MAE 473/573 Computer Vision and Image Processing Report 1

Ben Cutri

PN: 50225383

Instructor: Dr. Junsong Yuan

Fall 2021

OpenCV Version: 3.4.2.17

1. Image Stitching

The goal of this task is to stitch two images (“left.jpg” and “right.jpg”) together to construct a panorama image. A python script called “task1.py” was provided to code and call a function called “solution(left_img, right_img)”. The required tasks for this section are explained in detail and shown below:

(i) Extract Keypoint:

This task involved finding the keypoints of an image as well as extracting the features for these key points. SIFT Detection and description methods were chosen for the key points and the end result is a matrix of keypoints, as seen in the section of code below:

```
# create SIFT object
sift = cv2.xfeatures2d.SIFT_create()

# detect SIFT features in both images
keypoints_L, descriptors_L = sift.detectAndCompute(left_img, None)
keypoints_R, descriptors_R = sift.detectAndCompute(right_img, None)
```

The code creates a keypoint detector and a keypoint descriptor for each feature (for the left and right image).

(ii) Match Keypoint:

For this task, the keypoints were matched by comparing their feature distance using KNN. Ratio testing was used to find an acceptable distance between the features. See the code below:

```
# calculating and ranking feature distances for each keypoint
# right matching
m_R = []
md_R=[]

for i in range(len(descriptors_R)):
  y = np.asarray(descriptors_L)
  x = np.asarray(descriptors_R[i])
  m_R.append(np.linalg.norm(y-x,axis=1))
  c1 = c2 = float('inf')

  for j in m_R[i]:
    if j <= c1:
      c1, c2 = j, c1
    elif j < c2:
      c2 = j
  md_R.append((c1, c2))

# cross checking the match to ensure the match is good (symmetric)
# left matching
m_L = []
md_L=[]

for i in range(len(descriptors_L)):
  y = np.asarray(descriptors_R)
  x = np.asarray(descriptors_L[i])
  m_L.append(np.linalg.norm(y-x, axis=1))
  c1 = c2 = float('inf')

  for j in m_L[i]:
    if j <= c1:
      c1, c2 = j, c1
    elif j < c2:
      c2 = j
  md_L.append((c1, c2))
```

```
# left index
ratio = 0.75
index_L=[]

for i in range(len(md_L)):
  y = md_L[i][0]/md_L[i][1]
  if y < ratio:
    index_L.append(i)

# right index
index_R=[]

for i in range(len(md_R)):
  y = md_R[i][0]/md_R[i][1]
  if y < ratio:
    index_R.append(i)

# using keypoints
fkp_L = []
for i in index_L:
    fkp_L.append(keypoints_L[i])

fkp_R = []
for i in index_R:
    fkp_R.append(keypoints_R[i])

r_pts = cv2.KeyPoint_convert(fkp_R)
l_pts = cv2.KeyPoint_convert(fkp_L)

H = ransacHomography(l_pts, r_pts)
```

(ii) Estimate Homography:

The homography matrix was created using RANSAC algorithms and solving the least squares method. The code for creating and utilizing the homography matrix is shown below:

```
def generateRandom(SER_Pts, DES_Pts, N):
  r = np.random.choice(len(SER_Pts), N)
  DES = [DES_Pts[i] for i in r]
  SER = [SER_Pts[i] for i in r]
  return np.asarray(SER, dtype=np.float32), np.asarray(DES, dtype=np.float32)


def ransacHomography(SER_Pts, dst_Pts):
    max_LDES = []
    max_LSER = []
    max_I = 0
    finalH = None

    for i in range(6000):
        SERP, DESP = generateRandom(SER_Pts, dst_Pts, 4)
        H = findH(SERP, DESP)
        inlines = 0
        line_DES = []
        line_SER = []

        for p_1, p_2 in zip(SER_Pts, dst_Pts):
            p_1U = (np.append(p_1, 1)).reshape(3, 1)
            p_2e = H.dot(p_1U)
            p_2e = (p_2e / p_2e[2])[:2].reshape(1, 2)[0]

            if np.linalg.norm(p_2 - p_2e) < 5:
                inlines += 1
                line_DES.append(p_2)
                line_SER.append(p_1)

        if inlines > max_I:
            max_I = inlines
            print(max_I)
            max_LDES = line_DES.copy()
            max_LDES = np.asarray(max_LDES, dtype=np.float32)
            max_LSER = line_SER.copy()
            max_LSER = np.asarray(max_LSER, dtype=np.float32)
            H_final = H

        if max_I > inlines:
            break

    return H_final
```

(iv) Warp & Stitch Images:

