Caleb Johnson
EE 7700

Final Project Report

1. Paper

The paper aims to separate a rainy image into two layers, the background layer and the rain layer. Then by subtracting the rain layer, a rainless image can be produced. A link to the paper is found here:
https://openaccess.thecvf.com/content_ICCV_2017/papers/Zhu_Joint_Bi-Layer_Optimization_ICCV_2017_paper.pdf
No codes are provided for this paper, so my implementation will rely on the mentioned methods and given mathematical equations, which I will now describe. The method is broken into three parts: Calculating rainy patches/rain direction, the prior/optimization equations, and the optimization step.

a) Calculate rainy patches and rain direction

The method for this part involves three steps.

Step 1: Locate rain dominated regions.
The authors state that background details in rain dominated regions tend to have less variance. Based on this assumption they perform the following calculation to find N rain dominated windows:

- Compute the gradient angle at each pixel in the image
- Use sliding window to create histogram of angles in each window from 0 to pi
    - Sliding window has size of 31x31
    - Stride of 8
- For each window determine δ, the amount of angles in the highest bin and its two adjacent bins
    - These regions have less background details and are thus likely to be rainy
- N windows with highest δ designated as rainy regions
    - N set to 20

This results in rainy windows, highlighted in red in the following image:

Step 2: Calculate global rain direction
The following calculation is performed considering each of the N rainy windows to find a global rain direction:

- Utilize canny edge detector to find edges in each rainy window
- Use Hough transform to find longest line in each window
- Find slope of longest line
- Median of all slopes in these windows decided as global rain direction

Author's result:

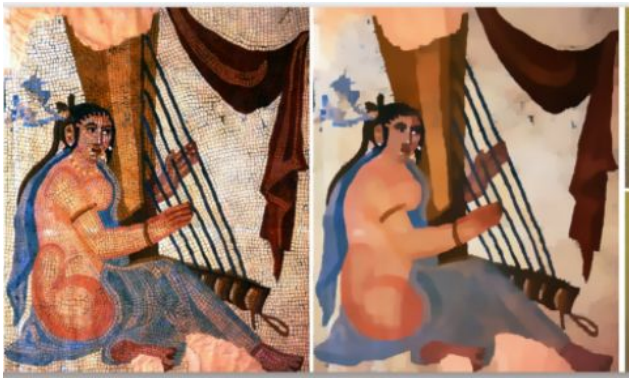
Step 3: Select rainy patches
This is accomplished simply by randomly selecting 20N. patch_width x patch_width patches within each rainy window. N = 20 so there are 400 rainy patches, and patch_width is set to 7 so they are 7x7 patches.

b) Optimization Equation
Now we utilize the rainy patches and global rain direction for consideration in the optimization equation. The optimization equation is as follows:

$$\min_{\mathbf{B},\mathbf{R}} ||\mathbf{I}-\mathbf{B}-\mathbf{R}||_F^2+\lambda_1\Psi(\mathbf{B})+\lambda_2\Phi(\mathbf{B})+\lambda_3\Omega(\mathbf{R})$$

The three equations there, called the priors, each have a specific purpose within the total optimization equation. These will now each be explained.

Sparsity prior: Ψ(B)
The intuition for this prior is to utilize Centralized Sparse Representation to separate an image from is rainy details. CSR is usually used to remove texture from images, such as in the following example:



So it is intuitive that such a technique can be used to remove blemishing rain details from an image as well. However, it can be seen that some details of the painting itself are removed in this example- This is where the other 2 priors come into play.
The equation the authors give for the sparsity prior is as follows:

$$\Psi(\mathbf{B}) = \|\boldsymbol{\alpha}\|_1 + \gamma \sum_{i \in \mathbf{B}} \|\alpha_i - \mu_i\|_1$$

Where…

- α_i: Sparse code of patch centered at pixel i
    - Utilizing centralized sparse representation
- α: Vector of all α_i
- γ: weight
- μ_i: Weighted average of the sparse codes of M nonlocal patches most similar to patch at pixel i

And μ_i is calculated in the following manner:

$$\mu_i = \frac{1}{\Upsilon} \sum_{m=1}^{M} \tau_{i,m} \alpha_{i,m},$$

Where…

- $\alpha_{i,m}$: Sparse code of mth similar patch
- $\tau_{i,m}$: Distance between that patch and pixel i
- Y: Sum of $\tau_{i,m}$ for normalization

This is the reference given by the authors for their implementation of CSR:
https://ieeexplore-ieee-org.libezp.lib.lsu.edu/stamp/stamp.jsp?tp=&arnumber=6126377

Rain Direction Prior: Φ(B)
This prior aims to preserve background details which would otherwise be removed in the CSR step. The intuition of this prior is that it highlights lines which go in the same direction as rain streaks. This prior can then be used to identify background details which may be mistaken as rain streaks so that they are not removed from the image.

