ECEN 898 Python 3

Shawn Wimer

April 2021

1 Introduction

The generalized Hough transform is a technique intended for the description of arbitrary shapes. Instead of working in on pixels in the image space the work is done in parameter space, which associates curves and their position relative to a reference point. The detection of objects in images requires first calculating the parameter space for a reference image of the object or similar object, which can be extended to the use of multiple references, then involves calculating where the reference point should be for curves in the test set in an "accumulator." A shortcoming of this simplistic approach is that it will best detect objects of the same scale and rotation, as parameter space is not inherently scale or rotation invariant.

2 Technical methodology

To overcome the transform-variance shortcoming the standard technique of creating accumulators for a variety of scales and rotations was used. Otherwise the techniques are

2.1 Pre-processing

To build a "face" model, each reference image (Figure 1) was blurred with a Gaussian kernel of $\sigma = 2$ and either sized 9x9 (references A and B) or 5x5 (references C). References A and B were blurred with a larger kernel because the lines in their faces and hair were more apparent and led to many extraneous lines and blurring reference C with that same size removed too many of the lines.

The test image (chosen from Figure 2) was blurred as well with Gaussians of $\sigma = 3$. The kernel sizes of the first two images are again 9x9, but the third image required a larger kernel in order to remove some extra lines.

2.2 Canny edge detection

The approach from the first lab was used here excepting two changes and so can largely be consulted for effects. No threshold changes are exposed to the user and were hence outside the scope of this paper. The derivatives in both direction were both calculated through kernel arithmetic but they are now calculated in tandem rather than sequentially in order to save time. The `arctan` function was replaced with `np.arctan2` in order to easily capture the entire angle space and simplify later hysteresis.

(a) Reference A (b) Reference B (c) Reference C

Figure 1: Reference images

(a) Test A (b) Test B (c) Test C

Figure 2: Test images

2.3 R-table and accumulator

The r-table was constructed in an ordinary fashion with no variability, though the magnitude of the gradient at the point is kept for later use. For each edge pixel, the gradient direction rounded to one decimal place, the vector from the reference point to the pixel, and the magnitude of the gradient are saved in a dictionary that looks like so: `(theta: (rho: magnitude))`[1]. If there are multiple instances of single direction-vector pairs, the magnitude is the sum of all of their magnitudes. This should not occur when using just one test image unless it is very large and the gradient directions of pixels in a small region far from the reference point collapse into one discretized value (e.g. a very large image with arcs of a circle centered about the reference point) but can more readily occur when using multiple reference images.

To create the accumulator, for each edge pixel in the test image the gradient direction is used to look-up the vector to the reference point using the dictionary above. A vote, weighted by the gradient magnitude value, is added to that point. A standard method is used to overcome

[1]The parentheses should be curly braces but `\lstinline` does not render the closing braces correctly here.

the transform-variance shortcoming: for each edge pixel a number of other vote locations are determined by adding a rotation and/or scaling, saving separate pages for each rotation/scaling pair. This results in a four-dimensional image and can lead to large processing times if one is not selective.

2.4 Peak detection and match determination

A large box filter of size 25x25 is cross-correlated with the accumulator space in order to combine close votes without agglomeration nor clustering. The peaks are then subjected to a threshold, though this is mainly for intelligibility when viewing verbose output, as it has no effect on our ultimate selection.

Selecting the most likely peak is simply the selection of the point with the maximum value. This will return a point and the scale and rotation of the detected object. Since the goal of this lab is to detect a single face there is no need to determine other candidate points.

3 Experimental details

Four parameters are configurable in this script: scales and rotations to check, additive noise, and the test image. The faces in the test images are all smaller than the faces in the reference images so scaling will probably be essential in accurately detecting each face. Since there are a number of scales fine control of that parameter will not be necessary. However, the faces in all images are near vertical, so rotation is unlikely to play much of a role. The additive noise is passed as a percent of the maximum of the accumulator page. The test image will play the greatest role. While the reference images are varied (older man, younger woman, young woman), two of the test images are older men and one is an older woman with all of her hair visible, something not seen in any reference image. Since no woman of her apparent age is in the reference set it should be more difficult to detect her face.

