Aidan Fischer 9/30/2021

HW1 Writeup I pledge my honor that I have abided by the Stevens Honor System

Source Code

```
class Orientation(Enum):
    VERT = 1
    HORIZ = 2
    #Slopes / = pos \ = neg
    DIAG_POS = 3
    DIAG_NEG = 4

# Set up the command line parser, with defaults of kangaroo.pgm, and sigma
of 1
parser = argparse.ArgumentParser(description="Edge Detection.")
parser.add_argument("--image", "-i", default="kangaroo.pgm", type=str,
required=False, help="Path to the image file to process")
parser.add_argument("--sigma", "-s", default=1, type=int, metavar="σ",
required=False, help="Sigma value for gaussian function")
args = parser.parse_args()

#Get the value of the gaussian function for a given sigma, x, and y
#Splits the calculation into a few lines to make it easier to read.
def gaussian_function(sigma, x, y):
    partial1 = 1/(2*math.pi*math.pow(sigma,2))
    partial2 = -(math.pow(x,2)+math.pow(y,2))/(2*math.pow(sigma,2))
    partial2 = math.exp(partial2)
    return partial1*partial2

#Generate the gaussian filter with the given sigma
#Offsets x and y to center the maximum value.
def gaussian_filter(sigma):
    f = np.zeros((sigma*6+1,sigma*6+1))
    for x in range(-(sigma*3),sigma*3+1):
        for y in range(-(sigma*3),sigma*3+1):
            f[x+sigma*3][y+sigma*3] = gaussian_function(sigma, x, y)
    f = f * 1 / sum(sum(f))
    return f

#Perform edge copying to pad image for filtering
def pad_image(pad_by, image):
```

```
    #Get size of image
    bounds = np.shape(image)
    #Cast pad_by, otherwise lots of complaints about wanting ints instead
of floats.
    pad_by = int(pad_by)
    #Create the padded array
    padded = np.zeros((bounds[0] + 2 * pad_by, bounds[1] + 2 * pad_by))
    #Time for some fun python array manipulation magic!
    #This is magnitudes more efficient than a loop-based approach
    #Note, access by im[row,column]
    #Step 1: Copy image into proper position in padded array
    padded[pad_by:bounds[0]+pad_by,pad_by:bounds[1]+pad_by] = image
    #Step 2: Expand corners of image in pad_by x pad_by corners in padded
array.
    padded[0:pad_by,0:pad_by] = image[0,0] # Top Left
    padded[0:pad_by,bounds[1]+pad_by:] = image[0,bounds[1]-1] # Top Right
    padded[bounds[0]+pad_by:,0:pad_by] = image[bounds[0]-1,0] # Bottom
Left
    padded[bounds[0]+pad_by:,bounds[1]+pad_by:] =
image[bounds[0]-1,bounds[1]-1] # Bottom Right
    #Step 3: Expand edges into pad_by x edge length edges of padded array.
    #The matrix -> transpose (.T) transitions on the right side of the
    #first two assignments allow for a row vector to convert to a column
vector
    #so that the expansion over to the left side works properly.
    padded[pad_by:bounds[0]+pad_by,0:pad_by] = np.matrix(image[:,0]).T #
Left edge
    padded[pad_by:bounds[0]+pad_by,bounds[1] + pad_by:bounds[1] + pad_by *
2] = np.matrix(image[:, bounds[1] - 1]).T # Right edge
    padded[0:pad_by,pad_by:bounds[1]+pad_by] = image[0,:] # Top edge
    padded[bounds[0] + pad_by:bounds[0] + pad_by * 2,
pad_by:bounds[1]+pad_by] = image[bounds[0] - 1,:] # Bottom edge
    return padded

def apply_filter(image, filter):
    filter_bounds = int((np.shape(filter)[0] - 1) / 2)
    bounds = np.shape(image)
    #Get the edge-copied image
    padded = pad_image(filter_bounds, image)
    #Create a result array
```

```
    result = np.zeros(bounds)
    #Loop over the pixels.
    for r in range(bounds[0]):
        for c in range(bounds[1]):
            #Take the dot product of the window in padded and the filter,
and place in result
            #The trickery with .flat and np.array is due to a quirk with
numpy's dotproduct
            #function, where it does matrix mult for 2d arrays for some
reason.
            result[r, c] =
np.array(padded[r:r+filter_bounds*2+1,c:c+filter_bounds*2+1].flat).dot(fil
ter.flat)
    return result

def sobel_threshold(sobeled):
