Computer Vision HW1

R08942065 楊宗山

- **How to design joint bilateral filter**

A. Initiate the variables in the function

```
r = int(3 * self.sigma_s)
kernel = int(2 * r + 1)
guidance = guidance / 255
output = np.zeros(input.shape)
```

B. Padding the guidance and original picture with `cv2.copyMakeBorder()`.
Should be careful here, the padding algorithm is different from `np.pad(mode='reflect')`

```
# padding
input = cv2.copyMakeBorder(input, r, r, r, r,cv2.BORDER_REFLECT)
guidance = cv2.copyMakeBorder(guidance, r, r, r, r,cv2.BORDER_REFLECT)
if len(guidance.shape) == 2:
    guidance = np.expand_dims(guidance, axis=2)
```

C. Construct G_s filter which is independent of the position of the kernel, and the distance of the grid is calculated by the `np.meshgrid()` function.

```
# bulid G_s (independent to guidance)
bound = np.arange(2 * r + 1) - r   # [-r ... 0 ... r]
x_coor, y_coor = np.meshgrid(bound, -bound.T)
G_s = np.square(x_coor) + np.square(y_coor)
G_s = np.exp(-G_s / (2 * np.square(self.sigma_s)))
```

D. The convolution part is directly conducted.
G_r filter is made from a copy of the guidance picture with the same size of kernel.
The convolution is implemented with `np.multiply()`, and the mathematic forms are the same as the homework pdf file.

```
# brute force conv
for row in range(output.shape[0]):
    for col in range(output.shape[1]):
        # bulid G_r (depends on guidance)
        G_r = guidance[row:row+kernel, col:col+kernel, :] - guidance[row+r, col+r, :]
        G_r = np.sum(np.square(G_r), axis=2)
        G_r = np.exp(-G_r / (2 * np.square(self.sigma_r)))
        # make window on input image
        window = input[row:row+kernel, col:col+kernel, :]
        # convolution
        filter = np.multiply(G_s, G_r)
        denominator = np.sum(filter)
        output[row, col, 0] = np.sum(np.multiply(filter, window[:, :, 0])) / denominator
        output[row, col, 1] = np.sum(np.multiply(filter, window[:, :, 1])) / denominator
        output[row, col, 2] = np.sum(np.multiply(filter, window[:, :, 2])) / denominator
```

- **How to implement local minima selection**

  A. construct a `dict` object to save the cost of each weight

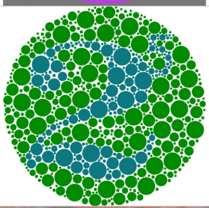
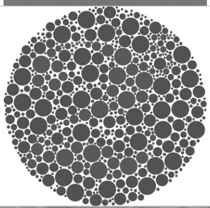
```python
    costmap = {} # make costmap of 66 points
    cnt = 0
    JBF = Joint_bilateral_filter(sigma_s, sigma_r, border_type='reflect')
    for w_r in range(11):
        for w_g in range(11 - w_r):
            print('Process {} points\r'.format(cnt), end='')
            w_b = 10 - w_r - w_g
            img_gray = rgb2gray(img, w_r/10., w_g/10., w_b/10.)
            img_out = JBF.joint_bilateral_filter(img, img_gray).astype(np.uint8)
            costmap['{},{},{}'.format(w_r, w_g, w_b)] = cost_fn(img, img_out)
```

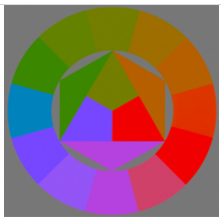
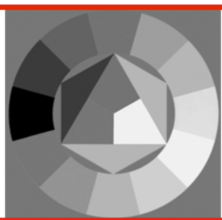
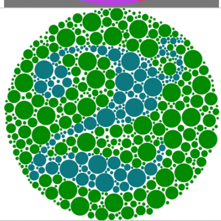
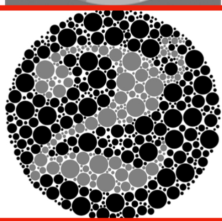
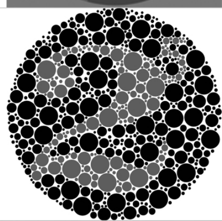
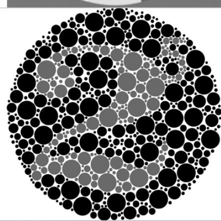
  B. brutal force search for each point in the cost map

```python
def lower_than_neighbor(w_r, w_g, w_b, costmap, key):
    if w_r > 10 or w_g > 10 or w_b > 10:
        return True
    elif w_r < 0 or w_g < 0 or w_b < 0:
        return True
    return costmap[key] < costmap['{},{},{}'.format(w_r, w_g, w_b)]

def check_min(costmap, key):
    w_r = int(key.split(',')[0])
    w_g = int(key.split(',')[1])
    w_b = int(key.split(',')[2])
    return lower_than_neighbor(w_r-1, w_g+1, w_b, costmap, key)\
           and lower_than_neighbor(w_r-1, w_g, w_b+1, costmap, key)\
           and lower_than_neighbor(w_r+1, w_g-1, w_b, costmap, key)\
           and lower_than_neighbor(w_r, w_g-1, w_b+1, costmap, key)\
           and lower_than_neighbor(w_r+1, w_g, w_b-1, costmap, key)\
           and lower_than_neighbor(w_r, w_g+1, w_b-1, costmap, key)
```

- **Results**

Original Picture	Conventional RGB2GRAY
	
	
	

Original Picture	Advanced RGB2GRAY		
			
			
			

2a.png: **(0.0, 1.0, 0.0)**, (1.0, 0.0, 0.0), (0.0, 0.0, 1.0)
2b.png: **(1.0, 0.0, 0.0)**, (0.8, 0.0, 0.2), (0.6, 0.0, 0.4)
2c.png: **(1.0, 0.0, 0.0)**, (0.2, 0.2, 0.6), (0.0, 0.3, 0.7)