3.1 Timing and cross-correlation

In previous labs calculating the cross-correlation of kernels and images was calculated directly. In this lab, as the vote space is very large and the experience of the last lab showed that direct calculation could be very slow, `scipy.signal.correlate2d` was used in order to take advantage of cross-correlation in Fourier space. The slowest part should still be the peak detection, which involves filtering the entire vote space with a moderately large box filter. Each section was timed for comparison.

4 Results

4.1 Face model

By iterating through each offset vector in the r-table and adjusting points by the accompanying gradient magnitude, the face model can be saved as an image, shown in Figure 3. Though very subjective, it does appear to be quite "face-like," accentuating features apparently weighted as crucial, such as the sides of the head, the mouth, and the right eye, though also emphasizing some unanticipated features, such as something that is probably hair, the nostrils, and many lines in the

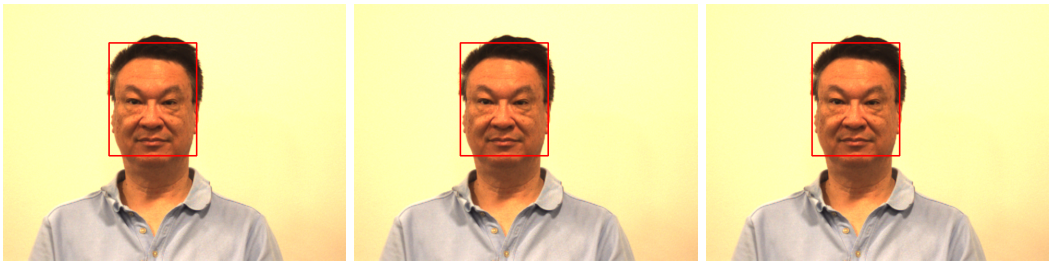
Figure 3: Face model

face. The eyes in general are weighted less than one might expect and the chin is nearly absent. The use of more reference images could shed some light on these results.

4.2 Test A

Figure 4 shows some apparently uninteresting results: three images perfectly encapsulate the face while two found his shoulder. However, the only difference is in the set of rotation angles used. In fact, the rotation found for the face is 1° for the first two results and 2.5° for the second two results, implying that the detector is very sure the rotation of the face is near 2.5°, which seems wrong, prompting a direct measurement of 1.5°, so the detector is more accurate than it seems. Accumulator noise has little effect for this image.

(a) Rotation: ±1°, noise=1% (b) Rotation: ±1°, noise=3%



(c) Rotation: ±2.5°, noise=0% (d) Rotation: ±2.5°, noise=1% (e) Rotation: ±2.5°, noise=3%

Figure 4: Results for Test A

4.3 Test B

The process seems quite robust (though slightly offset) for this image, as can be seen in Figure 5, which may be because the man seems to resemble the face model. The points are consistent within each rotation and vary only by one pixel in each direction between the two. Again, the rotation is either 1° or 2.5°, even though the face seems nearly vertical; a direct measurement is approximately 0.3°. Noise seems to have absolutely no effect. The scale is 0.5, which the minimum possible and obviously smaller than the actual face in the image.

4.4 Test C

These results (Figure 6) are less consistent, as was anticipated. Additionally, the box reveals that her face is more circular than all the other images, another difference that would make it more difficult to detect her face. At low rotation and noise the detector finds a slightly larger and slightly offset face, but it is in the correct area. Increasing noise then confounds the detector, which labels her neck. At higher rotation and no noise it labels most of her face, but with more noise it becomes inaccurate. The chosen rotations are more varied, ranging from $-0.4°$ to 2.5°; a direct measurement is near 0°.

4.5 Timing

Using `correlate2D` improved the run-time by a great amount. Instead of taking ten minutes to blur one image with a moderately sized kernel as occurred in the segmentation lab, the worst case here for blurring all reference images and one test image together was 0.25 seconds. Most time was

(a) Rotation: ±1°, noise=1% (b) Rotation: ±1°, noise=3%

(c) Rotation: ±2.5°, noise=0% (d) Rotation: ±2.5°, noise=1% (e) Rotation: ±2.5°, noise=3%

Figure 5: Results for Test B

(a) Rotation: ±1°, noise=0% (b) Rotation: ±1°, noise=1% (c) Rotation: ±1°, noise=3%

