Aidan Fischer CS558

Homework 2 – Line Detection 10/17/2021

**Code**

**main.py**

*def* main():

    image = Image.get\_image("./road.png")

    filtered = Filter.apply\_filter(image, Gaussian.gaussian\_filter(*sigma* = 1))

    hessian\_responses = Hessian.get\_responses(filtered)

    image\_ransac = Image.copy\_image(image)

    image\_ransac = RANSAC.apply\_RANSAC(hessian\_responses, image\_ransac)

    Image.save\_image(Image.to\_array(image\_ransac), "road\_RANSAC.png")

    image\_hough = Image.copy\_image(image)

    image\_hough = Hough.apply\_hough(hessian\_responses, image\_hough)

    Image.save\_image(Image.to\_array(image\_hough), "road\_Hough.png")

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Image.py**

*def* get\_image(*path*):

    image = Image.open(*path*)

    return np.array(image)

*def* save\_image(*image\_array*, *name*):

    image = Image.fromarray(*image\_array*)

    image.convert("RGB").save(*name*)

    return

*def* copy\_image(*image*):

    return np.copy(*image*)

*def* to\_array(*PILimage*):

    return np.array(*PILimage*)

**Filter.py**

SOBEL\_HORIZ = np.array([

    [-1, -2, -1],

    [0,0,0],

    [1,2,1]

])

SOBEL\_VERT = np.array([

    [-1,0,1],

    [-2,0,2],

    [-1,0,1]

])

#Filter application similar to hw1, except ignoring pixels for which pixels of filter

#are outside boundary.

*def* apply\_filter(*image*, *f*):

    im = np.copy(*image*)

    filter\_bounds = int((np.shape(*f*)[0] - 1) / 2)

    bounds = np.array(np.shape(im))

    bounds[0] -= filter\_bounds \* 2

    bounds[1] -= filter\_bounds \* 2

    #Get the edge-copied image

    #Create a result array

    result = np.zeros(bounds)

    #Loop over the pixels.

    for r in trange(bounds[0], *desc* = 'Applying filter', *leave*=False):

        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.round(np.array(im[r:r+filter\_bounds\*2+1,c:c+filter\_bounds\*2+1].flat).dot(*f*.flat))

    result2 = np.copy(result)

    return result2

# Place the filtered image into the bounds of the original image, with zeros.

*def* unbound\_image(*filtered*, *orig\_bounds*):

    bounds = np.shape(*filtered*)

    diff = (int((*orig\_bounds*[0] - bounds[0])/2), int((*orig\_bounds*[1] - bounds[1])/2))

    new = np.zeros(*orig\_bounds*)

    new[diff[0] : diff[0] + bounds[0], diff[1] : diff[1] + bounds[1]] = *filtered*

    return new

#Non-maximum suppression of a 3x3 area

*def* nms(*arr*):

    bounds = np.shape(*arr*)

    detsub = [[*arr*[x,y] if *arr*[x,y] == np.max(*arr*[x-1:x+2,y-1:y+2]) else 0 for y in range(1,bounds[1]-1)] for x in trange(1, bounds[0]-1, *desc*='Applying Non-maximum Suppresion', *leave*=False)]

    result = np.copy(*arr*)

    result[1:bounds[0]-1,1:bounds[1]-1] = detsub

    return np.array(result)

**Gaussian.py**

*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

**Hessian.py**

*def* threshold(*det*):

    return np.array([[pt if pt > 60 else 0 for pt in row] for row in *det*])

*def* get\_responses(*image*):

    # "/8"s act as normalizations.

    xx = Filter.apply\_filter(Filter.apply\_filter(*image*, Filter.SOBEL\_HORIZ/8), Filter.SOBEL\_HORIZ/8)

    xy = Filter.apply\_filter(Filter.apply\_filter(*image*, Filter.SOBEL\_HORIZ/8), Filter.SOBEL\_VERT/8)

    yy = Filter.apply\_filter(Filter.apply\_filter(*image*, Filter.SOBEL\_VERT/8), Filter.SOBEL\_VERT/8)

    det = np.multiply(xx,yy) - np.multiply(xy,xy)

    det = threshold(det)

    det = Filter.nms(det)

    return det

*def* to\_point\_array(*hessian*):