Once the homography matrix was created, the images were sitched together to create a uniform image. The maximum and minimum x and y coordinates were also calculated, as seen in the code below:

```
rows_1, cols_1 = left_img.shape[:2]
rows_2, cols_2 = right_img.shape[:2]

corners_1 = np.float32([[0,0], [0,rows_1], [cols_1, rows_1], [cols_1,0]]).reshape(-1,1,2)
corners_2 = np.float32([[0,0], [0,rows_2], [cols_2, rows_2], [cols_2,0]]).reshape(-1,1,2)

perspective_corners_1 = cv2.perspectiveTransform(corners_1, H)
corners = np.concatenate((corners_2, perspective_corners_1), axis=0)

[x_min, y_min] = np.int32(corners.min(axis=0).ravel())
[x_max, y_max] = np.int32(corners.max(axis=0).ravel())

TY = -y_min
TX = -x_min
T = np.asarray([[1,0,TX],[0,1,TY],[0,0,1]])
H_T = np.dot(T,H)
im_out = cv2.warpPerspective(left_img,H_T, (x_max-x_min,y_max-y_min))

im_out[TY:TY+rows_2,TX:TX+cols_2] = right_img
```

2. Grayscale Image Processing

The goal of this task was to utilize two different forms of image processing (image denoising and edge detection) to analyze and manipulate a digital image. A python script called “task2.py” was provided to code and call several functions. The required tasks for this section are explained in detail and shown below:

(i) Denoise:

To denoise the provided image, a 3x3 median filter was applied across the image. A function was created called “filter(img)” that takes a given image, applies the filter with zero padding (to maintain its size), and outputs a denoised image. The code for this function can be seen below:

```
def filter(img):
    denoise_img = np.zeros((img.shape))
    padded = np.pad(array=img,pad_width=1,mode='constant')

    for i in range(padded.shape[0]-2):
        for j in range(padded.shape[1]-2):
            img_window = padded[i:(i+3),j:(j+3)]
            y = np.sort(img_window)
            median = np.median(y)
            denoise_img[i][j] = median

    return denoise_img
```

The function creates a “denoise_img” matrix full of zeros and pads the provided image with zeros. Then for each 3x3 pixel location (i and j) the function sorts the values, finds the median, and fills it into the zdenoise_img matrix. The final output is a denoised image called “denoise_img” (which has now had a median filter applied to it) which can be seen below:

(ii) Edge Detect:

Two different functions were created to detect the edges within the image. First, a convolution function called “convolve2d(img, kernel)” was coded to flip the kernel and apply it to the provided image (applying convolution to the image). The code for this function is seen below:

```
def convolve2d(img, kernel):
    kernel = np.flipud(np.fliplr(kernel))
    padded = np.pad(array=img,pad_width=1,mode='constant')
    conv_img = np.zeros((img.shape))

    for i in range(padded.shape[0]-2):
        for j in range(padded.shape[1]-2):
            img_window = padded[i:(i+3),j:(j+3)]
            conv_img[i][j] = np.multiply(img_window, kernel).sum()

    return conv_img
```

The function convolve2d takes the given kernel and flips it vertically and horizontally as well as pads the provided images with zeros (to maintain size). Then for each pixel, the kernel is multiplied with a 3x3 image window to transform the image through convolution. The final output is a fully convoluted image called “conv_img”.

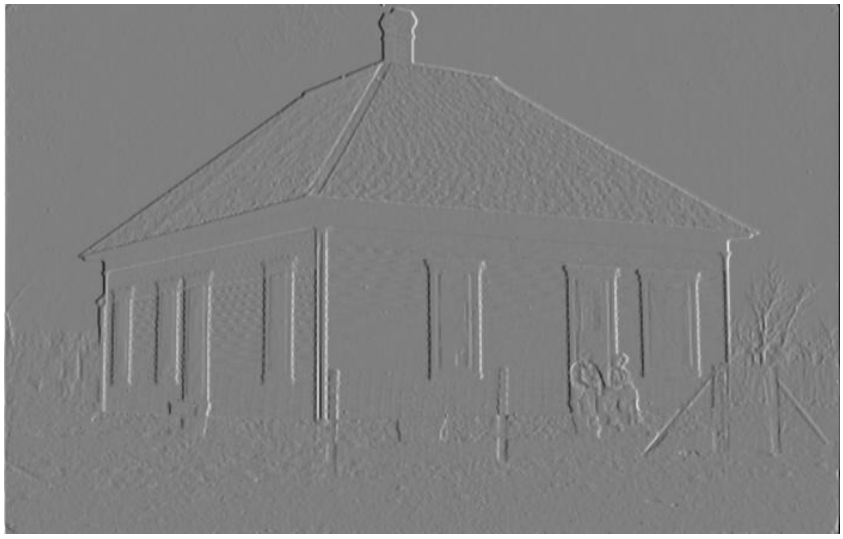
Next, based on the “convolve2d(img, kernel)” function, an edge detection function called “edge_detect(img)” was built to find the edges of the image using provided Sobel operators. The code for this is seen below:

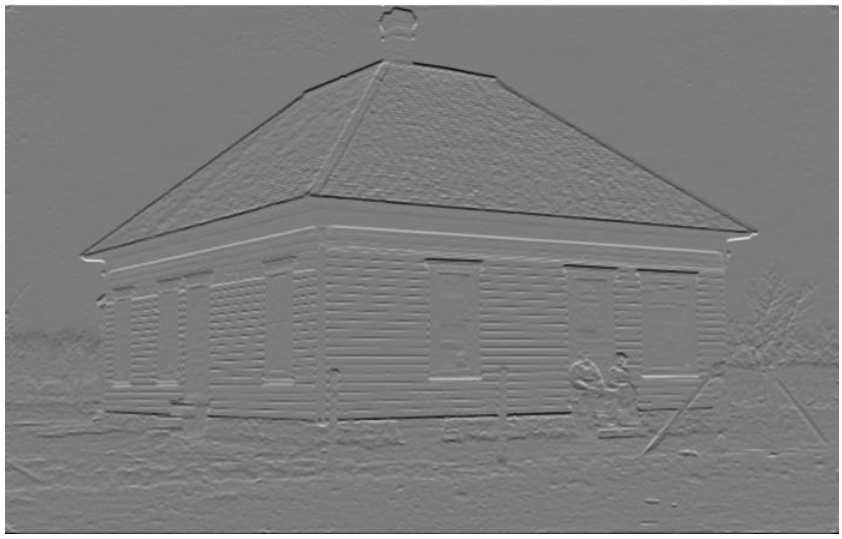
```
def edge_detect(img):
    edge_y = convolve2d(img,sobel_y)
    edge_x = convolve2d(img,sobel_x)
    edge_mag = np.sqrt(np.square(edge_x) + np.square(edge_y))

    edge_y = 255 * (edge_y - np.min(edge_y)) / (np.max(edge_y) - np.min(edge_y))
    edge_x = 255 * (edge_x - np.min(edge_x)) / (np.max(edge_x) - np.min(edge_x))
    edge_mag = 255 * (edge_mag - np.min(edge_mag)) / (np.max(edge_mag) - np.min(edge_mag))

    return edge_y, edge_x, edge_mag
```

The function edge_detect takes the provided image and sobel operators and applies convolve2d to create two images called edge_y and edge_x. Then, an image called edge_mag is created using the formula for magnitude Then, those images are normalized to maintain their range and contrast using the provided normalization formula (so the max pixel value = 255 and the min pixel value = 0). The final output is 3 images (edge_y, edge_x, and edge_mag) that show the edges in the x/y direction as well as an image showing both of their magnitudes as seen below:





(iii) Diagonal Edge:

This task called for the design of two 3x3 kernels that could detect the diagonal edges of the image along both the 45° and 135° directions. The code and final kernels matrices are shown below:

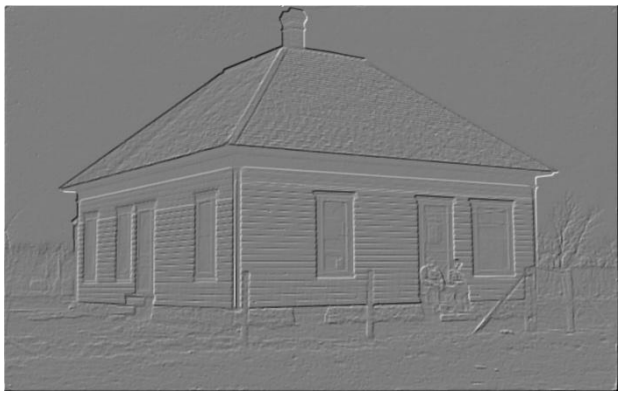
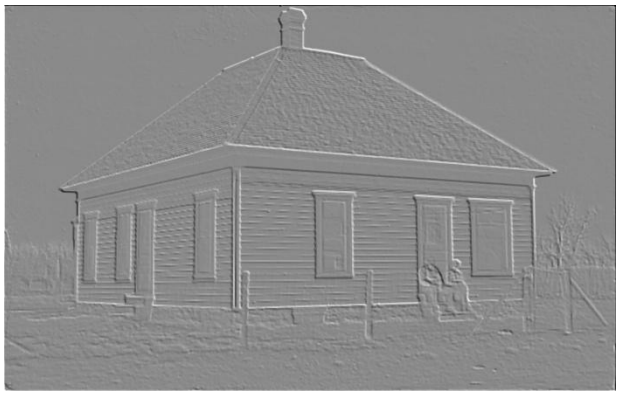
```
def edge_diag(img):
    sobel_135 = np.array([[0, -1, -2], [1, 0, -1], [2, 1, 0]]).astype(int)
    sobel_45 = np.array([[2, 1,0], [1,0, -1], [0, -1, -2]]).astype(int)

    edge_135 = convolve2d(img,sobel_135)
    edge_45 = convolve2d(img,sobel_45)

    edge_135 = 255 * (edge_135 - np.min(edge_135)) / (np.max(edge_135) - np.min(edge_135))
    edge_45 = 255 * (edge_45 - np.min(edge_45)) / (np.max(edge_45) - np.min(edge_45))

    print(sobel_135)
    print(sobel_45)
    return edge_135, edge_45
```

