

IMVFX HW1

1. Explanation for all parts of your code

```
def knn_matting(image, trimap, my_lambda=100):  
    [h, w, c] = image.shape  
    image = image.astype(np.float32) / 255.0  
    trimap = trimap / 255.0  
    foreground = (trimap == 1.0).astype(int)  
    background = (trimap == 0.0).astype(int)  
  
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)  
    h_channel = hsv[:, :, 0]  
    s_channel = hsv[:, :, 1]  
    v_channel = hsv[:, :, 2]  
  
    cos_h = np.cos(h_channel * 2 * np.pi)  
    sin_h = np.sin(h_channel * 2 * np.pi)  
    x_coord = np.repeat(np.arange(h)[:, np.newaxis], w, axis=1)  
    y_coord = np.repeat(np.arange(w)[np.newaxis, :], h, axis=0)  
  
    feature_vector = np.stack([cos_h, sin_h, s_channel, v_channel, x_coord, y_coord], axis=-1)  
    feature_vector = feature_vector.reshape(-1, 6)
```

def knn_matting 這個函式，可以接收三個參數 image、trimap 和一個可選的 my_lambda，預設值為 100。首先將 image 和 trimap 進行數值範圍轉換，使像素值在 [0, 1] 範圍內。然後根據 trimap 創建了 foreground 和 background，接下來將 image 轉換為 HSV color space，並分離出 HSV channel，再計算了特徵向量，其中包括 H channel 的 cosine 和 sine 值，S channel、V channel 以及像素的 x 和 y 座標。

```
knn = sklearn.neighbors.NearestNeighbors(n_neighbors=10, n_jobs=4).fit(feature_vector)  
distances, indices = knn.kneighbors(feature_vector)  
  
C = np.max(distances)  
weights = 1 - distances / C  
rows = np.repeat(np.arange(h * w), 10)  
A = scipy.sparse.coo_matrix((weights.ravel(), (rows, indices.ravel()))), shape=(h * w, h * w))
```

接下來對每個像素找到了 K(預設 K=10)個 Nearest Neighbor，使用 K-nearest neighbors 模型，再使用距離計算每個像素與其最近鄰居之間的相似性，建立了 affinity matrix A。

```

D = scipy.sparse.diags(np.ravel(A.sum(axis=1)))
L = D - A

marked = (foreground + background).astype(float).ravel()
M = scipy.sparse.diags(marked)
v = foreground.ravel().astype(float)

alpha = cg(L + my_lambda * M, my_lambda * v)[0]
alpha = alpha.reshape(h, w)

return alpha

```

之後我們根據 affinity matrix 計算了 Laplace matrix L ，這一步是 matting 算法中創建 linear system 的重要部分。在建立 linear system 後，用 sparse matrix 表示已 mark 的 foreground 和 background，並且用 Conjugate Gradient 解這個 linear system 來找到最佳的 alpha 值。最後這個 alpha 矩陣被重塑回原始圖像的形狀，這個 alpha 矩陣就代表了每個像素的透明度值。

```

if __name__ == '__main__':
    start_time = time.time()
    image = cv2.imread('./image/bear.png')
    trimap = cv2.imread('./trimap/bear.png', cv2.IMREAD_GRAYSCALE)

    alpha = knn_matting(image, trimap)
    alpha = alpha[:, :, np.newaxis]
    foreground = (image * alpha).astype(np.uint8)

    background = cv2.imread('sky.png')
    background = cv2.resize(background, (image.shape[1], image.shape[0]))
    result = alpha * image + (1 - alpha) * background

    cv2.imwrite('./result/bear.png', result)
    end_time = time.time()
    execution_time = end_time - start_time
    print(f"執行時間：{execution_time} 秒")

```

在主函式中首先讀取原始圖片和 trimap 圖片，然後用 knn_matting 函數來計算 alpha 矩陣。接著利用 alpha 矩陣將原始 foreground 與一個新的 background 合成，創建出一張看起來像是 foreground 在新 background 前的圖片。最後，這個合成圖片被保存下來，並且 print 出所有過程所需的時間。