(d) Rotation: ±2.5°, noise=0% (e) Rotation: ±2.5°, noise=1% (f) Rotation: ±2.5°, noise=3%

Figure 6: Results for Test C

spent finding all peaks due to the blurring of the entire vote space; 29.2 seconds minimum and 34.1 seconds maximum. Creating the accumulator was the next most taxing task: 8.7 seconds minimum

and 10.1 seconds maximum. Building the r-table consistently took 5.1 seconds to 5.3 seconds. It is clear that the largest determiner of time taken is the size of the accumulator, which is linear in the number of edge pixels found but polynomial in the number of scales and/or rotations to be checked. If test images had a random orientation and scale rather than consistent values for each the entire process would take much longer as there is no intuitive and generalizable mechanism for stopping the process early.

5 Conclusions

It wasn't expected that the scale parameter would be very interesting but Test B revealed that there might be some hidden caveats, though each image consistently has one scale associated with it. The rotation displayed seemingly odd behavior but when the actual rotation was closely examined it was closer than at first glance. Both of these results are still slightly spurious and warrant further investigation.

Test C performed better than expected though still worse than the other tests. While the face was mostly found half of the time and partially the other half the detection seems haphazard, as it is not very consistent. It is possible that it is due to chance, as the highest noise is never consistent with no noise while the lower level of noise is consistent with either none or the higher level.

The timing analysis was correct. Without relying on a home-brewed directly-calculated cross-correlation function, most time was spent on the goal of determining the center of the face. With larger numbers of rotations or scales to check the time could grow to be much larger, as it is polynomial in both terms, but when some constraints can be assumed the time is manageable.

Taken together, these results show that the generalized Hough transform is a fairly robust technique for object detection, though not without its quirks regarding translation- and rotation-variance workarounds. A key advantage is that it would be trivial to add more reference images, which should improve the quality greatly, and would only increase the time building the r-table and later the accumulator. With a large enough number of images only the time for the r-table would increase, so long as the gradient direction is discretized (as in this lab). The greatest time is spent in filtering the vote space, which doesn't depend directly upon the number of reference images.

