            cond1 = weak_edges[ny, nx]
            cond2 = edges[ny, nx]

            if cond1 == True and cond2 == False:
                edges[ny, nx] = True
                queue.append((ny, nx))

    return edges

def canny(img, kernel_size=5, sigma=1.4, high=20, low=15):
    """ Implement canny edge detector by calling functions above.

    Args:
        img: binary image of shape (H, W).
        kernel_size: int of size for kernel matrix.
        sigma: float for calculating kernel.
        high: high threshold for strong edges.
        low: low threashold for weak edges.
    Returns:
        edge: numpy array of shape(H, W).
    """
```

```
    kernel = gaussian_kernel(size=5, sigma=sigma)
    smoothed_image = conv(img, kernel)

    G, theta = gradient(smoothed_image)

    non_max_supp_img = non_maximum_suppression(G, theta)

    strong_edges, weak_edges = double_thresholding(non_max_supp_img, low, high)

    edges = link_edges(strong_edges, weak_edges)

    return edges


def hough_transform(img):
    """ Transform points in the input image into Hough space.

    Use the parameterization:
        rho = x * cos(theta) + y * sin(theta)
    to transform a point (x,y) to a sine-like function in Hough space.

    Args:
        img: binary image of shape (H, W).

    Returns:
        accumulator: numpy array of shape (m, n).
        rhos: numpy array of shape (m, ).
        thetas: numpy array of shape (n, ).
    """
    # Set rho and theta ranges
    W, H = img.shape
    diag_len = int(np.ceil(np.sqrt(W * W + H * H)))
    rhos = np.linspace(-diag_len, diag_len, diag_len * 2 + 1)
    thetas = np.deg2rad(np.arange(-90.0, 90.0))

    # Cache some reusable values
    cos_t = np.cos(thetas)
    sin_t = np.sin(thetas)
    num_thetas = len(thetas)

    # Initialize accumulator in the Hough space
    accumulator = np.zeros((2 * diag_len + 1, num_thetas), dtype=np.uint64)
    ys, xs = np.nonzero(img)

    # Transform each point (x, y) in image
    # Find rho corresponding to values in thetas
    # and increment the accumulator in the corresponding coordiate.
    for i in range(len(xs)):
        x = xs[i]
        y = ys[i]
```

```
    for theta_idx in range(num_thetas):
        first_var = x * cos_t[theta_idx]
        second_var = y * sin_t[theta_idx]
        third_var = diag_len

        rho = int(round(first_var + second_var) + third_var)

        accumulator[rho, theta_idx] += 1

return accumulator, rhos, thetas
```