2. Description on what you have done in experiments and show the results

在這次實驗中，我利用 knn matting 將圖 1、圖 2、圖 3 的背景更換成圖 4，合成新圖片圖 5、圖 6、圖 7。



圖 1



圖 2

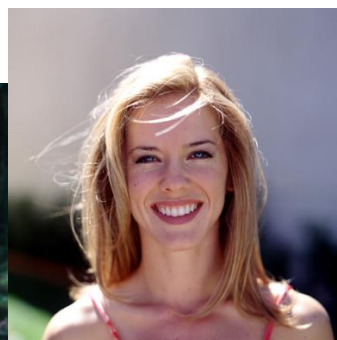


圖 3

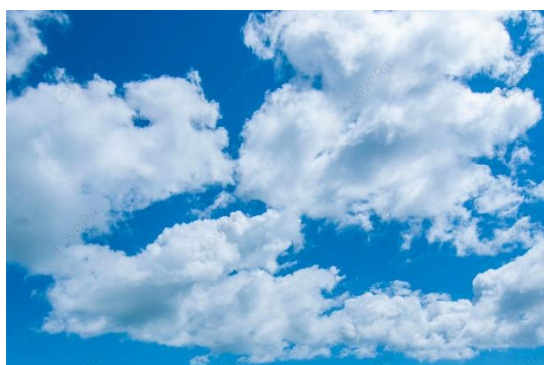


圖 4



圖 5





圖 6



圖 7

下面兩張圖皆是 $\lambda=100$ ，然後用不同 number of K 去合成的圖跟執行時間

| number of K=10 | number of K=50 |
|---|--|
|  |  |

```
C:\Users\user\release_hw1>python release.py  
執行時間：0.487821102142334 秒
```

```
C:\Users\user\release_hw1>python release.py  
執行時間：1.071518898010254 秒
```

上面兩張圖看起來差異不大，可是因為是不同 number of K 去合成的圖，所以執行時間 number of K=50 會大於 number of K=10。

3. Description on what you have done for the bonus

- a. Solving for a large linear system is extremely time-consuming, so find some other methods to speed it up and make a comparison on how long do these methods take.

```
alpha= spsolve(L + my_lambda * M, my_lambda * v)
```

在第 1 部分的程式碼，我是用 Conjugate Gradient(scipy.sparse.linalg.cg)來解線性系統，這個方法它不是透過直接計算來求解整個方程組，而是透過迭代過程逐漸逼近解的方法，會比直接求解快很多，缺點是可能沒有非常精確的解。在這個部分我改使用 scipy.sparse.linalg.spsolve 這個直接求解的方法，去比較跟 Conjugate Gradient 解法的成效跟計算時間



| scipy.sparse.linalg.cg | scipy.sparse.linalg.spsolve |
|--|--|
|  |  |
| <pre>C:\Users\user\release_hw1>python release.py 執行時間：0.487821102142334 秒</pre> | <pre>執行時間：22.654871940612793 秒</pre> |

上面兩張圖皆是在 lambda=100、number of K=10 用不同解線性系統的方法合成的圖片，圖片看起來沒有差很多，但是執行時間用 Conjugate Gradient 解線性系統，會比用 spsolve 直接求解的快很多。

- b. Try different ways of representing feature vectors

```
r_channel = image[:, :, 2]  
g_channel = image[:, :, 1]  
b_channel = image[:, :, 0]  
  
feature_vector = np.stack([r_channel, g_channel, b_channel], axis=-1)  
feature_vector = feature_vector.reshape(-1, 3)
```


feature vectors 從 HSV color space+ spatial coordinate 改成用 RGB color space，上圖為 feature vectors 是 RGB color space 的程式碼。

| HSV color space + spatial coordinate | RGB color space |
|---|--|
|  |  |

上面兩張圖皆是在 $\lambda=100$ 、number of $K=10$ 以不同 color space feature vectors 合成的圖片，很明顯可以看出 HSV color space + spatial coordinate 的合成圖片比 RGB color space 的合成圖片好。

c. Work on more images







d. Implement KNN by myself

```
def find_knn(feature_vectors, k):  
    num_samples = feature_vectors.shape[0]  
    indices = np.zeros((num_samples, k), dtype=int)  
    k_distances = np.full((num_samples, k), np.inf)  
  
    for i in range(num_samples):  
        diffs = feature_vectors - feature_vectors[i]  
        dists = np.linalg.norm(diffs, axis=1)  
        dists[i] = np.inf  
  
        partitioned_indices = np.argpartition(dists, k)[:k]  
        closest_dists = dists[partitioned_indices]  
  
        sorted_indices = np.argsort(closest_dists)  
        indices[i, :] = partitioned_indices[sorted_indices]  
        k_distances[i, :] = closest_dists[sorted_indices]  
  
    return k_distances, indices
```