The rain direction prior is calculated in the following manner:

$$\mathbf{\Phi}(\mathbf{B}) = \sum_{i \in \mathbf{B}} \frac{1}{\mathbf{\Theta}_i(\mathbf{B}) + \varepsilon_1}$$

And $\Theta_i(B)$ is calculated as:

$$\mathbf{\Theta}_i(\mathbf{B}) = \frac{|\vec{d} \cdot \vec{g_i}|}{||\vec{g_i}|| + \varepsilon_1}$$

Where…

- ε: Small constant to avoid division by 0
- Θ: Rain direction map, compares gradient in this patch to global direction
    - d: Global rain direction
    - g_i: Gradient at pixel i

Rain Layer Prior: Ω(R)

This prior aims to restore details which have been pushed to R back into B. Whereas the rain direction prior attempts to prevent the details from being removed in the first place, this prior wants to take details which have already been removed and add them back. This prior is calculated in the following manner:

$$\mathbf{\Omega}(\mathbf{R}) = \sum_{i \in \mathbf{R}} \left\{ w_x(i) \left(\partial_x \mathbf{R}_i\right)^2 + w_y(i) \left(\partial_y \mathbf{R}_i\right)^2 \right\}$$

With $w_x(i)$ and $w_y(i)$ calculated as:

$$w_x(i) = |\partial_x \mathbf{\Gamma}_i(\mathbf{I})|^\eta \text{ and } w_y(i) = |\partial_y \mathbf{\Gamma}_i(\mathbf{I})|^\eta$$

And $\Gamma_i(I)$ calculated as:

$$\mathbf{\Gamma}_i(\mathbf{I}) = \min_r \|P_i - \tilde{P}_r\|_2^2$$

Where…

- wx(i) and wy(i): Smoothing weights on pixel i in x and y directions
    - η: Constant sensitivity parameter
    - Γi(I): Similarity map comparing each patch in I to rain dominated patch
        - Pr: rth rain patch
        - Pi: patch at pixel i

c) Optimization Step

Now the previous equations are utilized within an iterative step equation which is used to calculate B and R for every iteration based on the previous B. Initially, B = I (the original image) and R is a blank image.

B^{t+1} is set by the equation:

$$\min_{\mathbf{B}} ||\mathbf{I} - \mathbf{B} - \mathbf{R}_{k-1}||_F^2 + \lambda_2 \Phi(\mathbf{B}) + \frac{\beta}{2}||\mathbf{B} - \mathbf{D} \circ \boldsymbol{\alpha}^t - \frac{1}{\beta}\mathbf{H}^t||_F^2$$

Where…

- H: Lagrange multiplier of linear constraint, initialized to B
- β: Penalty Parameter
- D = Ω(R)
- α is again the CSR representation

Next, α^{t+1} is updated using B^{t+1}:

$$\min_{\boldsymbol{\alpha}} \frac{\beta}{2}||\mathbf{B}^{t+1} - \mathbf{D} \circ \boldsymbol{\alpha}^t - \frac{1}{\beta}\mathbf{H}^t||_F^2 + \lambda_1 \boldsymbol{\Psi}(\mathbf{B}^{t+1})$$

And then H^{t+1} is updated using α^{t+1} and B^{t+1}:

$$\mathbf{H}^t + \beta(\mathbf{B}^{t+1} - \mathbf{D} \circ \boldsymbol{\alpha}^{t+1})$$

Finally, R^{t+1} is set as I - B^{t+1}.

The authors utilize several iterations of this step and get the following result:

2. Auxiliary Functions

This section will briefly show some auxiliary functions utilized within our implementation. Frobenius norm produces a numerical norm value from an image using numpy.

```
# Returns frobenius norm of an image
def frobenius_norm(I):
    return np.linalg.norm(I, ord=None, axis=None, keepdims=False)
```

Normalize returns a normalized vector.

```
# Normalize vector
def normalize(v):
    return v / np.sqrt(np.sum(v ** 2))
```

Img_Normalize normalizes a grayscale image.

```
# Convert the image into the range of [0.0, 1.0]
def img_normalize(img):
    min_val = np.min(img.ravel())
    max_val = np.max(img.ravel())
    if max_val == 0 and min_val == 0: # Image is only 0s, just return that
        return img
    output = (img.astype('float')-min_val)/(max_val - min_val)
    return output
```

Partial_both_grayscale gets the partial derivative of a grayscale image in both the x and y directions.

```
# Gets x and y derivative images
def partial_both_grayscale(img):
    rows = img.shape[0]
    cols = img.shape[1]
    gx = np.zeros((rows-2, cols-2), float)
    gy = np.zeros((rows-2, cols-2), float)
    for i in range(1, rows-2):
        for j in range(1, cols-2):
            gx[i-1, j-1] = (img[i, j+1]-img[i, j-1])
            gy[i-1, j-1] = (img[i+1, j]-img[i-1, j])
    return gx, gy
```

Gradient angles, given the x and y derivatives, finds the angle of the gradient at each pixel and returns an image of these values.

```
# Calculates the gradient angle at each pixel given both derivatives
def grad_angle(gx, gy):
    rows = gx.shape[0]
    cols = gx.shape[1]
    angles = np.zeros((rows, cols), float)
    for i in range(1, rows):
        for j in range(1, cols):
            y = gy[i-1, j-1]
            x = gx[i-1, j-1]
            angles[i-1, j-1] = math.atan2(y, x)
    return angles
```