The function edge_diag defines two new kernels (sobel_135 and sobel_45) which convolute the image in the 135° and 45° directions respectively. Then similarly to the previous edge detect function, the images are normalized and the final outputs (edge_135 and edge_45) are seen below:

3. Morphology Image Processing

The goal of this task was to utilize morphological methods of image processing (erode, dilate, open, and close) to remove noises from the provided image. A python script called "task3.py" was provided to code and call several functions. The two required tasks for this section are explained in detail and shown below:

(i) Morphological Denoise:

This task called for the creation of four functions ("morph erode(img)", "morph dilate(img)", "morph open(img)" and "morph close(img)") to remove noises from the provided image. A 3x3 matrix of ones was used for the structuring element and the result was a function called "denoise(img)" that runs the operations and returns a denoised binary image. The code for each of these functions is shown below:

```
def morph_erode(img):
    img = np.where(img==0,img,1)
    erd_img = np.zeros((img.shape))
    kernel = np.ones((3,3),np.int32)
    padded = np.pad(array=img,pad_width=1,mode='constant')

    for i in range(padded.shape[1]):
        for j in range(padded.shape[0]):
            img_window = padded[i:(i+3),j:(j+3)]
            if np.array_equal(img_window,kernel):
                erd_img[i][j] = 1

    return erd_img
```

The function morph_erode creates an erd_img matrix of all zeros, defines the structuring element (kernel) as a matrix of all ones, and pads the provided image with zeros. Then, for each pixel value, a 3x3 image window around each pixel is compared to the kernel. If the values are equal, then a one is stored into the erd_img matrix. The result is an eroded image called erd_img.

```
def morph_dilate(img):
    img = np.where(img==0,img,1)
    dil_img = np.zeros((img.shape))
    padded = np.pad(array=img,pad_width=1,mode='constant')

    for i in range(img.shape[1]):
        for j in range(img.shape[0]):
            img_win = img[i:(i+3),j:(j+3)]
            if 1 in img_win:
                dil_img[i][j] = 1

    return dil_img
```

The function morph_dilate creates a dil_img matrix of all zeros and pads the provided image with zeros. Then, for each pixel value, a 3x3 image window is constructed and if a value of one is found in the window, then a one is stored into the dil_img matrix. The result is a dilated image called dil_img.

```
def morph_open(img):
    erd_img = morph_erode(img)
    img_open = morph_dilate(erd_img)

    return img_open
```

```
def morph_close(img):
    dil_img = morph_dilate(img)
    img_close = morph_erode(dil_img)

    return img_close
```

The functions morph_open and morph_close take a provided image and perform erosion and dilation operations on the image (erosion then dilation for morph_open, verse dilation then erosion for morph_close). The result is im_open and img_close for the open and closed functions respectively.

```
def denoise(img):
    img_open = morph_open(img)
    denoise = morph_close(img_open)
    denoise = np.where(denoise==0,denoise,255)

    return denoise
```

The function denoise calls all of the previous functions by applying an open morph operation on the provided image followed by a closed morph operation. Then, the images dtype is converted back to uint8/int and the resulting output is a fully denoised image, which is shown below:

(ii) Morphological Denoise:

Using the previously define morphological operations, the boundary was extracted from the denoised image. A function was implemented called “boundary_img”, which can be seen below:

```
def boundary(img):
    erd_img = morph_erode(img)
    bound_img = img - erd_img

    return bound_img
```

The function boundary_img takes the provided image and erodes it using the morph_erode function as specified above. The boundary is then calculated by taking the provided image and subtracting it from the erd_img. The result is the boundary, an image called bound_img which is shown below: