Haze Detection and Removal in digital images with an extended application on videos

Sai Krishna Rohith K. - AP17110010024

Akhileshwar Reddy G. - AP17110010097

Pushyamitra K. - AP17110010061

**Abstract**

Haze is naturally an atmospheric effect. It is a combination of air-light and attenuation. Dehazing images is pivotal in research areas like image processing, pattern recognition, etc. It is crucial in several vision-based applications. There are techniques like Multiple image dehazing methods and Single image dehazing methods. Weather condition-based method, Polarization based method, Depth map-based method belong to the former and techniques like Contrast maximization method, Independent component analysis, Dark channel prior, DCP weighted filter, etc which belong to the latter.

In this project, we have implemented some of the above-mentioned algorithms and analyzed them on different input images in order to recommend a particular algorithm for a particular type of input. This process is extended and worked upon video inputs.

**Introduction**

In an open space, images tend to be influenced by several atmospheric factors like sunlight, ppm of fluids’ in air, etc leading to varying color contrast, blurred visibility, and increased color fading. The presence of haze, fog, smoke, dust, and noise present in the atmosphere is responsible for this.

Haze degrades content and obscures information of images, which can negatively impact vision-based decisionmaking in real-time systems. The air light and attenuation are two main phenomena responsible for haze formation. The air-light enhances the whiteness in the scene and contrast get reduced by attenuation.

Haze removal techniques help in recovering the contrast and color of the scene. These techniques have found many applications in the area of image processing such as consumer electronics, object detection, outdoor surveillance, etc.

Our research would contribute a novel way of applying a particular haze-removal algorithm for certain types of images. This project could be used as a foundation for future work and research. A Machine learning model can be trained using the concepts talked about in this project which would recommend and dehaze an image. So our work could be advantageous to researchers, companies, or individuals trying to implement efficient haze removal techniques. This project would also contribute a technique that would not trade off space or time for efficiency and speed. So, this is a novel/new technique. The research work also gives beneficial insights into how images can be dehazed quickly. Thus our project can potentially be used in real-time systems. This project is both a review of current research and a proposal for a new technique that could be applied upon images and videos.

**Related Works**

There have been several papers published in this particular domain of Image Dehazing. Most of them employ a Dark Channel Prior based algorithm. Though the DCP is celebrated to be one of the popular and successful dehazing techniques, it has its demerits like high computational complexity, over enhancement in the sky region, poor dehazing, and flickering artefacts in video processing, rendering it potentially a liability to be deployed on real-time systems. Similar kinds of DCP techniques with variations are applied in most of the publications.

While some other publications used a technique similar to depth map estimation, Air Light and Transmission map estimation, Depth Map Refinement, Evaluation of restoration model, and Haze image restoration.

However, a very high number of papers mostly focused on the accuracy of the algorithm leaving behind the complexity which turned out highly expensive. Some papers which claimed to show fast-image-defogging had elite hardware requirements.

**Proposed Work**

There are two main goals this project has. One is to analyze a different set of algorithms and techniques and provide a conclusive hypothesis that recommends a certain type of technique for a certain type of image. For example, if an image involves sky then algorithms like DCP would be fruitless as they tend to produce undesired artefacts in the resulting image. An Independent Component Analysis method or Anisotropic diffusion would work well in such cases.

The other goal is, to propose a novel technique that is fast and efficient for dehazing an image. This technique should not be space complexity wise or time complexity wise expensive. The technique must be implementable on embedded systems or Raspberry Pi since this needs to be deployable on real-time systems.

**Requirements**

* Hazed images, Ground Truth images
* A laptop or a computer with at least 4 GB RAM and 2 GB Storage.
* A python compiler (preferably Spyder).

**PART 1**

**Single Image Defogging using Dark Channel Prior**

This Dehaze algorithm contains three steps:

1. Determine the intensity of atmospheric light
2. Estimate transmission map
3. Clarify image

Briefly, the steps followed can be explained as First, the intensity of atmospheric light A is estimated form hazed image I(x). Then, the transmission map t(x) is estimated using A and I(x). Finally, the image is clarified with the image defogging model.

Step#1 Estimate intensity of atmospheric light:

Find the top 0.1% brightest pixels in the dark channel then choose one with the highest intensity as the representing of atmospheric light.

Step#2 Estimate transmission map:

First, find a dark channel based on a local area(coarsemap) Then, the transmission map t(x) is thereby obtained:

t(x) = 1 – defoggingParam \* darkPixelFromCoarseMap / AtmosphericLightIntensity

The defoggingParam is a value between 0 to 1. The higher value the lesser amount of fog would be kept for the distant objects.

Step#3 Clarify image:

Finally, the image is clarified by: J(x)=(I(x)- A)/max(t(x), t0)+A

Where J(x) is output, I(x) is input, t(x) is transmission map, A is atmospheric light, and t0 is set to a constant value to avoid dividing by zero.