Pixel gradient just gets the gradient of a single pixel given the pixels around it.

```
# Returns the gradient at a specific pixel
def pixel_gradient(I, x, y):
    vector = []
    partx = (I[x, y + 1] - I[x, y - 1]) / 2.0
    party = (I[x + 1, y] - I[x - 1, y]) / 2.0
    vector.append(partx / 3)
    vector.append(party / 3)
    return [vector[0], vector[1]]
```

Global rain vector simply represents the rain direction as a vector, by putting a 1 in the x position and the direction in the y position.

```
# Represents the global rain direction as a vector and returns
def global_rain_vector():
    return [1, global_rain_direction]
```

Get patch gets the starting x and y value of a patch centered at the given x and y value.

```
# Return starting x and y value of patch centered at pixel value
def get_patch(pixel):
    startx = int(max(0, pixel[0] - patchwidth/2))
    starty = int(max(0, pixel[1] - patchwidth/2))
    return (startx, starty)
```

Finally, patch difference finds the difference of two patches given their starting x and y positions.

```
# Based on starting pixels of two patches, slices the image and returns the difference of the two
def patch_difference(I, patch1, patch2):
    patch1_img = I[patch1[0]:patch1[0] + patchwidth, patch1[1]:patch1[1] + patchwidth]
    # Account for when patch reached edge of image and must be truncated
    adjustedwidth = patch1_img.shape[0]
    adjustedheight = patch1_img.shape[1]
    patch2_img = I[patch2[0]:patch2[0] + adjustedwidth, patch2[1]:patch2[1] + adjustedheight]
    return patch1_img - patch2_img
```

3. Implementation

I will now describe my code implementation of the author's original equations. Many changes were made between some of the code here and the final code; these will be explained in the following section (Modifications).

This section will function by posting screenshots of my code while explaining generally what that code is doing. The code screenshots will include comments explaining the individual lines more. The code begins by declaring the constant variables using parameters specified by the authors:

```
# Declaring parameter values, most of them are constant

lambda_1 = 1.0
lambda_2 = 0.001
lambda_3 = 0.01

W_r = 31 # Window size
W_s = 8 # Window step size
N = 20 # Number of rainy windows
patchwidth = 7 # Width of rainy patches

T = 2
K = 25 # Tuned parameter in range [4, 50]
```

It then reads in an image, converts it to grayscale, normalizes, it, takes the partial derivative in both directions, and uses these partial derivatives to compute the gradient angle on each pixel.

```
# Read in image, compute gradients, gradient angle, attempt to extract rainy patches from gradient angle

I = cv2.imread('TestUmbrella.png')

I_gray = cv2.cvtColor(I, cv2.COLOR_BGR2GRAY)

I_gray_norm = f.img_normalize(I_gray)
gx, gy = f.partial_both_grayscale(I_gray)

gradient_angles = f.grad_angle(gx, gy)
```

Now that these initial steps have been taken, we must begin the process to identify rainy patches as specified by the authors.

First, we implement a loop which goes through each window of specified dimension within the image. It performs the process specified by the authors by placing each gradient angle within bins in a histogram, and checking the number of values in the most filled bin and its two adjacent bins, this value equals delta for this window. These windows are then sorted by descending delta and the top N windows are chosen as the rainy windows.

```
# Windows is a list of 2 tuples, the coords of the top-left value in each window
windows = list()
# Deltas is a parallel list of delta values
deltas = list()

# This loop implements a sliding window size W_r x W_r and with stride W_s
# For each window a histogram with 10 bins of gradient angles is calculated
# Delta for each window will equal the bin with the most elements and its two adjacent bins
# After delta is calculated for each window, the N windows with the highest delta are selected as rainy window
y = 0
x = 0
while True:
    # Construct a histogram of angle values for each window
    histogram = np.zeros(10, int)
    # Window size is W_r x W_r
    # x and y indicate the top-left corner of the window
    for ix in range(x, x+W_r):
        for iy in range(y, y+W_r):
            angle = gradient_angles[ix, iy]
            for anglebin in range(1, 11):
                # This looks complicated but just checks which range the angle resides in
                # For example, when bin = 1, it checks if angle is in range [0, pi/10)
                # When bin = 2, it checks if angle is in range [pi/10, 2*pi/10), etc.
                if (anglebin - 1) * math.pi / 10 <= angle < anglebin * math.pi / 10:
                    histogram[anglebin-1] += 1
                    break # Break after finding correct bin
    # Now must calculate delta, which equals the number of values in the bin with the most values, and its two adjacent bins
    maxindex = np.argmax(histogram)
    delta = histogram[maxindex]
    if maxindex > 0: # Consider previous adjacent bin unless this is the first bin
        delta += histogram[maxindex - 1]
    if maxindex < 9: # Consider next adjacent bin unless this is the last bin
        delta += histogram[maxindex + 1]
    window = (x, y)
    windows.append(window) # Add to list of windows
    deltas.append(delta) #Add to list of deltas
    # Now must determine how to increment x and y values
    if x + W_s + W_r > n_row - 1 and y + W_s + W_r > n_col - 1: # If next window is outside image in both dimensions, end has been reached
        break
    elif x + W_s + W_r > n_row - 1: # If next window would be outside image in x direction, reset x direction and go to next row
        x = 0
        y += W_s
    else: # Otherwise just increment x direction
        x += W_s

# Now we want to find the index of the max deltas, and get the windows with the same index
# These will be considered the rainy windows
rainywindows = list()
for i in range(0, N):
    # Get the max N delta values, append them to rainy windows, and remove those values from both lists
    maxindex = np.argmax(deltas)
    rainywindows.append(windows.pop(maxindex))
    deltas.pop(maxindex)
```

