

* **Binarization**

Binary

import numpy as np

from PIL import Image

# Load the image as a grayscale image

I = np.array(Image.open("yjsp.jpg").convert('L'))

# Set the threshold value

t = 127

# Apply the thresholding operation

Inew = np.where(I > t, 255, 0).astype('uint8')

# Display the binary image

Image.fromarray(Inew).show()



This code performs image binarization, which is a process of converting a grayscale image into a binary image. The binary image has only two possible pixel values, typically 0 and 255, which represent black and white, respectively.

The code first loads an image named "yjsp.jpg" as a grayscale image using the PIL library. It then sets a threshold value of 127, which is used to determine whether a pixel should be black or white in the binary image. Pixels with intensity values greater than the threshold value are set to 255 (white), while pixels with intensity values less than or equal to the threshold value are set to 0 (black). Finally, the binary image is displayed using the PIL library.

Overall, this code demonstrates a simple but effective technique for image binarization.

Double threshold Binary

import cv2

# Load the image

I = cv2.imread("yjsp.jpg",cv2.IMREAD\_COLOR)

# Set the threshold values

t1 = 127

t2 = 200

# Convert the image to grayscale

Igray = cv2.cvtColor(I, cv2.COLOR\_BGR2GRAY)

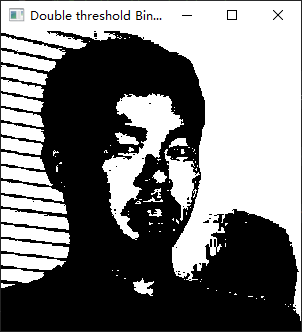
# Apply double threshold binary

ret, Inew = cv2.threshold(Igray, t1, 255,cv2.THRESH\_BINARY)

# Display the image

cv2.imshow( "Double threshold Binary" , Inew)

cv2.waitKey(0)



This code performs double threshold binary on an image. It first loads an image using OpenCV's imread function. Then, it sets two threshold values, t1 and t2. The image is converted to grayscale using cv2.cvtColor function. Finally, double threshold binary is applied using cv2.threshold function with the grayscale image and the two threshold values as inputs. The resulting binary image is displayed using cv2.imshow function. The program waits for a key press before closing the window.

Binarization by adaptive method

import cv2

# Read the image file "yjsp.jpg" in color mode using the imread() function from OpenCV.

I = cv2.imread("yjsp.jpg", cv2.IMREAD\_COLOR)

# Convert the color image "I" to grayscale using the cvtColor() function from OpenCV.

Igray = cv2.cvtColor(I, cv2.COLOR\_BGR2GRAY)

# Apply adaptive thresholding to the grayscale image "Igray" using the adaptiveThreshold() function from OpenCV.

Inew = cv2.adaptiveThreshold(Igray, 255,

                             cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C,

                             cv2.THRESH\_BINARY, 11, 2)

# Display the binary image using the imshow() function from OpenCV.

cv2.imshow( "Binarization by adaptive method" , Inew)

# Wait for a key press before closing the window.

cv2.waitKey(0)



This code reads an image file named "yjsp.jpg" in color mode using the imread() function from OpenCV. Then, it converts the color image to grayscale using the cvtColor() function from OpenCV. Finally, it applies adaptive thresholding to the grayscale image using the adaptiveThreshold() function from OpenCV. The resulting binary image is displayed using the imshow() function from OpenCV, and the program waits for a key press before closing the window.

Binarization by the Otsu method

# This code reads an image and applies the Otsu method for binarization

import cv2

# Reading the image in grayscale

I = cv2.imread("yjsp.jpg", cv2.IMREAD\_GRAYSCALE)

# Applying the Otsu method for binarization

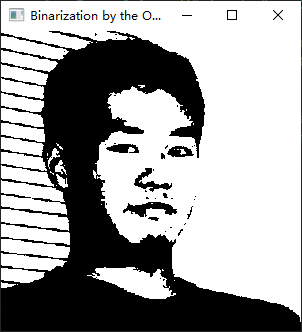
ret, Inew = cv2.threshold(I, 0, 255,

                          cv2.THRESH\_OTSU)

# Displaying the binarized image

cv2.imshow( "Binarization by the Otsu method" , Inew)

cv2.waitKey(0)



This code reads an image in grayscale using OpenCV's cv2.imread() function. Then, it applies the Otsu method for binarization using cv2.threshold() function. The Otsu method is a thresholding technique that automatically calculates a threshold value from image histogram for a bimodal image. Finally, it displays the binarized image using cv2.imshow() function and waits for a key event using cv2.waitKey() function.

* **Segmentation**

SkinColorSegmentation

# Importing necessary libraries

import numpy as np

import cv2

# Defining function for skin color segmentation

def SkinColorSegmentation(imname):

  # Reading image

  img = cv2.imread(imname, cv2.IMREAD\_COLOR)

  # Converting image to HSV color space

  hsv = cv2.cvtColor(img, cv2.COLOR\_BGR2HSV)

  (\_h, \_s, \_v) = cv2.split(hsv)

  # Creating an array of zeros with the same shape as \_h

  skin3 = np.zeros(\_h.shape, dtype=np.uint8)

  (x, y) = \_h.shape

  # Looping through each pixel of the image

  for i in range(0, x):

    for j in range(0, y):

      # Checking if the pixel is within the skin color range

      if (\_h[i][j] > 7) and (\_h[i][j] < 20) and (\_s[i][j] > 28) and (\_s[i][j] < 255) and (\_v[i][j] > 50) and (

        \_v[i][j] < 255):

        # Setting the pixel value to 255 if it is within the skin color range

        skin3[i][j] = 255

      else:

        # Setting the pixel value to 0 if it is not within the skin color range

        skin3[i][j] = 0

  # Displaying the original image and the skin color segmented image

  cv2.imshow(imname, img)

  cv2.imshow(imname + " Skin3 HSV", skin3)

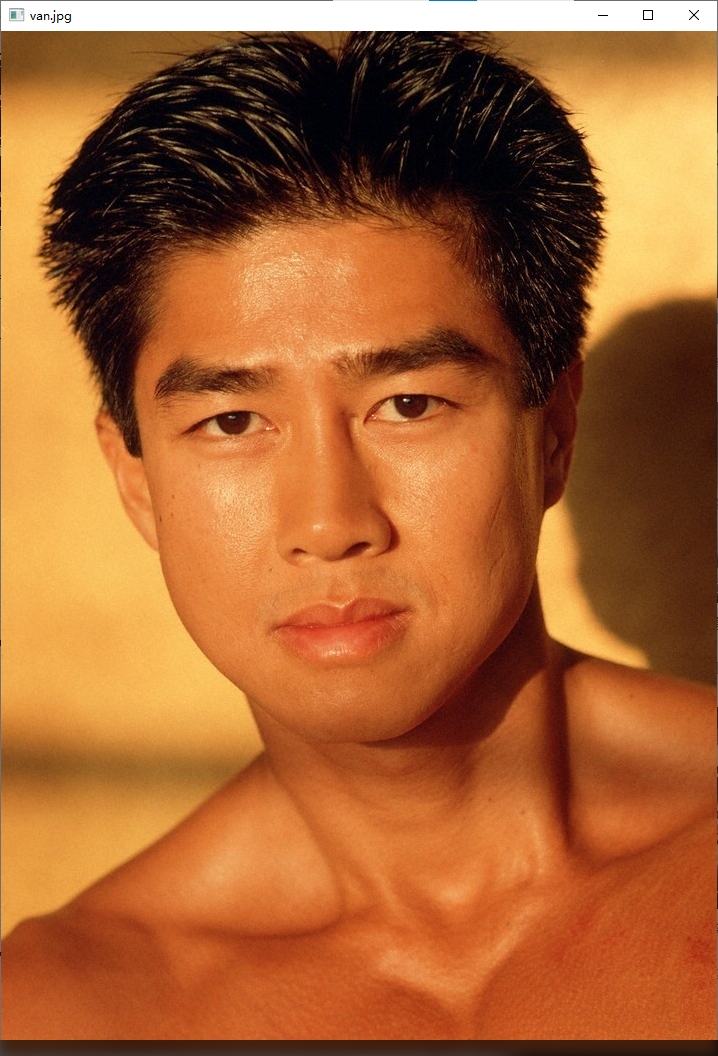
# Running the function on the given image

if \_\_name\_\_=="\_\_main\_\_":

  imname="van.jpg"

  SkinColorSegmentation(imname)

  cv2.waitKey(0)





This code is for skin color segmentation of an image. It reads an image and converts it to the HSV color space. Then it loops through each pixel of the image and checks if the pixel is within the skin color range. If the pixel is within the skin color range, it sets the pixel value to 255, otherwise it sets the pixel value to 0. Finally, it displays the original image and the skin color segmented image.

WeberSegmentation

import cv2

import numpy as np

# Define the Weber function

def weber(i):

  if i<0:

    return 0

  if i>255:

    return 255

  if i<=88:

    return int(20-12\*i/88)

  if i<=138:

    return int(0.002\*(i-88)\*(i-88))

  return int(7\*(i-138)/117+13)

# Define the Weber Segmentation function

def WeberSegmentation(fn= "yjsp.jpg",fn\_out=None):

  # Read the image

  I=cv2.imread(fn,cv2.IMREAD\_COLOR)

  if not isinstance(I,np.ndarray) or I.data==None:

    print("Error reading file \"{}\"".format(fn))

    exit()

  # Show the source image

  cv2.imshow("Source",I)

  # Convert the image to grayscale

  Igray=cv2.cvtColor(I,cv2.COLOR\_BGR2GRAY)

  cv2.imshow("Grayscale",Igray)

  # Initialize the Weber image and the Weber2 image

  Iweber =np.zeros\_like(Igray)

  Iweber2=np.zeros\_like(Igray)

  n=1

  # Perform Weber segmentation

  while(Iweber==0).any():

    Imin=Igray[Iweber==0].min()

    Iw=weber(Imin)

    n=n+1

    mask=np.logical\_and(Igray >=Imin,Igray<=Imin+Iw)

    Iweber[mask]=n

    Iweber2[mask]=Imin

  n=n-1

  Iweber=Iweber-1

  # Show the Weber segmentation image in JET colormap

  cv2.imshow("Weber segmentation JET",cv2.applyColorMap((Iweber.astype(np.float32)\*255/(n+1)).astype(np.uint8),cv2.COLORMAP\_JET))

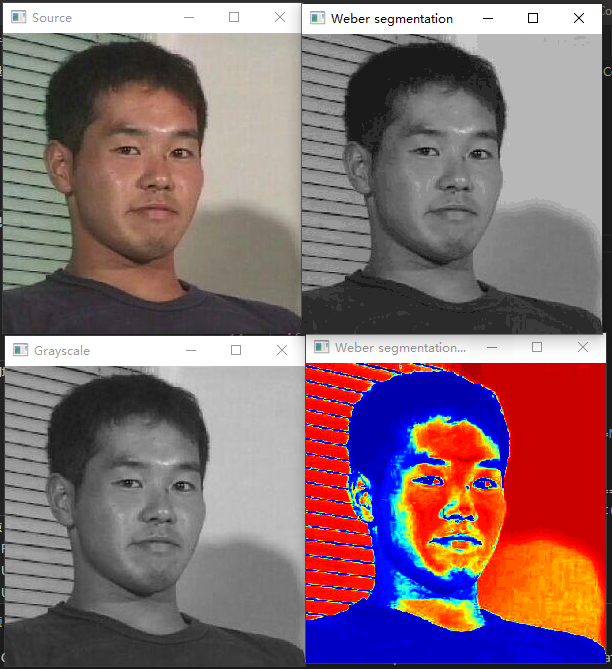
  # Show the Weber segmentation image

  cv2.imshow("Weber segmentation",Iweber2)

  cv2.waitKey(0)

# Call the Weber Segmentation function

WeberSegmentation("yjsp.jpg")



This code is an implementation of Weber Segmentation algorithm. It reads an image, converts it to grayscale, and then performs Weber segmentation on it. The Weber function is defined to calculate the Weber value of a pixel intensity. The Weber value is used to segment the image. The output of the segmentation is shown in two windows: one with the Weber segmentation image in JET colormap and the other with the Weber segmentation image.