**Artificial Multiple Exposure for Image Dehazing**

The purpose of this technique is to build a spatially-varying image enhancement technique capable of removing the visual effect of haze, bypassing the need for estimating transmission and airlight. However, there is underlying information in Koschmieder’s model that can be useful to understand the kind of solution we should expect to obtain. To see this, let us consider an input hazy image I(x ) with intensity values varying in [0, 1]. Then, any solution J(x ) to the image dehazing problem needs to contain intensity values lower than I(x ). This can be shown by

t(x ) = A − I(x ) A − J(x ) . Since t (x ) ∈ [0, 1], it follows that A − I(x ) ≤ A − J(x ), and it can be concluded that J(x ) ≤ I(x )∀x . According to the above observation, the technique introduced proposes to make use of the information present in a set of over-exposed versions E = {I 1 (x ),I 2 (x ), ...,I n (x )} of the original hazy image I(x ). Underexposing I(x ) will always lead to the presence of decreased intensities. However, when I(x ) is globally underexposed, not every region on it contains useful information, since insufficient exposure will darken I(x ) too much. For this reason, all images in E are fused by means of a simple and efficient multiple-exposure fusion strategy relying on a Laplacian pyramid decomposition of the set of over-exposed images. The resulting image is a haze-free version of

I(x ).

**FUSION BASED DEHAZING**

The fusion method takes two inputs derived from the original image. The first input I1 is obtained by white balancing the original hazy image. The second input I2 is selected in order to increase the contrast in hazy regions. In this approach this is obtained automatically by subtracting from the original image I the average luminance value of the entire image I~(I bar).

Luminance weight map manages the luminance gain in the output image. This map computes the standard deviation between every R,G and B color channels and each pixel luminance L of the input. This overcomes the degradation induced by I2 in the haze-free regions ensuring a seamless transition between the inputs I1, I2. This map also tends to reduce the global contrast and colorfulness. However, these effects are overcome by deſning two additional weights: chromatic (colorfulness) and saliency (global contrast). Chromatic weight map controls the saturation gain in the output image. To obtain this map, for every pixel is computed the distance between its saturation value S and the maximum of the saturation range using a Gauss curve: d = exp (−(S−Smax)^2/ 2σ^2) with a standard deviation σ = 0.3.

Saliency weight map identiſes the degree of conspicuousness with respect to the neighborhood regions.

In the fusion process, the inputs are weighted by speciſc computed maps in order to conserve the most signiſcant detected features. Each pixel (i, j) of the output F is computed by summing the inputs Ik weighted by corresponding normalized weight maps :



where l represents the number of the pyramid levels and L {I} is the Laplacian version of the input I while G {W¯} represents the Gaussian version of the normalized weight map of the W¯.

**PART 2**

Initially, an input video stream is loaded into the memory. A python script breaks down the video into a certain number of frames. Now a concept called Mutual information is used to create a lookup table.

Mutual information between two sets measures how much can be known from one set if only the other set is known. Given a set of values $ A=\{a_i\} $. Its entropy $ H(A) $ is defined by $ H(A) = \sum_i^n{- p(a_i) \log({p(a_i)})} $ where $ p(a_i) $ are the probabilities of the values in the set. Entropy can be interpreted as a measure of the mean uncertainty reduction that is obtained when one of the particular values is found during sampling. Given two sets $ A=\{a_i\} $ and $ B=\{b_i\} $ its joint entropy is given by the joint probabilities $ p_(a_i,b_i) $ as $ H(A,B) = \sum_i^n{-p(a_i,b_i) * log( p(a_i, b_i) )} $. Mutual information is obtained by subtracting the entropy of both sets from the joint entropy, as:$ H(A,B)-H(A)-H(B) $, and indicates how much uncertainty about one set is reduced by the knowledge of the second set. Mutual information is the metric of choice when an image from different modalities need to be registered.

Now that the frames are loaded into the memory this concept is applied on each frame and a certain value is filled in the lookup table.

When the dehazing algorithm is applied to the images, if the corresponding value is with a certain boundary (threshold values) then that frame is not dehazed, instead the max(prev frame’s lookup, next frame’s lookup) is used.

This would significantly reduce the amount of time taken for dehazing. And the metric used to compare the output of the algorithm to the ground truth is Normalized correlation which is Correlation between intensity values divided by the square rooted autocorrelation of both target and reference objects: $ \frac{\sum_i^n{a_i * b_i }}{\sum_i^n{a_i^2}\sum_i^n{b_i^2}} $. This metric allows registering objects whose intensity values are related by a linear transformation.

**Codes**

Part 1

Single image defogging:

from \_\_future\_\_ import division

import cv2

import numpy as np

class Channel\_value:

val = -1.0

intensity = -1.0

def find\_intensity\_of\_atmospheric\_light(img, gray):

top\_num = int(img.shape[0] \* img.shape[1] \* 0.001)

toplist = [Channel\_value()] \* top\_num

dark\_channel = find\_dark\_channel(img)

for y in xrange(img.shape[0]):

for x in xrange(img.shape[1]):

val = img.item(y, x, dark\_channel)

intensity = gray.item(y, x)

for t in toplist:

if t.val < val or (t.val == val and t.intensity < intensity):

t.val = val

t.intensity = intensity

break

max\_channel = Channel\_value()

for t in toplist:

if t.intensity > max\_channel.intensity:

max\_channel = t

return max\_channel.intensity

def find\_dark\_channel(img):