    #Convert detected points into array of point tuples.

    points = []

    for r in range(np.shape(*hessian*)[0]):

        for c in range(np.shape(*hessian*)[1]):

            if *hessian*[r,c] != 0:

                points.append((c,r))

    return np.array(points)

**RANSAC.py**

*def* total\_least\_squares(*sample*):

    #Step 1: Get UTU, given by 5th set of notes.

    x\_array = *sample*[:,0]

    y\_array = *sample*[:,1]

    x\_minus\_avg = x\_array-np.average(x\_array)

    y\_minus\_avg = y\_array-np.average(y\_array)

    utu00 = np.sum(x\_minus\_avg\*\*2)

    utu11 = np.sum(y\_minus\_avg\*\*2)

    utu01 = utu10 = np.sum(np.multiply(x\_minus\_avg,y\_minus\_avg))

    utu = np.array([

        [utu00, utu01],

        [utu10, utu11]

    ])

    #Step 2: get normal eigenvector related to smallest eigenvalue.

    values, vectors = np.linalg.eig(utu)

    min\_vector = vectors[:, np.where(values == np.min(values))[0][0]]

    #Step 3: get d

    d = min\_vector[0] \* np.average(x\_array) + min\_vector[1] \* np.average(y\_array)

    #Step 4: return a,b,d, eigenvalue (to rank fit)

    return (min\_vector[0],min\_vector[1],-d,np.min(values))

*def* get\_inliers(*points*, *fit*, *param\_t*):

    a, b, d, \_ = *fit*

    inliers = []

    for pt in *points*:

        x, y = pt

        dist = abs(a \* x + b \* y + d)/math.sqrt(a\*\*2+b\*\*2)

        if dist <= *param\_t*:

            inliers.append(pt)

    return inliers

*def* get\_N(*points*, *param\_s*, *param\_p*, *param\_t*):

    N = np.count\_nonzero(*points*)

    samples = 0

    e = 0.5

    mi = 0

    bar = tqdm(*desc* = 'Finding optimal N...', *total* = N, *leave* = False)

    while N > samples:

        sample\_points = *points*[np.random.choice(*points*.shape[0], *param\_s*, *replace*=False), :]

        fit = total\_least\_squares(np.array(sample\_points))

        inliers = get\_inliers(*points*, fit, *param\_t*)

        inlier\_ratio = len(inliers)/len(*points*)

        if inlier\_ratio > mi:

            e = 1 - inlier\_ratio

            mi = inlier\_ratio

            N = math.log(1 - *param\_p*)/math.log(1 - math.pow(1 - e, *param\_s*))

        samples += 1

        bar.total = math.ceil(N)

        bar.update(1)

    return math.ceil(N)

*def* apply\_RANSAC(*hessian*, *image*):

*hessian* = Filter.unbound\_image(*hessian*, np.shape(*image*))

    points = Hessian.to\_point\_array(*hessian*)

    points\_source = np.copy(points)

    param\_s = 2

    param\_t = 8

    param\_p = 0.999

    param\_n = 50000

    param\_d = 12

    lines = []

    im = Image.fromarray(*image*)

    draw = ImageDraw.Draw(im)

    ransac = trange(param\_n, *desc* = 'Running RANSAC... Found 0 Lines')

    for \_ in ransac:

        sample = np.array(points\_source[np.random.choice(points\_source.shape[0], param\_s, *replace*=False), :])

        fit = total\_least\_squares(sample)

        inliers = get\_inliers(points, fit, param\_t)

        inliers = np.array(inliers)

        if len(inliers) >= param\_d:

            lines.append((total\_least\_squares(inliers),inliers))

            points\_source = np.array([pt for pt in points\_source if not any([np.array\_equal(s, pt) for s in inliers])])

            ransac.set\_description(f'Running RANSAC... Found {len(lines)} Lines.')

            if(len(points\_source) < param\_d):

                print("Not enough remaining outliers to form another line. Breaking out early.")

                break

    best\_lines = sorted(lines, *key* = *lambda* *x*: *x*[0][-1])

    for i in range(min(4, len(best\_lines))):

        pts = sorted(best\_lines[i][1], *key* = *lambda* *x*: *x*[0])

        draw.line([pts[0][0],pts[0][1],pts[-1][0],pts[-1][1]], *width* = 1)

        for pt in pts:

            bounds = [pt[0] - 1, pt[1] - 1, pt[0] + 2, pt[1] + 2]

            draw.rectangle(bounds, *width* = 1)

    return im

**Hough.py**

#How much to discretize in the p direction. for example, this = 1 means p -> [0,1), [1,2), ...