* **Segmentation2**

import cv2 as cv

import numpy as np

##########################################################################

## Mouse event handler function

# @param[in] event Mouse event id

# @param[in] x X coordinate of the mouse event

# @param[in] y Y coordinate of the mouse event

# @param[in] flags Event flags

# @param[in] param Additional parameters passed to event. In current

#            implementation it is a tuple containing source image in

#            RGB color space, same image in Lab color space and an array

#            to store selected pixel coordinates.

## Mouse event handler function

def MouseHandler(event, x, y, flags, param):

    # Only double click event is processed

    if event != cv.EVENT\_LBUTTONDBLCLK:

        return

    # Extract data from parameters

    if param is None:

        return

    I, Ilab, sampleAreas, radius, colorMarksBGR = param

    # Add current double-clicked point to the list

    sampleAreas.append((x, y))

    # Create new image with marked pixel

    I2 = I.copy()

    for pix in sampleAreas:

        cv.circle(I2, pix, radius, (0, 0, 255), 1)

    cv.imshow("Source", I2)

    # Calculate color means

    colorMarks = []

    for pix in sampleAreas:

        # Create a mask for the current pixel location

        mask = np.zeros\_like(Ilab[0])

        cv.circle(mask, pix, radius, 255, -1)

        # Calculate the mean values of the a and b channels of the Lab color space within the mask

        a = Ilab[1].mean(where=mask > 0)

        b = Ilab[2].mean(where=mask > 0)

        # Add the mean values to the list of color marks

        colorMarks.append((a, b))

        # Calculate the mean BGR color within the mask and add it to the list

        colorMarksBGR.append(I[mask > 0, :].mean(axis=(0)))

    # Calculate distance and create segmented areas

    labels = np.zeros\_like(Ilab[0], dtype=np.uint8)

    for i, pix in enumerate(colorMarks):

        # Calculate the Euclidean distance between each pixel in the Lab color space and the color mark

        d = np.sqrt((Ilab[1] - pix[0])\*\*2 + (Ilab[2] - pix[1])\*\*2)

        # Create a mask for pixels that are within radius distance of the color mark

        labels[d < np.sqrt(2)\*radius] = i + 1

    # Show segmented image

    cv.imshow("Segmented", cv.cvtColor(labels\*50, cv.COLOR\_GRAY2BGR))

## Perform the segmentation in the CIE Lab color space using the nearest neighbor method

# @param[in] fn Image file name

def CIELabSegmentation(fn='yjsp.jpg'):

    # Read an image from file

    I = cv.imread(fn, cv.IMREAD\_COLOR)

    if not isinstance(I, np.ndarray) or I.data is None:

        print(f"Error reading file \"{fn}\"")

        return

    cv.imshow("Source", I)

    # Convert to CIE Lab color space

    Ilab = cv.cvtColor(I, cv.COLOR\_BGR2LAB)

    Ilab = cv.split(Ilab)

    # Define mouse callback function

    sampleAreas = []

    colorMarksBGR = []

    cv.setMouseCallback("Source", MouseHandler, (I, Ilab, sampleAreas, 10, colorMarksBGR))

    # Start an infinite event processing loop

    while True:

        key = cv.waitKey(20) & 0xFF

        if key == 27:

            break

        elif key == 114:

            cv.destroyAllWindows()

            sampleAreas = []

            colorMarksBGR = []

            cv.imshow("Source", I)

            cv.setMouseCallback("Source", MouseHandler, (I, Ilab, sampleAreas, 10, colorMarksBGR))

    cv.destroyAllWindows()

if \_\_name\_\_ == "\_\_main\_\_":

    CIELabSegmentation("yjsp.jpg")



This is a script that performs image segmentation in the CIE Lab color space using the nearest neighbor method. The script reads an image from file, converts it to the CIE Lab color space, and then displays the image. The user can then double-click on the image to select a region of interest, which is used to calculate the mean values of the a and b channels of the Lab color space within the region. These mean values are then used as color marks to segment the image. The script creates a new image with marked pixels, calculates the mean values of the a and b channels of the Lab color space within the selected region, and then calculates the Euclidean distance between each pixel in the Lab color space and the color mark. Pixels that are within a certain radius distance of the color mark are assigned to the same segment. The segmented image is then displayed.

* **Texture Segmentation**

import cv2

import numpy as np

from skimage import filters, morphology

from PA1\_Appendix import bwareaopen, imfillholes

# Read the image in grayscale

I = cv2.imread("jzdc.jpg",cv2.IMREAD\_GRAYSCALE)

# Calculate the entropy of the image

E = filters.rank.entropy(I,morphology.square(9)).astype(np.float32)

# Normalize the entropy image

Eim = (E - E.min()) / (E.max() - E.min())

# Display the entropy image

cv2.imshow("E", Eim)

# Threshold the entropy image using Otsu's method

ret, BW1 = cv2.threshold(np.uint8(Eim \* 255), 0, 255,cv2.THRESH\_OTSU)

# Display the thresholded image

cv2.imshow("BW1", BW1)