return np.unravel\_index(np.argmin(img), img.shape)[2]

def clamp(minimum, x, maximum):

return max(minimum, min(x, maximum))

def dehaze(img, light\_intensity, windowSize, t0, w):

size = (img.shape[0], img.shape[1])

outimg = np.zeros(img.shape, img.dtype)

for y in xrange(size[0]):

for x in xrange(size[1]):

x\_low = max(x-(windowSize//2), 0)

y\_low = max(y-(windowSize//2), 0)

x\_high = min(x+(windowSize//2), size[1])

y\_high = min(y+(windowSize//2), size[0])

sliceimg = img[y\_low:y\_high, x\_low:x\_high]

dark\_channel = find\_dark\_channel(sliceimg)

t = 1.0 - (w \* img.item(y, x, dark\_channel) / light\_intensity)

outimg.itemset((y,x,0), clamp(0, ((img.item(y,x,0) - light\_intensity) / max(t, t0) + light\_intensity), 255))

outimg.itemset((y,x,1), clamp(0, ((img.item(y,x,1) - light\_intensity) / max(t, t0) + light\_intensity), 255))

outimg.itemset((y,x,2), clamp(0, ((img.item(y,x,2) - light\_intensity) / max(t, t0) + light\_intensity), 255))

return outimg

imageName = raw\_input() # eg. fg5.jpg

img = cv2.imread(imageName)

cv2.namedWindow("result", cv2.CV\_WINDOW\_AUTOSIZE)

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

light\_intensity = find\_intensity\_of\_atmospheric\_light(img, gray)

w = 0.95

t0 = 0.55

outimg = dehaze(img, light\_intensity, 20, t0, w)

name = imageName+'\_out\_t0'+str(t0)+'\_w'+str(w)+'.jpg'

print name

cv2.imwrite(name, outimg)

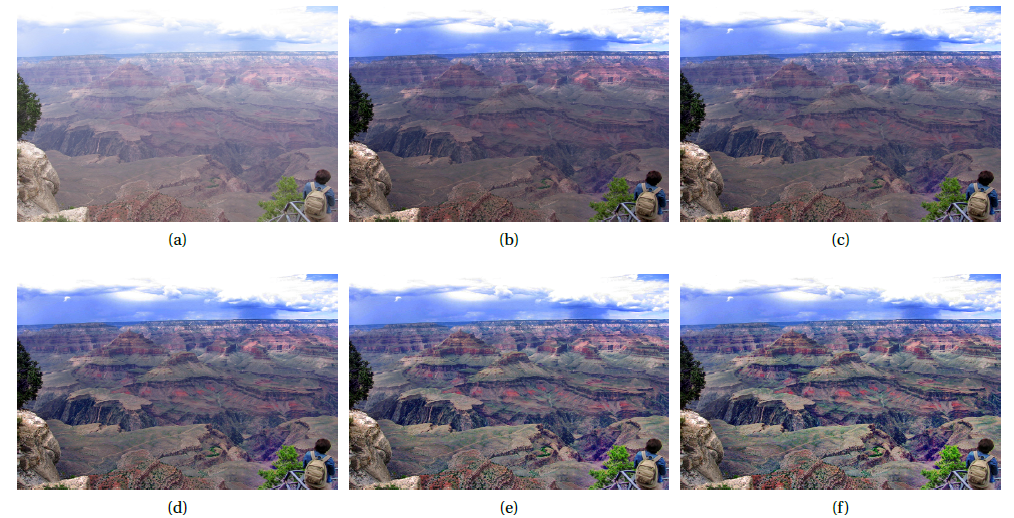
Result:



**AME**

The code can be found at: <https://github.com/rkat7/AME>

Result:



**Dehazing by Fusion**

Code:

|  |  |
| --- | --- |
|  | import numpy as np |
|  | import matplotlib.pyplot as plt |
|  | import cv2 as cv |
|  |  |
|  | # skimage imports |
|  | from skimage.util import img\_as\_ubyte, img\_as\_float64 |
|  | from skimage.color import rgb2gray |
|  | from skimage.color import rgb2hsv |
|  |  |
|  | class ImageDehazing: |
|  | def \_\_init\_\_(self, verbose=False): |
|  | '''Function to initialize class variables''' |
|  | self.image = None |
|  | self.verbose = verbose |
|  |  |
|  | def \_\_clip(self, image=None): |
|  | '''Function to clip images to range of [0.0, 1.0]''' |
|  | # Validate parameters |
|  | if image is None: |
|  | return None |
|  |  |
|  | image[image < 0] = 0 |
|  | image[image > 1] = 1 |
|  | return image |
|  |  |
|  | def \_\_show(self, images=None, titles=None, size=None, gray=False): |
|  | '''Function to display images''' |
|  | # Validate parameters |
|  | if images is None or titles is None or size is None: |
|  | return |
|  |  |
|  | plt.figure(figsize=size) |
|  |  |
|  | plt.subplot(1, 2, 1) |
|  | if gray is True: |
|  | plt.imshow(images[0], cmap='gray') |
|  | else: |
|  | plt.imshow(images[0]) |
|  | plt.title(titles[0]) |
|  | plt.axis('off') |
|  |  |
|  | plt.subplot(1, 2, 2) |
|  | if gray is True: |
|  | plt.imshow(images[1], cmap='gray') |
|  | else: |
|  | plt.imshow(images[1]) |
|  | plt.title(titles[1]) |
|  | plt.axis('off') |
|  |  |
|  | plt.show() |
|  |  |
|  | def white\_balance(self, image=None): |
|  | '''Function to perform white balancing operation on an image''' |
|  | # Validate parameters |
|  | if image is None: |
|  | return None |
|  |  |
|  | image = img\_as\_float64(image) |
|  |  |
|  | # Extract colour channels |
|  | R = image[:, :, 2] |
|  | G = image[:, :, 1] |
|  | B = image[:, :, 0] |
|  |  |
|  | # Obtain average intensity for each colour channel |
|  | mean\_R = np.mean(R) |
|  | mean\_G = np.mean(G) |
|  | mean\_B = np.mean(B) |
|  |  |
|  | mean\_RGB = np.array([mean\_R, mean\_G, mean\_B]) |
|  |  |
|  | # Obtain scaling factor |
|  | grayscale = np.mean(mean\_RGB) |
|  | scale = grayscale / mean\_RGB |
|  |  |
|  | white\_balanced = np.zeros(image.shape) |
|  |  |
|  | # Rescale original intensities |
|  | white\_balanced[:, :, 2] = scale[0] \* R |
|  | white\_balanced[:, :, 1] = scale[1] \* G |
|  | white\_balanced[:, :, 0] = scale[2] \* B |
|  |  |
|  | # Clip to [0.0, 1.0] |
|  | white\_balanced = self.\_\_clip(white\_balanced) |
|  |  |
|  | # Display result (if verbose) |
|  | if self.verbose is True: |
|  | self.\_\_show( |
|  | images=[self.image, white\_balanced], |
|  | titles=['Original Image', 'White Balanced Image'], |
|  | size=(15, 15) |
|  | ) |
|  | return white\_balanced |
|  |  |
|  | def enhance\_contrast(self, image=None): |
|  | '''Function to enhance contrast in an image''' |
|  | # Validate parameters |
|  | if image is None: |
|  | return None |
|  |  |
|  | image = img\_as\_float64(image) |
|  |  |
|  | # Extract colour channels |
|  | R = image[:, :, 2] |
|  | G = image[:, :, 1] |
|  | B = image[:, :, 0] |
|  |  |
|  | # Obtain luminance using predefined scale factors |
|  | luminance = 0.299 \* R + 0.587 \* G + 0.114 \* B |
|  | mean\_luminance = np.mean(luminance) |
|  |  |
|  | # Compute scale factor |
|  | gamma = 2 \* (0.5 + mean\_luminance) |
|  |  |
|  | # Scale mean-luminance subtracted colour chanels |
|  | enhanced = np.zeros(image.shape) |
|  | enhanced[:, :, 2] = gamma \* (R - mean\_luminance) |
|  | enhanced[:, :, 1] = gamma \* (G - mean\_luminance) |
|  | enhanced[:, :, 0] = gamma \* (B - mean\_luminance) |
|  |  |
|  | # Clip to [0.0, 1.0] |
|  | enhanced = self.\_\_clip(enhanced) |
|  |  |
|  | # Display result (if verbose) |
|  | if self.verbose is True: |
|  | self.\_\_show( |
|  | images=[self.image, enhanced], |
|  | titles=['Original Image', 'Contrast Enhanced Image'], |
|  | size=(15, 15) |
|  | ) |
|  |  |
|  | return enhanced |
|  |  |
|  | def luminance\_map(self, image=None): |
|  | '''Function to generate the Luminance Weight Map of an image''' |
|  | # Validate parameters |
|  | if image is None: |
|  | return None |
|  |  |
|  | image = img\_as\_float64(image) |
|  |  |