以下解釋上圖的 code:

先通過 `feature_vectors.shape[0]` 取得特徵向量的數量，即圖片中有多少像素點需要進行 Nearest Neighbor Search。然後建兩個空的 numpy 數組 `indices` 和 `k_distances`，分別用來存儲每個點的 `k` 個 Nearest Neighbor 的索引和對應的距離，並將距離初始化為無限大表示還未計算。接下來使用一個 `for` 迴圈遍歷每一個點，計算該點與圖片中所有其他點的差值，然後用歐幾里得距離公式計算出距離。為了避免點自己成為自己的 Neighbor，將自身的距離設置為無限大。之後使用 `np.argpartition` 函數找出這些距離中最小的 `k` 個值的索引。然後根據這些索引，提取並排序這些最近的距離，最終將排序好的索引和距離存入之前創建的 `indices` 和 `k_distances`。這樣就完成了對每一個點的 `k` 個 Nearest Neighbor Search，最後返回這兩個數組。

| 使用 sklearn 的 knn | Implement KNN by myself |
|---|--|
|  |  |

```
C:\Users\user\release_hw1>python release.py  
執行時間：0.47243595123291016 秒
```

執行時間：662.7870011329651 秒

上面表格是合成後的結果跟執行時間，上面兩張圖皆是在 $\lambda=100$ 、number of $K=10$ 的狀態下合成的，單看結果跟直接用 sklearn 的 knn 沒甚麼差別(就是 Description on what you have done in experiments and show the results 這部分的結果)，可是用 sklearn 的 knn 的執行時間只要 0.4724 秒，用上面程式碼的執行時間卻要 662.787 秒，明顯比用 sklearn 的 knn 慢很多。

e. Compare with other matting methods

```
def grabcut_matting(image, trimap):  
  
    mask = np.where((trimap == 255), cv2.GC_FGD,  
                    np.where((trimap == 0), cv2.GC_BGD,  
                              cv2.GC_PR_BGD)).astype('uint8')  
  
    bgdModel = np.zeros((1, 65), np.float64)  
    fgdModel = np.zeros((1, 65), np.float64)  
  
    mask, bgdModel, fgdModel = cv2.grabCut(image, mask, None, bgdModel, fgdModel, iterCount=5, mode=cv2.GC_INIT_WITH_MASK)  
  
    alpha = np.where((mask == cv2.GC_FGD) | (mask == cv2.GC_PR_FGD), 1, 0).astype('float32')  
  
    return alpha
```

我這部分使用 GrabCut 去跟 knn matting 做比較，以下是上圖 code 的解釋：grabcut_matting 函式接收 image 和 trimap 作為輸入。在這個函式首先根據 trimap 生成一個 mask，這個 mask 標記了圖片中的前景（使用 cv2.GC_FGD）、背景（使用 cv2.GC_BGD）以及可能的前景/背景（使用 cv2.GC_PR_BGD）。這樣的初始化告訴 GrabCut 算法哪部分是絕對的前景、哪部分是絕對的背景，哪部分還未確定。接下來為 GrabCut 算法創建了前景和背景模型，這些模型是以圖片像素分佈為基礎的 Gaussian Mixture Model，用來估計像素屬於前景或背景的機率。然後用 cv2.grabCut 函數，將圖片、初始化的 mask 和前景/背景模型作為輸入，並執行 GrabCut 算法。在 code 中，指定了 iterCount=5，這意思是 GrabCut 會進行五次迭代來改善前景和背景的分割。cv2.grabCut 執行完成後，會根據最終的 mask 生成 alpha。在 alpha 中，前景像素的 alpha 值被設置為 1（完全不透明），背景像素的 alpha 值被設置為 0（完全透明），這樣可以通過與原圖相乘來獲取 foreground。

| | |
|--------------------------------|---------|
| knn matting (K=10, lambda=100) | GrabCut |
|--------------------------------|---------|



從上面兩張圖可以看出來 knn matting 合成後的圖片，在人物邊緣比 GrabCut 合成後的圖片還要滑順。