# Remove small objects from the thresholded image

BWao = bwareaopen(BW1, 2000)

# Define a structuring element for morphological operations

nhood = cv2.getStructuringElement(cv2.MORPH\_RECT, (9, 9))

# Close the thresholded image

closeBWao = cv2.morphologyEx(BWao, cv2.MORPH\_CLOSE, nhood)

# Fill holes in the closed image

Mask1 = imfillholes(closeBWao)

# Display the processed images

cv2.imshow("After bwareaopen 1", BWao)

cv2.imshow("After close 1", closeBWao)

cv2.imshow("After fill holes 1", Mask1)

# Find contours in the filled image

contours, h = cv2.findContours(Mask1, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_NONE)

# Create a binary image of the contour boundary

boundary = np.zeros\_like(Mask1)

cv2.drawContours(boundary, contours, -1, 255, 1)

# Create a segmented image using the boundary

segmentResults = I.copy()

segmentResults[boundary != 0] = 255

# Display the segmented image

cv2.imshow("Segment result 1", segmentResults)

# Create a masked image of the original image

I2 = I . copy ()

I2 [ Mask1 != 0] = 0

# Calculate the entropy of the masked image

E2 = filters . rank . entropy ( I2 , morphology . square (9)). astype ( np . float32 )

# Normalize the entropy image

Eim2 = ( E2 - E2 . min ()) / ( E2 . max () - E2 . min ())

# Display the entropy image

cv2.imshow("E2", Eim2)

# Threshold the entropy image using Otsu's method

ret , BW2 = cv2 . threshold ( np . uint8 ( Eim2 \* 255) , 0 , 255 , cv2 . THRESH\_OTSU )

# Display the thresholded image

cv2.imshow("BW2", BW2)

# Remove small objects from the thresholded image

BW2ao = bwareaopen ( BW2 , 2000)

# Close the thresholded image

closeBW2ao = cv2 . morphologyEx ( BW2ao , cv2 . MORPH\_CLOSE , nhood )

# Fill holes in the closed image

Mask2 = imfillholes ( closeBW2ao )

# Display the processed images

cv2.imshow("After bwareaopen 2", BWao)

cv2.imshow("After close 2", closeBWao)

cv2.imshow("After fill holes 2", Mask1)

# Find contours in the filled image

contours2 , h = cv2 . findContours ( Mask2 , cv2 . RETR\_TREE , cv2 . CHAIN\_APPROX\_NONE )

# Create a binary image of the contour boundary

boundary2 = np . zeros\_like ( Mask2 )

cv2 . drawContours ( boundary2 , contours2 , -1 , 255 , 1)

# Create a segmented image using the boundary

segmentResults2 = I2 . copy ()

segmentResults2 [ boundary2 != 0] = 255

# Display the segmented image

cv2.imshow("Segment result 2", segmentResults2)

# Create a masked image of the original image

texture1 = I . copy ()

texture1 [ Mask2 == 0] = 0

# Create a masked image of the original image

texture2 = I . copy ()

texture2 [ Mask2 != 0] = 0

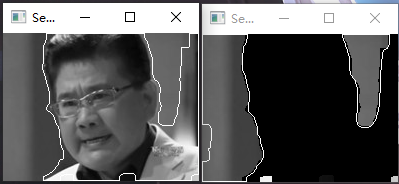
# Display the texture images

cv2.imshow("Texture 2", texture2)

# Wait for user input

cv2.waitKey(0)

****

****

This code is an implementation of texture segmentation using entropy filtering. The code reads an image in grayscale, calculates the entropy of the image, normalizes the entropy image, and displays it. Then, it thresholds the entropy image using Otsu's method and displays the thresholded image. It removes small objects from the thresholded image, closes the thresholded image, fills holes in the closed image, and displays the processed images. It finds contours in the filled image, creates a binary image of the contour boundary, and creates a segmented image using the boundary. It displays the segmented image. It creates a masked image of the original image, calculates the entropy of the masked image, normalizes the entropy image, and displays it. Then, it thresholds the entropy image using Otsu's method and displays the thresholded image. It removes small objects from the thresholded image, closes the thresholded image, fills holes in the closed image, and displays the processed images. It finds contours in the filled image, creates a binary image of the contour boundary, and creates a segmented image using the boundary. It displays the segmented image. It creates a masked image of the original image and displays the texture images. Finally, it waits for user input.