|  | # Generate Luminance Map |
|  | luminance = np.mean(image, axis=2) |
|  | luminancemap = np.sqrt((1 / 3) \* (np.square(image[:, :, 0] - luminance + np.square(image[:, :, 1] - luminance) + np.square(image[:, :, 2] - luminance)))) |
|  |  |
|  | # Display result (if verbose) |
|  | if self.verbose is True: |
|  | self.\_\_show( |
|  | images=[self.image, luminancemap], |
|  | titles=['Original Image', 'Luminanace Weight Map'], |
|  | size=(15, 15), |
|  | gray=True |
|  | ) |
|  | return luminancemap |
|  |  |
|  | def chromatic\_map(self, image=None): |
|  | '''Function to generate the Chromatic Weight Map of an image''' |
|  | # Validate parameters |
|  | if image is None: |
|  | return None |
|  |  |
|  | image = img\_as\_float64(image) |
|  |  |
|  | # Convert to HSV colour space |
|  | hsv = rgb2hsv(image) |
|  |  |
|  | # Extract Saturation |
|  | saturation = hsv[:, :, 1] |
|  | max\_saturation = 1.0 |
|  | sigma = 0.3 |
|  |  |
|  | # Generate Chromatic Map |
|  | chromaticmap = np.exp(-1 \* (((saturation - max\_saturation) \*\* 2) / (2 \* (sigma \*\* 2)))) |
|  |  |
|  | # Display result (if verbose) |
|  | if self.verbose is True: |
|  | self.\_\_show( |
|  | images=[self.image, chromaticmap], |
|  | titles=['Original Image', 'Chromatic Weight Map'], |
|  | size=(15, 15), |
|  | gray=True |
|  | ) |
|  |  |
|  | return chromaticmap |
|  |  |
|  | def saliency\_map(self, image=None): |
|  | '''Function to generate the Saliency Weight Map of an image''' |
|  | # Validate parameters |
|  | if image is None: |
|  | return None |
|  |  |
|  | image = img\_as\_float64(image) |
|  |  |
|  | # Convert image to grayscale |
|  | if(len(image.shape) > 2): |
|  | image = rgb2gray(image) |
|  | else: |
|  | image = image |
|  |  |
|  | # Apply Gaussian Smoothing |
|  | gaussian = cv.GaussianBlur(image,(5, 5),0) |
|  |  |
|  | # Apply Mean Smoothing |
|  | image\_mean = np.mean(image) |
|  |  |
|  | # Generate Saliency Map |
|  | saliencymap = np.absolute(gaussian - image\_mean) |
|  |  |
|  | # Display result (if verbose) |
|  | if self.verbose is True: |
|  | self.\_\_show( |
|  | images=[self.image, saliencymap], |
|  | titles=['Original Image', 'Saliency Weight Map'], |
|  | size=(15, 15), |
|  | gray=True |
|  | ) |
|  |  |
|  | return saliencymap |
|  |  |
|  | def image\_pyramid(self, image=None, pyramid\_type='gaussian', levels=1): |
|  | '''Function to generate the Gaussian/Laplacian pyramid of an image''' |
|  | # Validate parameters |
|  | if image is None: |
|  | return None |
|  |  |
|  | image = img\_as\_float64(image) |
|  |  |
|  | # Generate Gaussian Pyramid |
|  | current\_layer = image |
|  | gaussian = [current\_layer] |
|  | for i in range(levels): |
|  | current\_layer = cv.pyrDown(current\_layer) |
|  | gaussian.append(current\_layer) |
|  |  |
|  | if pyramid\_type == 'gaussian': |
|  | return gaussian |
|  | # Generate Laplacian Pyramid |
|  | elif pyramid\_type == 'laplacian': |
|  | current\_layer = gaussian[levels-1] |
|  | laplacian = [current\_layer] |
|  | for i in range(levels - 1, 0, -1): |
|  | shape = (gaussian[i-1].shape[1], gaussian[i-1].shape[0]) |
|  | expand\_gaussian = cv.pyrUp(gaussian[i], dstsize=shape) |
|  | current\_layer = cv.subtract(gaussian[i-1], expand\_gaussian) |
|  | laplacian.append(current\_layer) |
|  | laplacian.reverse() |
|  | return laplacian |
|  |  |
|  | def fusion(self, inputs=None, weights=None, gaussians=None): |
|  | '''Function to fuse the pyramids together''' |
|  | # Validate parameters |
|  | if inputs is None or weights is None or gaussians is None: |
|  | return None |
|  |  |
|  | fused\_levels = [] |
|  |  |
|  | # Perform Fusion by combining the Laplacian and Gaussian pyramids |
|  | for i in range(len(gaussians[0])): |
|  | if len(inputs[0].shape) > 2: |
|  | for j in range(inputs[0].shape[2]): |
|  | # Generate Laplacian Pyramids |
|  | laplacians = [ |
|  | self.image\_pyramid(image=inputs[0][:, :, j], pyramid\_type='laplacian', levels=len(gaussians[0])), |
|  | self.image\_pyramid(image=inputs[1][:, :, j], pyramid\_type='laplacian', levels=len(gaussians[0])) |
|  | ] |
|  |  |
|  | # Adjust rows to match |
|  | row\_size = np.min(np.array([ |