Now, our aim is to find the longest Hough line in each rainy window and record its angle. The median of these angles is then selected as the global rain direction.

```
# Now the goal is to find the longest rain streak (edge) in each window and take their slope
# The median of the list of slopes is the global rain direction
I_windows = I.copy()
rainslopes = list()
for window in rainywindows:
    # Perform canny edge detection on each window to hopefully get edges for rain streaks
    cropped = I[window[1]:window[1]+W_r, window[0]:window[0]+W_r]
    cropped_canny = cv2.Canny(cropped, 0, 100) # Experimentally, this threshold seems to be the best at showing the rain streaks as edges

    # cv2.imshow('Rainy Window', cropped)
    # cv2.imshow('Rainy Window Edges', cropped_canny)
    # cv2.waitKey(0)
    # cv2.destroyAllWindows()

    # Now use hough transform to identify lines (rain streaks) and try to find longest one
    lines = cv2.HoughLines(cropped_canny, 1, np.pi / 180, 20)
    if not (lines is None): # Once window identification works, this should never be false
        lengths = list()
        for line in lines:
            # Do some calculations to find two points (start and end) of the line and calculate its length
            rho = line[0][0]
            theta = line[0][1]
            a = math.cos(theta)
            b = math.sin(theta)
            x0 = a * rho
            y0 = b * rho
            pt1 = (int(x0 + 1000*(-b)), int(y0 + 1000 * a)) # Line start
            pt2 = (int(x0 - 1000*(-b)), int(y0 - 1000 * a)) # Line end
            length = math.sqrt((pt2[0] - pt1[0])**2 + (pt2[1] - pt2[0])**2) # Calculates length of line
            lengths.append(length) # Add to length list
        maxindex = np.argmax(lengths) # Get maximum length
        maxline = lines[maxindex] # This is the line with maximum length
        rainslopes.append(maxline[0][1]) # Add the theta of the maximum length line to the list of rain slopes

    # Draw rectangle over each window to identify them
    bottomright = (window[0] + W_r, window[1] + W_r)
    cv2.rectangle(I_windows, window, bottomright, (0, 0, 255), 2)


# Take median rain slope from the list of rain slopes of the longest line in each rainy window, this is the global rain direction
global_rain_direction = np.median(rainslopes)
# print(global_rain_direction)
# Draw line representing global rain direction on image
if global_rain_direction > 0: # positive slope, draw from bottom left
    bottomleft = (0, I_windows.shape[0])
    adjust_amount = math.ceil(I_windows.shape[0] / global_rain_direction) # Want to draw line slightly past edge
    adjusted_y = int(adjust_amount*global_rain_direction) # Calculate y value of point based on slope and how far to go
    adjusted = (bottomleft[0] + adjust_amount, bottomleft[1] - adjusted_y) # Adjust x forward and y up to make a sloped line
    cv2.line(I_windows, bottomleft, adjusted, (0, 255, 0), 1)
else: # negative slope, draw from bottom right
    abs_slope = abs(global_rain_direction)
    bottomright = (I_windows.shape[1], I_windows.shape[0])
    adjust_amount = max(math.ceil(I_windows.shape[0] / abs_slope), math.ceil(I_windows.shape[1] / abs_slope)) # Want to draw line slightly past edge
    adjusted_y = int(adjust_amount*abs_slope) # Calculate y value of point based on slope and how far to go
    adjusted = (bottomright[0] - adjust_amount, bottomright[1] - adjusted_y) # Adjust x backward and y up to make a sloped line
    cv2.line(I_windows, bottomright, adjusted, (0, 255, 0), 1)
```

Now, we select 20N rainy patches randomly within the rainy windows.

```
# Next step: Extract rainy patches of size patchwidth x patchwidth
# Want total of 20N rainy patches, chosen randomly
rainy_patches = list()
for i in range(0, 20*N):
    windowindex = random.randint(0, N-1) # Random int in range [0, N-1] to select random window
    selectedwindow = rainywindows[windowindex]
    # Random x, y coordinates, make sure they don't go past edge
    startx = random.randint(0, W_r - patchwidth - 1)
    starty = random.randint(0, W_r - patchwidth - 1)
    patch = (startx, starty)
    rainy_patches.append(patch)

    # Draw rectangle over each patch to identify them
    topleft_patch = (selectedwindow[0] + startx, selectedwindow[1] + starty)
    bottomright_patch = (selectedwindow[0] + startx + patchwidth, selectedwindow[1] + starty + patchwidth)
    #cv2.rectangle(I_windows, topleft_patch, bottomright_patch, (0, 0, 255), 2)

# Show image with windows, lines, and patches drawn over it
cv2.imshow('Rainy Windows and Patches', I_windows)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Now we will show the process for calculating each of the optimization priors specified in the paper.
Here is the code for the calculation of Psi, the centralized sparse representation prior. This code is from before I added my approximation, but without the code for calculation of the CSR of a specific patch, which would be within function alpha_i.