6 Appendix

```
#!/usr/bin/env python

import sys, getopt, os
import math
import numpy as np
import cv2
import copy
import random
from timeit import default_timer as timer
from utils import *

def main(argv):
    usage = "main.py -i <image)> -s <scales> -r <rotation> -n <noise> -v <verbosity> "
    try:
        opts, args = getopt.getopt(argv, "hi:n:s:r:v:")
    except getopt.GetoptError:
        print(usage)
        print("Use\n\tmain.py -h\nto learn how to use this and run default settings")
        sys.exit(2)

    # assume a square kernel
    kernel_size = (9,9,5)
    sigma = (2,2,2)
    kernel_size_test = (9,9,11)
    sigma_test = (3,3,3)
    image = 'test/test_img003.png'
    image_n = 3
    verbose = False
    noise = 0
    scales = list(np.linspace(0.5, 0.8, 7))
    rotations = list(np.linspace(-2.5, 2.5, num=11))

    for opt, arg in opts:
        log("{0} {1}".format(opt, arg))
        if opt == '-h':
            print(usage)
            print("Example usage (defaults): main.py -i 1 -s 0.5,0.8 -r 2.5 -n 0 -v True"
                )
            print("\tImage (str): 1, 2, 3")
            print("\tScale (float (,float)): seven scales from val1 to val2; if no val2 
                only scale at val1")
            print("\tRotation maximum (int): eleven rotations from -value to +value, 
                unless value=0")
            print("\tNoise (float): percent of maximum pixel for range of noise added to 
                the accumulator")
            print("\tVerbosity (str): True, False")
            print("No required arguments")
        elif opt == '-i':
            if "1" in arg.lower():
                image = 'test/test_img001.png'
            elif "2" in arg.lower():
                image = 'test/test_img002.png'
            elif "3" in arg.lower():
                image = 'test/test_img003.png'
            else:
                print("Use\n\tmain.py -h\nto learn how to use the image argument; 
```

```
                defaulting␣to␣3")
            continue
        image_n = int(arg)
    elif opt == '-v':
        if "false" in arg.lower() or "f" == arg.lower():
            verbose = False
        elif "true" in arg.lower() or "t" == arg.lower():
            verbose = True
        else:
            print("Use\n\tmain.py␣-h\nto␣learn␣how␣to␣use␣the␣verbosity␣argument;␣
                defaulting␣to␣False")
            verbose = False
    elif opt == '-n':
        try:
            noise = float(arg)
        except ValueError:
            print("Noise␣must␣be␣a␣number␣0-100")
            sys.exit(2)
        if noise < 0 or noise > 100:
            print("Noise␣must␣be␣a␣number␣0-100")
            sys.exit(2)
    elif opt == '-s':
        try:
            start = float(arg.rsplit(",")[0])
            end = float(arg.rsplit(",")[1])
            scales = list(np.linspace(start,end, 7))
        except ValueError:
            print("Scales␣must␣be␣floats.")
            sys.exit(2)
        except IndexError:
            print("Single-value␣scale:␣{}".format(start))
            end = start
            scales = list(np.linspace(start, start, 1))
            sys.exit(2)
    elif opt == '-r':
        try:
            val = float(arg)
        except ValueError:
            print("Rotation␣must␣be␣a␣float.")
            sys.exit(2)
        if val == 0.0:
            rotations = list(np.linspace(0, 0, num=1))
        else:
            rotations = list(np.linspace(-val, val, num=11))

# prepare file names
file_suffix = image.rsplit("/", 1)[1].rsplit(".", 1)[0] + '_s_{0:.3f},{1:.3f}_r_{2:.3
    f}_n_{3}'.format(scales[0], scales[-1], rotations[-1],noise)
logfile = 'out/' + file_suffix + '_info.txt'
open(logfile, 'w').close()
log("Scales:␣{0}\nRotations:␣{1}".format(scales, rotations), logfile)

# Force kernels to be odd
for size in kernel_size:
    if size % 2 == 0:
        size = size + 1

# load and blur reference images
```

```
start = timer()
refs = []
i = 1
for filename in os.listdir("ref/"):
    img = cv2.imread("ref/" + filename, cv2.IMREAD_GRAYSCALE)
    refs.append(blur(img, kernel_size[i], sigma[i]).round(decimals=2))
    if verbose:
        cv2.imwrite('out/' + filename + "blurred.png", refs[-1])

# load test image
img = cv2.imread(image, cv2.IMREAD_GRAYSCALE)
x = blur(img, kernel_size_test[image_n-1], sigma_test[image_n-1]).round(decimals=2)
if verbose:
    cv2.imwrite("out/{0}_blurred.png".format(file_suffix), x)
end = timer()
log("Time␣taken␣for␣loading␣and␣pre-processing␣images:␣{}".format(end-start), file=
    logfile)

# build r-table
start = timer()
table = buildRtable(refs, (refs[0].shape[0]/2,refs[0].shape[1]/2), (50,35), verbose)
end = timer()
log("Time␣taken␣to␣build␣r-table:␣{}".format(end-start), logfile)

# build accumulator
start = timer()

accum = genAccumulator(x, table, (55,40), rotations, scales, verbose)
if noise != 0.0:
    for a in accum:
        sc =  noise/100*np.amax(a)
        a += np.random.randint(-sc, high=sc, size=a.shape).astype(int)
        np.clip(a, 0, 255)
end = timer()
log("Time␣taken␣to␣build␣accumulator:␣{}".format(end-start), logfile)

if verbose:
    for s in scales:
        path = "out/{0:.3f}".format(s)
        if not os.path.exists(path):
            os.mkdir(path)
        for t in rotations:
            cv2.imwrite("out/" + str(s) + "/{0}_votes_{1:.3f}_{2:.3f}.png".format(
                file_suffix,t,s), (accum[:,:,rotations.index(t),scales.index(s)]*255/
                np.max(accum[:,:,rotations.index(t),scales.index(s)])).astype(np.
                uint8))

# find peaks
start = timer()
peaks = getPeaks(accum, np.max(accum)/2)
end = timer()
log("Time␣taken␣to␣find␣peaks:␣{}".format(end-start), logfile)

if verbose:
    for s in scales:
        path = "out/{0:.3f}".format(s)
        if not os.path.exists(path):
            os.mkdir(path)
```

```
            for t in rotations:
                cv2.imwrite("out/" + str(s) + "/{0}_peaks_{1:.3f}_{2:.3f}.png".format(
                    file_suffix,t,s), (peaks[:,:,rotations.index(t),scales.index(s)]).
                    astype(np.uint8))

    # choose the most likely peak
    start = timer()
    max = np.amax(peaks, axis=None)
    maxi = np.argmax(peaks, axis=None)
    index_max = np.unravel_index(maxi, peaks.shape)
    log("Scale␣for␣max:␣{0},␣rotation␣for␣max:␣{1}".format(scales[index_max[3]],
        rotations[index_max[2]]), logfile)
    log("Value␣of␣max:␣{0},␣position␣of␣maximum:␣{1}".format(peaks[index_max], (index_max
        [0], index_max[1])), logfile)
    end = timer()
    log("Time␣taken␣to␣choose␣most␣likely␣peak:␣{}".format(end-start), logfile)

    if verbose:
        cv2.imwrite("out/peaks_{0}.png".format(file_suffix), (peaks[:,:,index_max[2],
            index_max[3]]).astype(np.uint8))

    # save the final image
    img_BGR = cv2.imread(image, cv2.IMREAD_COLOR)
    box_size = (360,280)
    result = displayResult(img_BGR, (index_max[0], index_max[1]), box_size, rotations[
        index_max[2]], scales[index_max[3]])
    green_dot = copy.deepcopy(img_BGR)
    cv2.imwrite("out/{0}_result.png".format(file_suffix), green_dot)

    # save the "face" model
    face = np.zeros(refs[0].shape)
    ref_point = np.array((refs[0].shape[0]/2,refs[0].shape[1]/2))
    for theta in table:
        for rho in table[theta]:
            edge = ref_point + np.array(rho)
            face[edge.astype(int)[0],edge.astype(int)[1]] += table[theta][rho]
    face = (face*255/np.amax(face)).astype(int)
    cv2.imwrite("out/face_model.png", face)


if __name__ == "__main__":
    main(sys.argv[1:])

import sys, os
import math
import numpy as np
import copy
import cv2
from scipy.signal import correlate2d


def log(message, file=None):
    if not file:
        print(message)
    else:
        with open(file, 'a') as f:
            f.write(message + '\n')
```

```
def pad_array(img, amount, method='replication'):
    if amount < 1:
        return copy.deepcopy(img)
    re_img = np.zeros([img.shape[0]+2*amount, img.shape[1]+2*amount])
    re_img[amount:img.shape[0]+amount, amount:img.shape[1]+amount] = img
    if method == 'zero':
        pass # already that way
    elif method == 'replication':
        re_img[0:amount,amount:img.shape[1]+amount] = np.flip(img[0:amount, :], axis=0) #
             top
        re_img[-1*amount:, amount:img.shape[1]+amount] = np.flip(img[-2*amount:-amount,
            :], axis=0) # bottom
        re_img[:, 0:amount] = np.flip(re_img[:, amount:2*amount], axis=1) # left
        re_img[:, -1*amount:] = np.flip(re_img[:, -2*amount:-amount], axis=1) # right

    return re_img


def Gaussian2D(size, sigma):
    # simplest case is where there is no Gaussian
    if size==1 or sigma==0:
        return np.array([[0,0,0],[0,1,0],[0,0,0]])

    # parameters
    peak = 1/2/np.pi/sigma**2
    width = -2*sigma**2

    # Gaussian filter
    H = np.zeros([size, size])

    # populate the Gaussian
    if size % 2 == 1:
        k = (size - 1)/2
        for i in range(1, size+1):
            i_part = (i-(k+1))**2
            for j in range(1, size+1):
                H[i-1, j-1] = peak*math.exp((i_part + (j-(k+1))**2)/width)
    else:
        k = size / 2
        for i in range(1, size+1):
            i_part = (i-(k+0.5))**2
            for j in range(1, size+1):
                H[i-1, j-1] = peak*math.exp((i_part + (j-(k+0.5))**2)/width)

    # normalize the matrix
    H = H / np.sum(np.concatenate(H))
    return H


def blur(image, size, sigma):
    G = Gaussian2D(size, sigma)
    return correlate2d(image, G, mode='same', boundary='symm')


def gradient_calc(image):
    # get some arrays ready
    sobx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
```

```
    soby = sobx.transpose()
    phi = np.zeros(image.shape)
    M = np.zeros(image.shape)

    # need to slightly pad but we don't need to calculate the new borders
    img = pad_array(image, 1)
    x, y = img.shape
    for i in range(1, x-2):
        for j in range(1, y-2):
            # calculate both at once rather than separately
            dx = -1*img[i-1,j-1] -2*img[i-1,j] -1*img[i-1,j+1] +img[i+1,j-1] +2*img[i+1,j
                ] +img[i+1,j+1]
            dy = -1*img[i-1,j-1] -2*img[i,j-1] -1*img[i+1,j-1] +img[i-1,j+1] +2*img[i,j
                +1] +img[i+1,j+1]

            phi[i-1,j-1] = np.arctan2(dy,dx)/np.pi*180

            # magnitude
            M[i-1, j-1] = (dx**2+dy**2)**0.5
    return phi, M


def buildRtable(images, point, threshold, verbose=False):
    r_table = {}
    index = 0
    for img in images:
        # gradient calculations
        phi,M = gradient_calc(img)

        # we can ready some queues for threshold information
        strong_queue = []
        weak_list = []

        # non-maxima suppression and edge candidate detection
        N = copy.deepcopy(M)
        for row in range(0, M.shape[0]-1):
            for col in range(0, M.shape[1]-1):
                p = phi[row,col]
                # eight cases decomposed into four by arctan range (-90deg<->90deg)
                if p < 22.5 and p >= -22.5: # 4,6
                    coords = [[row-1, col], [row+1, col]]
                elif p < 67.5 and p >= 22.5: # 1,9
                    coords = [[row-1, col-1], [row+1, col+1]]
                elif p <=90 and p >= 67.5 or p <= -67.5 and p >= -90: # 2, 8
                    coords = [[row, col+1], [row, col-1]]
                else: # 3, 7
                    coords = [[row-1, col+1], [row+1, col-1]]
                if M[row, col] < M[coords[0][0], coords[0][1]] or M[row, col] < M[coords
                    [1][0], coords[1][1]]:
                    N[row,col] = 0

                # threshold control; values just for informative picture
                if N[row,col] > threshold[1]:
                    N[row,col] = 128
                    strong_queue.append((row,col))
                elif N[row,col] > threshold[0]:
                    N[row,col] = 64
                    weak_list.append((row, col))
```

```
                else:
                    N[row,col] = 0

        # edge strengthening
        while len(strong_queue) > 0:
            # get pixel at head
            px = strong_queue[-1]

            # remove the pixel from the queue
            strong_queue.pop(-1)

            # begin processing
            N[px[0], px[1]] = 255
            for i in range(-1,2):
                for j in range(-1,2):
                    # use our weak flag value to speed things up
                    if N[px[0]+i, px[1]+j] == 64:
                        # set it to the strong-but-not-processed value (which is unused
                            due to the queue)
                        N[px[0]+i, px[1]+j] = 128

                        # pop the pixel from the weak_list and add it to the queue
                        weak_list.pop(weak_list.index((px[0]+i, px[1]+j)))
                        strong_queue.append((px[0]+i, px[1]+j))

        # cull unverified weak edges
        for px in weak_list:
            N[px[0], px[1]] = 0

        if verbose:
            cv2.imwrite("out/{}_ref_edges.png".format(index), N.astype(np.uint8))
            cv2.imwrite("out/{}_ref_grad.png".format(index), M.astype(np.uint8))
            cv2.imwrite("out/{}_ref_phi.png".format(index), phi.astype(np.uint8)+180)
        index +=1

        # build r-table
        for i in range(0, N.shape[0]):
            for j in range(0, N.shape[1]):
                if N[i,j] == 255:
                    theta = round(phi[i,j], 1)
                    rho = (i-point[0], j-point[1]) # just a displacement vector
                    if theta in r_table.keys():
                        if rho not in r_table[theta]:
                            r_table[theta][rho] = M[i,j]
                        else:
                            r_table[theta][rho] += M[i,j]
                    else:
                        r_table[theta] = {rho: M[i,j]}
    return r_table


def genAccumulator(image, r_table, threshold, rotations=[0], scales=[1], verbose=False):
    ''' Find boundaries in image '''
    # gradient calculations
    phi,M = gradient_calc(image)

    # we can ready some queues for threshold information
    strong_queue = []
```

```
weak_list = []

# non-maxima suppression and edge candidate detection
N = copy.deepcopy(M)
for row in range(1, M.shape[0]-1):
    for col in range(1, M.shape[1]-1):
        p = phi[row,col] % 90
        # eight cases decomposed into four by arctan range (-90deg<->90deg)
        if p < 22.5 and p >= -22.5: # 4,6
            coords = [[row-1, col], [row+1, col]]
        elif p < 67.5 and p >= 22.5: # 1,9
            coords = [[row-1, col-1], [row+1, col+1]]
        elif p <=90 and p >= 67.5 or p <= -67.5 and p >= -90: # 2, 8
            coords = [[row, col+1], [row, col-1]]
        else: # 3, 7
            coords = [[row-1, col+1], [row+1, col-1]]
        if M[row, col] <= M[coords[0][0], coords[0][1]] or M[row, col] <= M[coords
            [1][0], coords[1][1]]:
            N[row,col] = 0

        # threshold control; values just for informative picture
        if N[row,col] > threshold[1]:
            N[row,col] = 128
            strong_queue.append((row,col))
        elif N[row,col] > threshold[0]:
            N[row,col] = 64
            weak_list.append((row, col))
        else:
            N[row,col] = 0

# edge strengthening
while len(strong_queue) > 0:
    # get pixel at head
    px = strong_queue[-1]

    # remove the pixel from the queue
    strong_queue.pop(-1)

    # begin processing
    N[px[0], px[1]] = 255
    for i in range(-1,2):
        for j in range(-1,2):
            # use our weak flag value to speed things up
            if N[px[0]+i, px[1]+j] == 64:
                # set it to the strong-but-not-processed value (which is unused due
                    to the queue)
                N[px[0]+i, px[1]+j] = 128

                # pop the pixel from the weak_list and add it to the queue
                weak_list.pop(weak_list.index((px[0]+i, px[1]+j)))
                strong_queue.append((px[0]+i, px[1]+j))

# cull unverified weak edges
for px in weak_list:
    N[px[0], px[1]] = 0

if verbose:
    cv2.imwrite("out/test_edges.png", N)
```

```
    # build vote-space
    P = np.zeros((image.shape[0], image.shape[1], len(rotations), len(scales)))
    for i in range(N.shape[0]):
        for j in range(N.shape[1]):
            if N[i,j] == 255:
                theta = round(phi[i,j],1)
                if theta in r_table.keys():
                    for rho in r_table[theta]:
                        p = (int(i-rho[0]), int(j-rho[1]))
                        for t in rotations:
                            tr = t*np.pi/180
                            cos = np.cos(tr)
                            sin = np.sin(tr)
                            for s in scales:
                                xr = int(i - (rho[0]*cos - rho[1]*sin)*s)
                                yr = int(j - (rho[0]*sin + rho[1]*cos)*s)
                                if -1 < xr < N.shape[0] and -1 < yr < N.shape[1]:
                                    P[xr, yr, rotations.index(t), scales.index(s)] +=
                                        r_table[theta][rho]

    return P


def getPeaks(accumulator, threshold):
    peaks = copy.deepcopy(accumulator)
    size = 25
    B = np.zeros((size,size)) + 1/(size*size)
    for t in range(peaks.shape[2]-1):
        for s in range(peaks.shape[3]-1):
            peaks[:,:,t,s] = correlate2d(peaks[:,:,t,s].astype(float), B, mode='same',
                boundary='symm')

    peaks = (peaks >= threshold) * peaks
    return peaks


def displayResult(image, center, size, rotation, scale):
    box = (size[0]*scale, size[1]*scale)
    uleft = (int(center[1]-box[1]/2), int(center[0]-box[0]/2))
    uright = (int(center[1]+box[1]/2), int(center[0]+box[0]/2))
    rect = np.zeros(image.shape)
    rect = cv2.rectangle(image, uright, uleft, (0,0,255), 2)


    return rect
```