#in vote accumulator

P\_DISCRETIZATION = 0.8

*def* apply\_hough(*hessian*, *image*):

    #Theta = 0-180

    #p = xcos(theta) + ysin(theta). Between cos(theta) and sin(theta) <= 1, p <= x + y

    #p = 0 - max(x+y)/PDISC = sum of bounds of image / discretization constant

*hessian* = Filter.unbound\_image(*hessian*, np.shape(*image*))

*hessian* = Hessian.to\_point\_array(*hessian*)

    #Base algorithm

    H = np.zeros((181, math.floor(np.sum(np.shape(*image*))/P\_DISCRETIZATION)))

    Hpt = [[[] for c in r] for r in H]

    for pt in tqdm(*hessian*, *desc*="Hough: Counting votes..."):

        x, y = pt

        for theta in range(181):

            t = math.radians(theta)

            p = float(x) \* math.cos(t) + float(y) \* math.sin(t)

            H[theta, math.floor(p/P\_DISCRETIZATION)] += 1

            Hpt[theta][math.floor(p/P\_DISCRETIZATION)].append(pt)

    im = Image.fromarray(*image*)

    draw = ImageDraw.Draw(im)

    #Local maxima

    maxima = Filter.nms(H)

    Hpt = np.array(Hpt, *dtype*=object)

    #Find and draw lines with most votes

    for \_ in range(4):

        #Weirdness with np.array conversions here due to how np.unravel\_index returns.

        theta, p = np.array(np.array(np.unravel\_index(np.argmax(maxima),np.shape(maxima))).flat)

        t = math.radians(theta)

        #Now generate a function we can use to get y values for specific x values so we

        #can plot the line with whatever extreme points voted for it.

*def* thisline(*x*, *pt*):

            #p = xcos(t) + ysin(t). Solve for y.

            #p - xcos(t) = ysin(t)

            #y = (p - xcos(t))/sin(t)

            if math.sin(t) == 0:

                return *pt*[1]

            return (p \* P\_DISCRETIZATION - *x* \* math.cos(t))/math.sin(t)

        #Get sorted points that voted.

        pts = sorted(Hpt[theta,p], *key* = *lambda* *x*: *x*[0])

        #Draw

        draw.line([pts[0][0], thisline(pts[0][0], pts[0]), pts[-1][0], thisline(pts[-1][0],pts[-1])])

        for pt in pts:

            bounds = [pt[0] - 1, pt[1] - 1, pt[0] + 2, pt[1] + 2]

            draw.rectangle(bounds, *width* = 1)

        # Remove this vote so it isn't double counted

        maxima[theta, p] = 0

    return im

**Resulting Images**

**RANSAC**

**A picture containing outdoor, sky, road, way

Description automatically generated**

**Hough**

A picture containing outdoor, road, apartment building

Description automatically generated

**Brief explanation**

Following the slides, RANSAC takes 2 random points a ton of times to gather a set of lines, removing inliers from the set of points to take randomly. It stops early if not enough points remain to get sufficient inliers to make a line.

Hough follows the base implementation without smoothing or strong gradient use. I tried to implement smoothing but ended up with a weird issue where it wasn’t properly populating the arrays. Loops over angle from 0-180, calculates p, then adds a vote to the proper place in the accumulator array. There is a constant called P\_DISCRETIZATION which is the discretization constant. If it is 1, then votes for each integer p rounded down. This is multiplied back out later to get the proper p for the line function. An addition I made was to accumulate the points that vote for each angle,p combination in order to simplify drawing the voting points onto the image later.