```
#########################################################################
############################### psi #####################################

# psi = sparsity prior
# This effectively removes rain streaks, but background details as well
gamma = 5  # some weight
M = 20

def regularize_psi(B):
    alphavector = normalize(alpha(B))
    rows = B.shape[0]
    cols = B.shape[1]
    # loop through each pixel and add to sum
    output = np.zeros((rows-1, cols-1), float)
    for i in range(rows - 1):
        for j in range(cols - 1):
            pixel = B[i, j]
            output[i][j] = normalize(alphavector[pixel] - mu_i(B, pixel))
    return alphavector + gamma * sum
```

```
# return vector of all alpha_i
def alpha(B):
    rows = B.shape[0]
    cols = B.shape[1]
    output = np.zeros((rows-1, cols-1), float)
    for i in range(rows - 1):
        for j in range(cols - 1):
            pixel = B[i, j]
            output = alpha_i(B, pixel)
    return output
```

Next the code for Phi, the rain direction prior. This is a simple implementation of the equations given in Section 1.

```
#############################################################################
############################## phi ##########################################

# phi = Rain direction prior
# Helps put back in background details that could be mistaken for streaks

epsilon_1 = 0.0001 # Constant to avoid division by 0

def regularize_phi(B):
    rows = B.shape[0]
    cols = B.shape[1]
    # loop through each pixel and assign value to pixel in output
    output = np.zeros((rows, cols), float)
    for i in range(rows-1):
        for j in range(cols-1):
            term = theta_i(B, i, j) + epsilon_1
            output[i][j] = (1 / term)
    return output

# compares rain direction and pixel gradient
def theta_i(B, x, y):
    gradient = pixel_gradient(B, x, y)
    numerator = np.dot(global_rain_vector(), gradient)
    denominator = np.linalg.norm(gradient) + epsilon_1
    return abs(numerator) / denominator
```

Finally, we consider the code for omega, the rain layer prior. This is again simply an implementation of the functions given in Section 1 for omega. It also utilizes the auxiliary patch difference function shown in Section 2.

```
##########################################################################
############################## omega #####################################

# omega = rain layer prior
# Push scene details from R back into B

eta = 1.2 # sensitivity parameter

def regularize_omega(R, I):
    rows = R.shape[0]
    cols = R.shape[1]
    # Calculate derivative of R
    R_norm = img_normalize(R)
    dx, dy = partial_both_grayscale(R_norm)
    # loop through each pixel and add to sum
    output = np.zeros((rows, cols), float)
    for i in range(rows-2):
        for j in range(cols-2):
            pixel = (i, j)
            dx_pixel = dx[pixel]
            dy_pixel = dy[pixel]
            biggamma = gamma_i(I, pixel)
            term1 = weight_x(I, biggamma, dx_pixel) * (dx_pixel ** 2)
            term2 = weight_y(I, biggamma, dy_pixel) * (dy_pixel ** 2)
            output[i][j] = (term1 + term2)
    return output

# Smoothing weight on pixel i in x direction
def weight_x(I, biggamma, partial):
    term = partial * biggamma
    return abs(term) ** eta

# Smoothing weight on pixel i in y direction
def weight_y(I, biggamma, partial):
    term = partial * biggamma
    return abs(term) ** eta

# Similarity map
def gamma_i(I, pixel):
    vector = list()
    patch = get_patch(pixel)
    # Go through each rain patch and compare similarity, return min of norm of similarities
    for rainy_patch in rainy_patches:
        difference = patch_difference(I, patch, rainy_patch)
        vector.append(frobenius_norm(difference))
    return min(vector)
```

Now finally we will show the code which implements the optimization step. B is initialized as I, R as 0s, H as B and alpha as the CSR result.

```
# Initialize vars for use in minimization code
B = I_gray_norm.copy()
B = B.astype('float32') # Must convert everything to float32 for opencv functions to work
R = I_gray_norm - B
f.global_rain_direction = global_rain_direction
f.rainy_patches = rainy_patches
f.patchwidth = patchwidth
alpha = f.img_normalize(f.alpha(B))
beta = .01
H = B.copy() # Lagrange multiplier of linear constraint. Should equal (B - D * alpha) but D will initially equal 0 so initial just = B
```

The optimization step then updates B^{t+1} first, then α^{t+1}, next H^{t+1}, and finally R^{t+1}, as specified by the authors of the paper.