    return np.array([[0 if pixel < 100 else pixel for pixel in row] for
row in sobeled])

def gradient(sobelhoriz, sobelvert):
    bounds = np.shape(sobelhoriz)
    output = np.zeros(bounds)
    orientations = np.zeros(np.shape(sobelhoriz), dtype=Orientation)
    output =
[[math.sqrt(math.pow(sobelhoriz[i,j],2)+math.pow(sobelvert[i,j],2)) for j
in range(bounds[1])] for i in range(bounds[0])]
    output = sobel_threshold(output)
    for r in range(bounds[0]):
        for c in range(bounds[1]):
            if output[r, c] == 0:
                continue
            angle = math.atan(sobelvert[r, c]/sobelhoriz[r, c])
            pi = math.pi
            #Angles chosen based on slicing a circle in 8 equal slices,
and choosing 2 opposite slices for each orientation.
            if (angle > pi / 4 - pi / 16 and angle < pi / 4 + pi / 16) or
(angle > - pi / 4 - pi / 16 and angle < - pi / 4 + pi / 16):
                orientations[r, c] = Orientation.VERT
            elif (angle > - pi / 16 and angle < pi / 16) or (angle > pi /
2 - pi / 16 or angle < - pi / 2 + pi / 16):
```

```
                orientations[r, c] = Orientation.HORIZ
            elif (angle >= pi / 8 - pi / 16 and angle <= pi / 8 + pi / 16)
or (angle >= - 3 * pi / 8 - pi / 16 and angle <= - 3 * pi / 8 + pi / 16):
                orientations[r, c] = Orientation.DIAG_POS
            else:
                orientations[r, c] = Orientation.DIAG_NEG
    return (output, orientations)

def non_maximum_suppression(image, orientations):
    bounds = np.shape(image)
    output = np.zeros(bounds)
    test = pad_image(1, image)
    # c = check, p = pixel testing, e = edge
    for r in range(bounds[0]):
        for c in range(bounds[1]):
            # - e -
            # c p c
            # - e -
            if(orientations[r, c] == Orientation.VERT):
                pixel = test[r + 1, c + 1]
                check1 = test[r + 1, c]
                check2 = test[r + 1, c + 2]
                output[r, c] = pixel if max(pixel, check1, check2) ==
pixel else 0
            # - c -
            # e p e
            # - c -
            if(orientations[r, c] == Orientation.HORIZ):
                pixel = test[r + 1, c + 1]
                check1 = test[r, c + 1]
                check2 = test[r + 2, c + 1]
                output[r, c] = pixel if max(pixel, check1, check2) ==
pixel else 0
            # c - e
            # - p -
            # e - c
            if(orientations[r, c] == Orientation.DIAG_POS):
                pixel = test[r + 1, c + 1]
                check1 = test[r, c]
                check2 = test[r + 2, c + 2]
```

```
                output[r, c] = pixel if max(pixel, check1, check2) ==
pixel else 0
            # e - c
            # - p -
            # c - e
            if(orientations[r, c] == Orientation.DIAG_NEG):
                pixel = test[r + 1, c + 1]
                check1 = test[r + 2, c]
                check2 = test[r, c + 2]
                output[r, c] = pixel if max(pixel, check1, check2) ==
pixel else 0
    return output

def main():
    image = Image.open(args.image)
    if args.sigma > 0:
        gaussian = gaussian_filter(args.sigma)
        image = np.array(image)
        filtered = apply_filter(image, gaussian)
    else:
        filtered = np.array(image)

    sobel_vert = np.array([
        [-1,0,1],
        [-2,0,2],
        [-1,0,1]
    ])

    sobel_horiz = np.array([
        [-1,-2,-1],
        [0,0,0],
        [1,2,1]
    ])

    vert = apply_filter(filtered, sobel_vert)
    horiz = apply_filter(filtered, sobel_horiz)

    sobeled, orientations = gradient(horiz,vert)

    nms = non_maximum_suppression(sobeled, orientations)
```

```
    cv2.imwrite(f"sigma{args.sigma}-{args.image}", filtered)
    cv2.imwrite(f"sigma{args.sigma}-sobel-{args.image}", sobeled)
    cv2.imwrite(f"sigma{args.sigma}-nms-{args.image}", nms)

if __name__ == "__main__":
    main()
```