|  | laplacians[0][i].shape[0], |
|  | laplacians[1][i].shape[0], |
|  | gaussians[0][i].shape[0], |
|  | gaussians[1][i].shape[0] |
|  | ])) |
|  |  |
|  | # Adjust columns to match |
|  | col\_size = np.min(np.array([ |
|  | laplacians[0][i].shape[1], |
|  | laplacians[1][i].shape[1], |
|  | gaussians[0][i].shape[1], |
|  | gaussians[1][i].shape[1] |
|  | ])) |
|  |  |
|  | if j == 0: |
|  | intermediate = np.ones(inputs[0][:row\_size, :col\_size].shape) |
|  | # Fusion Step |
|  | intermediate[:, :, j] = (laplacians[0][i][:row\_size, :col\_size] \* gaussians[0][i][:row\_size, :col\_size]) + (laplacians[1][i][:row\_size, :col\_size] \* gaussians[1][i][:row\_size, :col\_size]) |
|  | fused\_levels.append(intermediate) |
|  |  |
|  | # Reconstruct Image Pyramids |
|  | for i in range(len(fused\_levels)-2, -1, -1): |
|  | level\_1 = cv.pyrUp(fused\_levels[i+1]) |
|  | level\_2 = fused\_levels[i] |
|  | r = min(level\_1.shape[0], level\_2.shape[0]) |
|  | c = min(level\_1.shape[1], level\_2.shape[1]) |
|  | fused\_levels[i] = level\_1[:r, :c] + level\_2[:r, :c] |
|  |  |
|  | # Clip fused image to [0.0, 1.0] |
|  | fused = self.\_\_clip(fused\_levels[0]) |
|  | if self.verbose is True: |
|  | self.\_\_show( |
|  | images=[self.image, fused], |
|  | titles=['Original Image', 'Fusion'], |
|  | size=(15, 15), |
|  | gray=False |
|  | ) |
|  | return fused |
|  |  |
|  | def dehaze(self, image=None, verbose=None, pyramid\_height=12): |
|  | '''Driver function to dehaze the image''' |
|  | # Validate parameters |
|  | if image is None: |
|  | return None |
|  |  |
|  | self.image = image |
|  |  |
|  | if len(image.shape) > 2 and image.shape[2] == 4: |
|  | self.image = image[:, :, :3] |
|  |  |
|  | # Set verbose flag (to decide whether each step is displayed) |
|  | if verbose is None: |
|  | pass |
|  | elif verbose is True: |
|  | self.verbose = True |
|  | else: |
|  | self.verbose = False |
|  |  |
|  | # Generating Input Images |
|  | white\_balanced = self.white\_balance(image=img\_as\_float64(self.image)) # First Input Image |
|  | contrast\_enhanced = self.enhance\_contrast(image=img\_as\_float64(self.image)) # Second Input Image |
|  |  |
|  | input\_images = [ |
|  | img\_as\_float64(white\_balanced), |
|  | img\_as\_float64(contrast\_enhanced) |
|  | ] |
|  |  |
|  | # Generating Weight Maps |
|  | weight\_maps = [ |
|  | # Weight maps for first image |
|  | { |
|  | 'luminance': self.luminance\_map(image=input\_images[0]), |
|  | 'chromatic': self.chromatic\_map(image=input\_images[0]), |
|  | 'saliency': self.saliency\_map(image=input\_images[0]) |
|  | }, |
|  |  |
|  | # Weight maps for second image |
|  | { |
|  | 'luminance': self.luminance\_map(image=input\_images[1]), |
|  | 'chromatic': self.chromatic\_map(image=input\_images[1]), |
|  | 'saliency': self.saliency\_map(image=input\_images[1]) |
|  | } |
|  | ] |
|  |  |
|  | # Weight map normalization |
|  | # Combined weight maps |
|  | weight\_maps[0]['combined'] = (weight\_maps[0]['luminance'] \* weight\_maps[0]['chromatic'] \* weight\_maps[0]['saliency']) |
|  | weight\_maps[1]['combined'] = (weight\_maps[1]['luminance'] \* weight\_maps[1]['chromatic'] \* weight\_maps[1]['saliency']) |
|  |  |
|  | # Normalized weight maps |
|  | weight\_maps[0]['normalized'] = weight\_maps[0]['combined'] / (weight\_maps[0]['combined'] + weight\_maps[1]['combined']) |
|  | weight\_maps[1]['normalized'] = weight\_maps[1]['combined'] / (weight\_maps[0]['combined'] + weight\_maps[1]['combined']) |
|  |  |
|  | # Generating Gaussian Image Pyramids |
|  | gaussians = [ |
|  | self.image\_pyramid(image=weight\_maps[0]['normalized'], pyramid\_type='gaussian', levels=pyramid\_height), |
|  | self.image\_pyramid(image=weight\_maps[1]['normalized'], pyramid\_type='gaussian', levels=pyramid\_height) |
|  | ] |
|  |  |
|  | # Fusion Step |
|  | fused = self.fusion(input\_images, weight\_maps, gaussians) |
|  |  |
|  | # Dehazing data |
|  | dehazing = { |
|  | 'hazed': self.image, |
|  | 'inputs': input\_images, |
|  | 'maps': weight\_maps, |
|  | 'dehazed': fused |
|  | } |
|  |  |
|  | self.image = None # Reset image |
|  |  |
|  | return dehazing |