```
# Iterative code for optimization
for t in range(0, 10):
    # Update B step 1: update B_t+1
    firstpart = f.img_normalize(I_gray_norm - B - R)
    secondpart = lambda_2 * f.img_normalize(f.regularize_phi(B))
    D = lambda_3 * f.img_normalize(f.regularize_omega(R, I_gray_norm))
    tonormalize_1 = B - D * alpha - (1 / beta) * H
    thirdpart = f.img_normalize(tonormalize_1)
    B = B + firstpart + secondpart + (beta / 2) * thirdpart
    B = f.img_normalize(B)
    B = B.astype('float32')
    # Update B step 2: update alpha_t+1
    tonormalize_2 = B - D * alpha - (1 / beta) * H
    alpha = f.img_normalize(tonormalize_2) + lambda_1 * f.img_normalize(f.regularize_psi(B))
    alpha = f.img_normalize(alpha)
    alpha = alpha.astype('float32')
    # Update B step 3: Update H_t+1
    H = H + beta * (B - D * alpha)
    H = f.img_normalize(H)
    H = H.astype('float32')
    # Update R
    R = I_gray_norm - B
    R = f.img_normalize(R)
    R = R.astype('float32')
```

4. Modifications

Several modifications were made to the original design shown in the previous section. These will now be discussed.

a) Rainy window detection

The method given by the authors did not seem to function for identifying rainy windows. Thus I instead used a method you proposed that looks for a number of lines going in the same direction within the image.

The surrounding code for iterating through windows, setting a delta value, then selecting the top N desirable values of these deltas is unchanged. What happens within each window, however, is changed. Each window is cropped from a version of the image which was run through a canny edge detector, with a threshold of 80. The Hough Line function is then run on this cropped window to identify lines within it. We only consider windows with at least 9 lines, otherwise delta is set to a very high value. If there are at least 9 lines within the window, then the standard deviation of the angles of each of these lines is taken. Delta for this window is then set to this standard deviation, and the N lowest deltas are set as the rainy windows.

```
# Use our own method where we find hough lines in a window with at least 7 lines
# Check each of their angles and save delta as the standard deviation of the angles
# This will make it so that windows with several lines going in the same direction will be considered
y = 0
x = 0
while True:
    # Window size is W_r x W_r
    # x and y indicate the top-left corner of the window
    cropped = canny_gray[y:y+W_r, x:x+W_r]
    # Now use hough transform to identify lines (rain streaks) and try to find longest one
    lines = cv2.HoughLines(cropped, 1, np.pi / 180, 20)
    delta = 1000000.0
    if not(lines is None) and lines.shape[0] > 8: # Want at least 7 lines to consider this window
        angles = list()
        for line in lines:
            angles.append(line[0][1])
        delta = np.std(angles) # Find the standard deviation of the angles, low std dev means lines are going in the same direction
    window = (x, y)
    windows.append(window) # Add to list of windows
    deltas.append(delta) # Add to list of deltas
    # Now must determine how to increment x and y values
    if x + W_s + W_r > n_row - 1 and y + W_s + W_r > n_col - 1: # If next window is outside image in both dimensions, end has been reached
        break
    elif x + W_s + W_r > n_row - 1: # If next window would be outside image in x direction, reset x direction and go to next row
        x = 0
        y += W_s
    else: # Otherwise just increment x direction
        x += W_s
```

The process for finding the global rain direction is then also the same as before, except that I found taking the max of the angles gave a better result than the median.

b) Centralized Sparse Representation

Due to being unable to reproduce the CSR paper, I instead roughly approximated a CSR result using smoothing. Through experimentation I found the best result was given using median smoothing with size 3. I actually implemented my own smoothing algorithm before I realized I could just utilize the ones already existing within OpenCV.

```
##########################################################################
########################### psi/alpha ####################################

# psi = sparsity prior
# This effectively removes rain streaks, but background details as well
# Since we could not get psi fully working we just use our own basic smoothing method
def regularize_psi(B):
    return alpha(B)

# Go to each patch in the picture and average values to smooth picture
def alpha(B):
    return cv2.medianBlur(B, 3)
```

c) Optimization Step

I found that the author's optimization step did not produce very good results for me. I thus attempted to create my own, simplified optimization step instead. Although the final result looks very simple, it was created through much experimentation:

```
# Iterative code for optimization
for t in range(0, 10):
    # Combine the three priors to produce a less rainy result over several iterations
    term1 = lambda_1 * f.img_normalize(f.regularize_psi(B))
    term2 = lambda_2 * f.img_normalize(f.regularize_phi(B))
    term3 = lambda_3 * f.img_normalize(f.regularize_omega(R, I_gray_norm))
    B = term1 + term2 - term3
    B = f.img_normalize(B)
    B = B.astype('float32')
    R = I_gray_norm - B
    R = f.img_normalize(R)
    R = R.astype('float32')
```

Along with this, I also tuned the lambda values through experimentation which gave better results when used. The final values were 1, .1, and .1 for Lambda 1, 2, and 3 respectively.

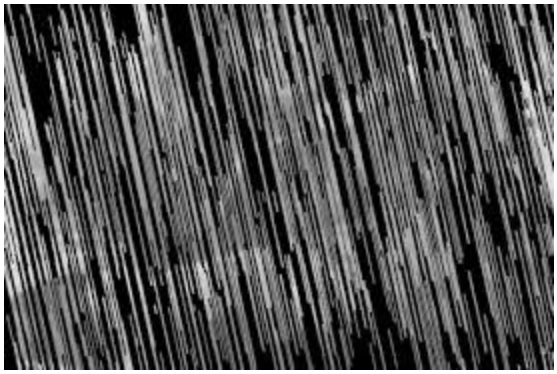
I also utilized 10 iterations of optimization, whereas the author's used only 3. I found that further iterations after 10 did not change the image too much.

5. Results

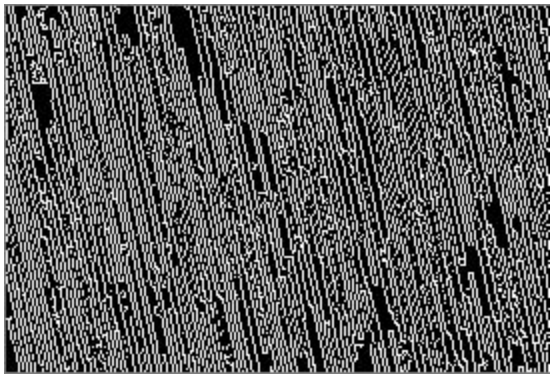
Now we will go through the results using some test images. We will first review the results for the rain window/rain direction detection part, then some sample outputs using the priors, and finally the processed images which have gone through the full process.

a) Rainy Windows and Rain Direction

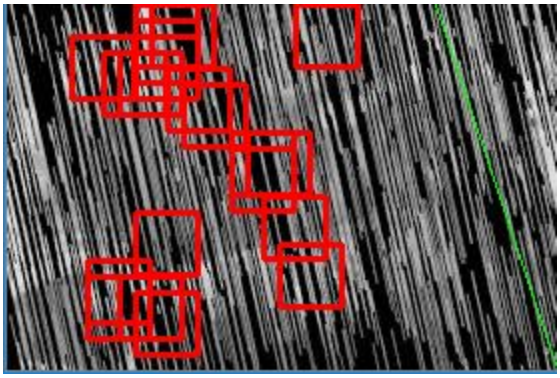
A test image which is fully rainy:



Canny edge detector on this image showing the streaks:



Global rain direction, the main purpose of using this test image:

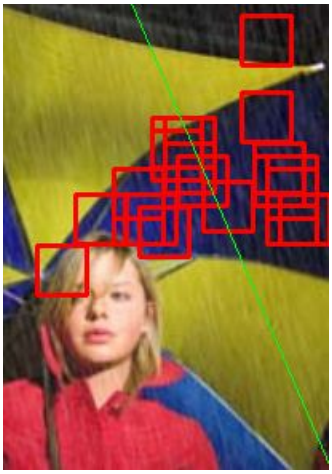


Our main test image:

Here is the author's identified rainy windows, mainly on the blue part of the umbrella:

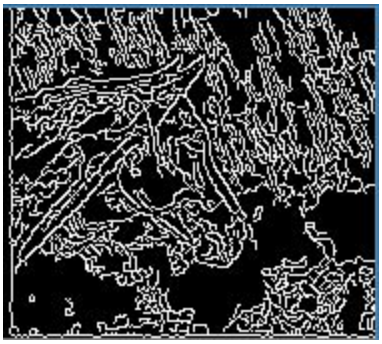
Canny edge detector, indeed showing most rain edges in the blue part of the umbrella:

Our result, which identified similar areas and the rain direction:

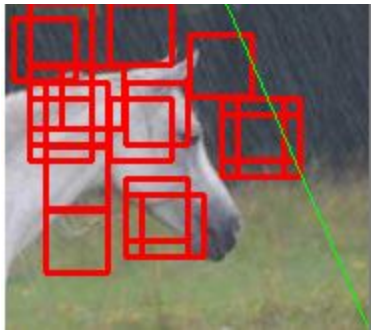


Here is a less successful result, using another test image:

The canny edge detector does not detect the streaks in this image as well:



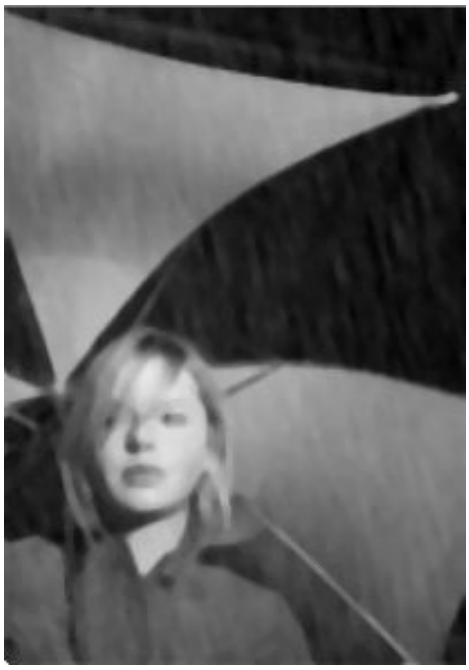
This causes the rainy windows to largely be identified around the edge of the horse's head, however the final direction is still correct:



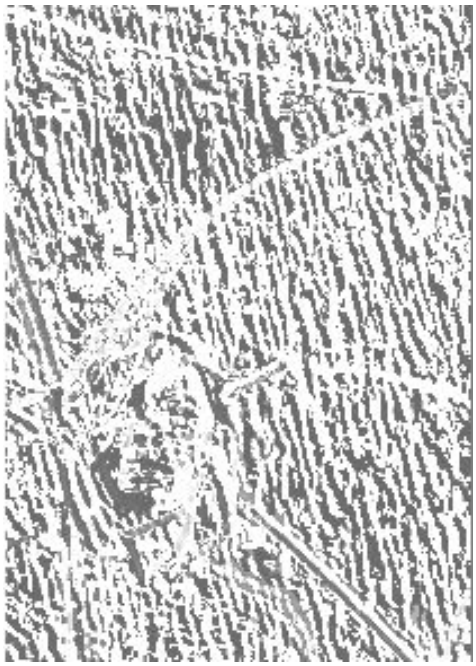
b) Prior Results

For this section we will use our main test image (the umbrella image). Here is that image in grayscale, for reference:

Using our smoothing method to approximate CSR, this result is produced, the Psi prior:



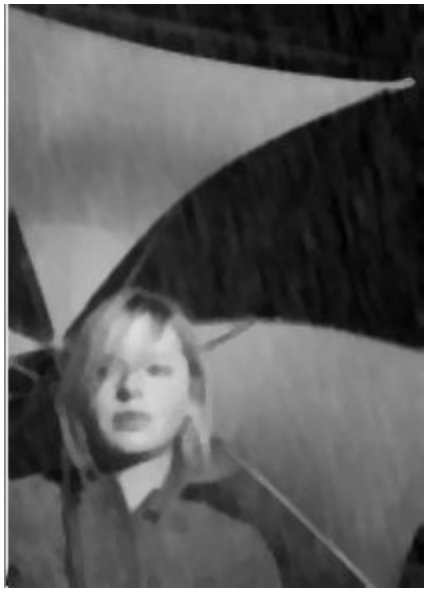
Now we will observe our Phi prior, which attempts to identify details in the same direction as rain streaks which could be mistaken for rain streaks:



Now we observe the Omega prior. For this sample it was run on the original image, though it should normally be run on the rain layer R. Here we attempt to identify details in R which should be pushed back into B:

c) Final Results

Now we will observe some final results, first using our main image. After running for 10 iterations, we found the following result with this image:



And here is the rain layer from this result, notice stronger lines from the blue umbrella regions:

For comparison, here is the author's result in grayscale:

This was quantitatively compared to our result by taking a sum of the absolute value of differences between our result and theirs. This sum was then divided by the number of pixels to obtain an average difference. This average difference was found to equal **0.08671782847397563**, which seems to be not too bad all things considered.

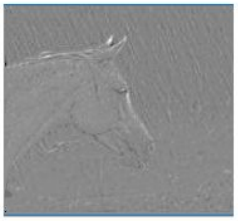
We also compared our result to the image simply being moved, to show the difference made by the optimization step. This average difference was **0.056993937446544296.**

A few more results:

Original Image

Our Result

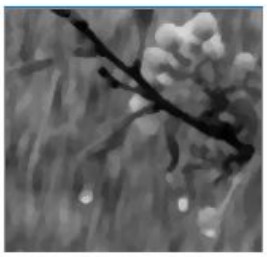
Result Rain Layer



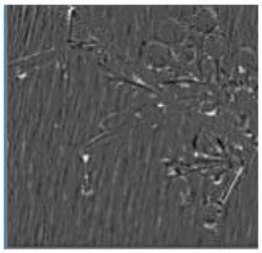
Author's Result

Original Image

Our Result



Result Rain Layer

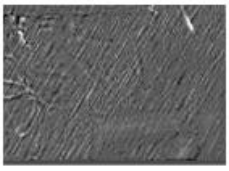


Author's Result

Original Image

Our Result

Result Rain Layer



Author's Result

6. Conclusion

Although several major modifications had to be made to the original author's design, in the end I believe I managed to create a result that at least works somewhat well.

The rainy regions identified are the same as the ones the authors look for, and we are able to successfully determine a rain direction.

Although the author's final results of the de-rained images are better, which can be seen just by looking at them, there is still some rain removal in my results- albeit with some other details being blurred as well from my use of median blurring to approximate CSR.

I believe that if I was able to get a fully functioning CSR system my results would have been roughly on par with the authors.

I learned a lot through the process of creating this, especially when deciding to try my own methods rather than blindly following the author's methods, ending up with functionality that worked better for me overall.

References

L. Zhu, C. Fu, D. Lischinski and P. Heng, "Joint Bi-layer Optimization for Single-Image Rain Streak Removal," *2017 IEEE International Conference on Computer Vision (ICCV)*, Venice, 2017, pp. 2545-2553.

W. Dong, L. Zhang and G. Shi, "Centralized sparse representation for image restoration," *2011 International Conference on Computer Vision*, Barcelona, 2011, pp. 1259-1266.

My implementation can be found here: https://github.com/cjohnson57/DerainingProject