Resulting Images

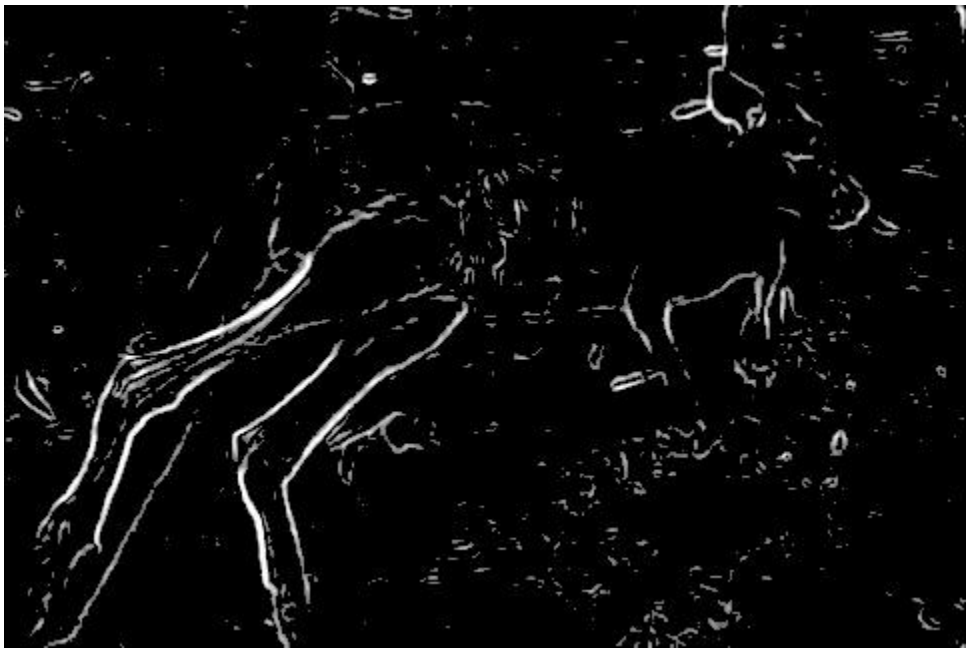
Gaussian filter

Sigma = 1

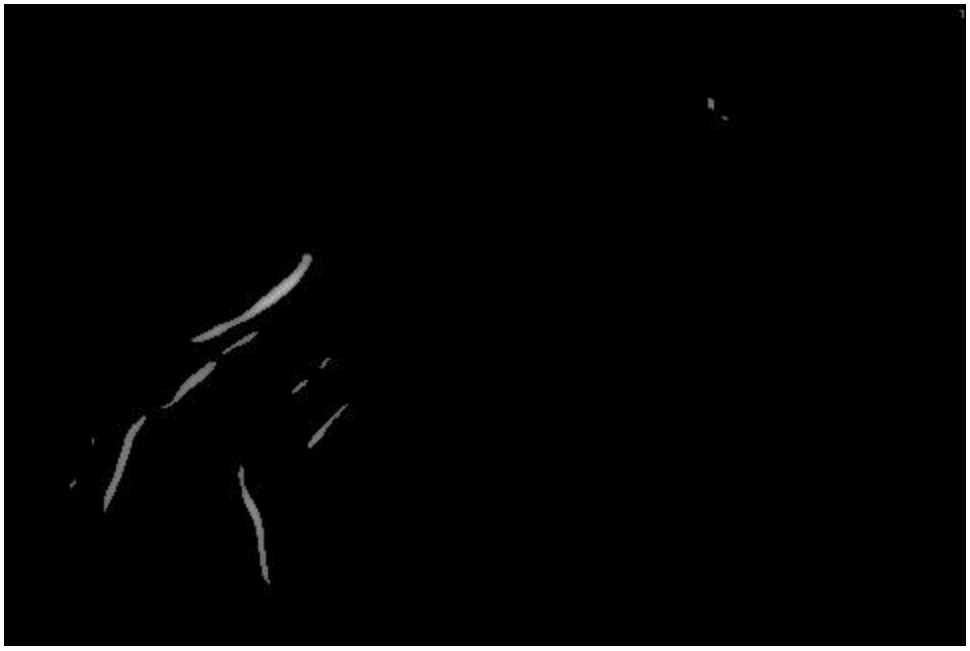
Sigma = 3

Sobel Derivative

Sigma = 1

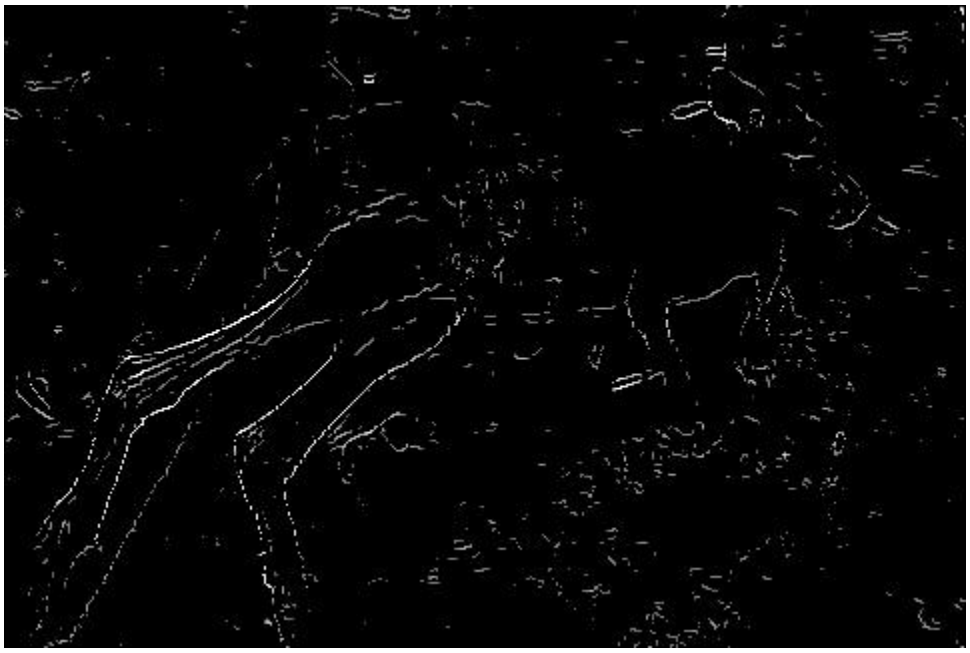


Sigma = 3

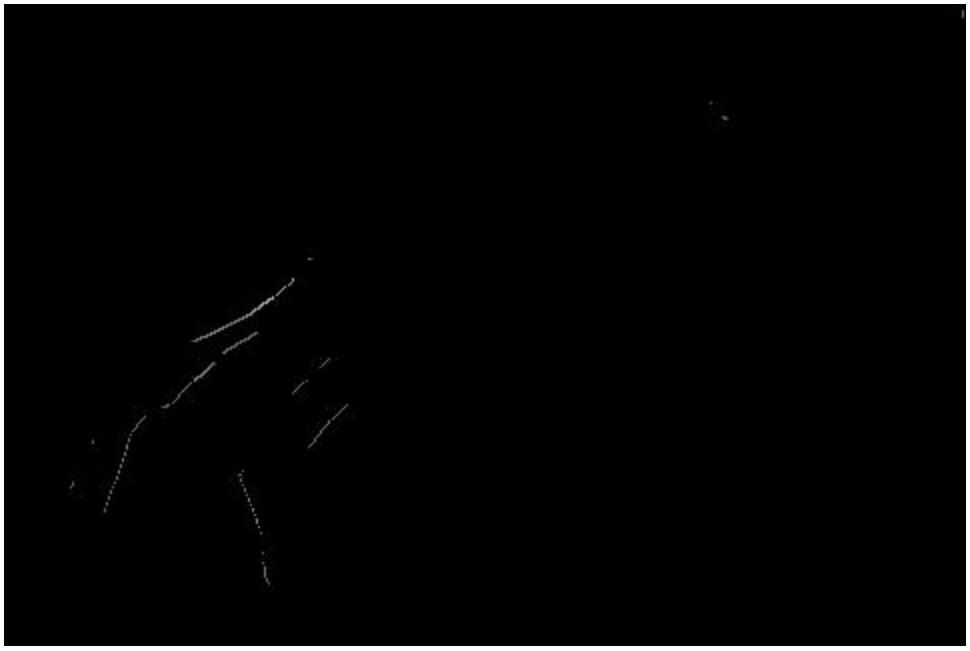


Non-maximum Suppression

Sigma = 1



Sigma = 3



What isn't obvious

1) Outputs 3 images where {x} is sigma value and {y} is image name
    a) sigma{x}_{y}.pgm - Gaussian applied
    b) sigma{x}_sobel_{y}.pgm - Previous with sobel applied
    c) sigma{x}_nms_{y}.pgm - Previous with non-maximum suppression applied
2) Install requirements with pip -r requirements.txt, however you invoke pip in your programming environment
3) Program invoked with no options defaults to kangaroo.pgm with sigma of 1
4) python3.9 hw.py -s {x} -i {y}
    a) {x} -> sigma value i.e. 2
    b) {y} -> image file i.e. kangaroo.pgm