Result:



PART 2

Converting video to frames:

import cv2

import numpy as np

import os

# Create a VideoCapture object and read from input file

cap = cv2.VideoCapture(r'C:\Users\TEMP\Desktop\figdp\481479901 ([online-video-cutter.com](http://online-video-cutter.com/)) (1).mp4')

# Check if camera opened successfully

if (cap.isOpened()== False):

print("Error opening video file")

# Read until video is completed

count=1

while(cap.isOpened()):

# Capture frame-by-frame

ret, frame = cap.read()

if ret == True:

# Display the resulting frame

print('Read %d frame: ' % count, ret)

cv2.imwrite(os.path.join(r'C:\Users\TEMP\Desktop\video12', "frame{:d}.jpg".format(count)), frame)

count += 1

# Break the loop

else:

break

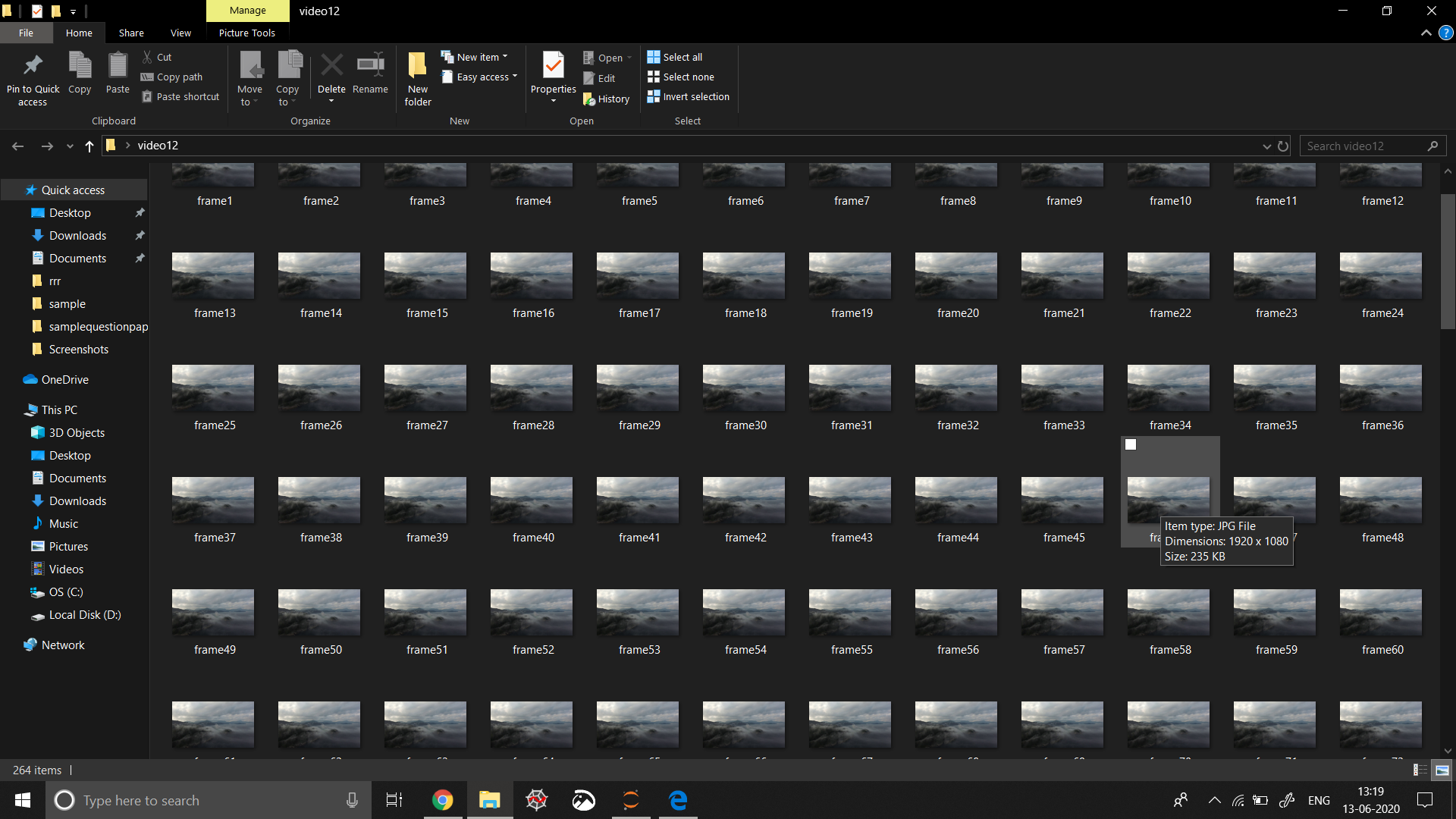
# When everything done, release

# the video capture object

cap.release()

# Closes all the frames

cv2.destroyAllWindows()



(frames split from video)

Converting Frames back to video:

import os

import cv2

from PIL import Image

# Checking the current directory path

print(os.getcwd())

# Folder which contains all the images

# from which video is to be generated

os.chdir(r"C:\Users\TEMP\Desktop\fog\SingleImageHazeRemover\sample\genvideo")

path = r"C:\Users\TEMP\Desktop\fog\SingleImageHazeRemover\sample\genvideo"

mean\_height = 0

mean\_width = 0

num\_of\_images = len(os.listdir('.'))

# print(num\_of\_images)

for file in os.listdir('.'):

im = Image.open(os.path.join(path, file))

width, height = im.size

mean\_width += width

mean\_height += height

# im.show() # uncomment this for displaying the image

# Finding the mean height and width of all images.

# This is required because the video frame needs import os

import cv2

from PIL import Image

# Checking the current directory path

print(os.getcwd())

# Folder which contains all the images

# from which video is to be generated

os.chdir(r"C:\Users\TEMP\Desktop\fog\SingleImageHazeRemover\sample\genvideo")

path = r"C:\Users\TEMP\Desktop\fog\SingleImageHazeRemover\sample\genvideo"

mean\_height = 0

mean\_width = 0

num\_of\_images = len(os.listdir('.'))

# print(num\_of\_images)

for file in os.listdir('.'):

im = Image.open(os.path.join(path, file))

width, height = im.size

mean\_width += width

mean\_height += height

# im.show() # uncomment this for displaying the image

# Finding the mean height and width of all images.

# This is required because the video frame needs

# to be set with same width and height. Otherwise

# images not equal to that width height will not get

# embedded into the video

mean\_width = int(mean\_width / num\_of\_images)

mean\_height = int(mean\_height / num\_of\_images)

# print(mean\_height)

# print(mean\_width)

# Resizing of the images to give

# them same width and height

for file in os.listdir('.'):

if file.endswith(".jpg") or file.endswith(".jpeg") or file.endswith("png"):

# opening image using PIL Image

im = Image.open(os.path.join(path, file))

# im.size includes the height and width of image

width, height = im.size

print(width, height)

# resizing

imResize = im.resize((mean\_width, mean\_height), Image.ANTIALIAS)

imResize.save( file, 'JPEG', quality = 95) # setting quality

# printing each resized image name

print(im.filename.split('\\')[-1], " is resized")

# Video Generating function

def generate\_video():

image\_folder = # make sure to use your folder

video\_name = 'mygeneratedvideo.avi'

os.chdir(r"C:\Users\TEMP\Desktop\fog\SingleImageHazeRemover\sample\genvideo")

images = [img for img in os.listdir(image\_folder)

if img.endswith(".jpg") or

img.endswith(".jpeg") or

img.endswith("png")]

# Array images should only consider

# the image files ignoring others if any

print(images)

frame = cv2.imread(os.path.join(image\_folder, images[0]))

# setting the frame width, height width

# the width, height of first image

height, width, layers = frame.shape

video = cv2.VideoWriter(video\_name, 0, 1, (width, height))

# Appending the images to the video one by one

for image in images:

video.write(cv2.imread(os.path.join(image\_folder, image)))

# Deallocating memories taken for window creation

cv2.destroyAllWindows()

video.release() # releasing the video generated

# Calling the generate\_video function

generate\_video()

# to be set with same width and height. Otherwise

# images not equal to that width height will not get

# embedded into the video

mean\_width = int(mean\_width / num\_of\_images)

mean\_height = int(mean\_height / num\_of\_images)

# print(mean\_height)

# print(mean\_width)

# Resizing of the images to give

# them same width and height

for file in os.listdir('.'):

if file.endswith(".jpg") or file.endswith(".jpeg") or file.endswith("png"):

# opening image using PIL Image

im = Image.open(os.path.join(path, file))

# im.size includes the height and width of image

width, height = im.size

print(width, height)

# resizing

imResize = im.resize((mean\_width, mean\_height), Image.ANTIALIAS)

imResize.save( file, 'JPEG', quality = 95) # setting quality

# printing each resized image name

print(im.filename.split('\\')[-1], " is resized")

# Video Generating function

def generate\_video():

image\_folder = # make sure to use your folder

video\_name = 'mygeneratedvideo.avi'

os.chdir(r"C:\Users\TEMP\Desktop\fog\SingleImageHazeRemover\sample\genvideo")

images = [img for img in os.listdir(image\_folder)

if img.endswith(".jpg") or

img.endswith(".jpeg") or

img.endswith("png")]

# Array images should only consider

# the image files ignoring others if any

print(images)

frame = cv2.imread(os.path.join(image\_folder, images[0]))

# setting the frame width, height width

# the width, height of first image

height, width, layers = frame.shape

video = cv2.VideoWriter(video\_name, 0, 1, (width, height))

# Appending the images to the video one by one

for image in images:

video.write(cv2.imread(os.path.join(image\_folder, image)))

# Deallocating memories taken for window creation

cv2.destroyAllWindows()

video.release() # releasing the video generated

# Calling the generate\_video function

generate\_video()

**Conclusion**

The Artificial Multiple Exposure technique can be used for images with sky, or nature ,or a lot of greenery involved.

The Single image defogging using Dark channel prior is best when applied on images with objects and less of blank space.

The Multi scale Fusion technique best applies when the input is buildings or high contrasts are involved as in the pictures of trees or greenery.

Depth map based model is best suited for satellite imagery.

Apart from the above observations, a novel way of dehazing image is proposed in this project. The findings of this project and the new proposal can be used as a foundation for further research in this field to make the algorithm and simpler and effective in all the types of outputs, i.e., a general technique which is effective for all kinds of image inputs. Also we plan on researching this topic as our capstone project for final year in undergraduation.

**REFERENCES**

* Ancuti, C. O., Ancuti, C., & Bekaert, P. (2010). *Effective single image dehazing by fusion. 2010 IEEE International Conference on Image Processing.* doi:10.1109/icip.2010.5651263
* Zhiming Tan, et.al, Fast Single image defogging
* Bolun Cai, et.al, DehazeNet: An End-to-End System for Single Image Haze Removal
* G. Dal Moro and L. Halounova, “Haze removal and data calibration for high-resolution satellite data,” Int. Journal of Remote Sensing, 2006.
* <https://www.researchgate.net/post/What_are_the_metrics_available_to_Compare_the_Image_Quality>