Índice

[Funcionalidad: Detección de bordes para el cálculo aproximado del perímetro. 1](#_Toc61800005)

[Funcionalidad: Detección de los píxeles que ocupa la fruta y su área 9](#_Toc61800006)

[Funcionalidad: Detección del color de la fruta 10](#_Toc61800007)

[Repositorio: 12](#_Toc61800008)

Funcionalidad: Detección de bordes para el cálculo aproximado del perímetro.

**Objetivo:** El algoritmo detectará los píxeles que conforman los bordes de una fruta o la verdura de la imagen que se pase por entrada.

**Procedimiento seguido:**

Para poder obtener los píxeles que pertenecen bordes de la imagen teníamos claro que debíamos utilizar una máscara de detección de flancos o bordes en algún punto del proceso y trabajar sobre aquellos píxeles resaltados por esta. (Principalmente, pensamos en utilizar la máscara de Sobel)

Para que una máscara de detección de flancos dé buen resultado, debemos aplicarlo sobre una imagen binarizada. Si lo aplicáramos sobre una imagen sin binarizar, los bordes serían mucho menos precisos y no coincidirían con el contorno de la fruta.

Al mismo tiempo, para poder binarizar una imagen y, en general, en muchos de los procedimientos utilizados en nuestro sistema, es necesario que la imagen sólo tenga un canal, es decir, que esté en escala de grises.

Teniendo en cuenta todo lo anterior decidimos que, para hallar los bordes de una imagen, es necesario realizar el siguiente procedimiento de técnicas: pasamos la imagen a escala de grises (de rgb a brillo), binarizamos la imagen para poder quedarnos con su “silueta”, aplicamos la máscara de detección de bordes y nos quedamos con los píxeles que forman el contorno de la fruta, que serán los que nos sirvan para calcular el perímetro.

Los algoritmos empleados, finalmente, para la detección de bordes son los siguientes:

(proceso ordenado de mayor a menor)

* deteccionBordes2.m Necesita imagen bordeada e imagen original
* prewitt.m, que llama a imRemap.m y a imConvolve.m(funciones imConvolve y aplicaMáscara) Necesita imagen binaria
* BinarizarImagenColor.m , que llama a rgb2gray.m,histogram.m, mvThreshold.m y a myImage2BW. Necesita imagen original

Comenzamos llamando a binarizarImagenColor(imagen original)

**Paso 1: transformación de una imagen a color a escala de grises**

En este paso, tenemos la imagen a color y queremos obtener la misma imagen en escala de grises. Para ello, llamamos a la función rgb2gray con nuestra imagen como parámetro de entrada. A continuación, explicamos rgb2gray. Hasta este punto, no hubo problemas, ya que la función la implementamos previamente a la hora de realizar la práctica 5:

Algoritmo:

function gray = rgb2gray(image) definimos la función con parámetro de entrada “image”, que es la imagen a color.

[filas, cols, canales] = size(image); extraemos información sobre los 3 tamaños de la imagen

i = [1:filas]; creamos un array con las posiciones de las filas

j = [1:cols]; creamos un array con las posiciones de las columnas

r=image(i,j,1); separamos los canales de la imagen

g=image(i,j,2);

b=image(i,j,3);

gray(i,j) = 0.299\*r + 0.587\*g + 0.114\*b; la nueva imagen, al ser en escala de grises, tendrá un único canal, es decir, una única matriz que indicará los niveles de gris de cada píxel.

end

En realidad, con esta función estamos transformando RGB al sistema de color XYZ y quedarnos con la componente Y, que representa la luminosidad del color.

**Paso 2: binarización de la imagen**

Una vez tenemos la imagen con un único canal, podemos pasar a binarizarla.

Para poder binarizar la imagen, necesitamos obtener el histograma de esta. La función “histogram” nos generará el histograma h (un array de tamaño 255) en el que cada posición representa un nivel de gris y, el valor de la posición, la cantidad de píxeles que tienen este valor. De esta manera, el algoritmo recorre los píxeles y aumenta en 1 las posiciones correspondientes a los niveles de gris que coinciden con los de los píxeles.

Esta vez, el parámetro de entrada “canal” será 1, ya que es una imagen en escala de grises.

function h = histogram(image, canal)

h=zeros(1,256);

[filas, cols, canales]= size(image);

for i=1:filas

for j=1:cols

pixel = image(i,j,canal)+1;

h(pixel) = h(pixel)+1;

end

end

end

Teniendo ya el histograma, podemos binarizar la imagen. Sin embargo, es muy importante escoger un buen umbral para no omitir ningún detalle importante de la fruta. Es por esto que no podemos elegir un umbral aleatorio y que sea el mismo para cada imagen. De esto se encarga las función mvThreshold y sus funciones auxiliares ACS, ACProm. Estas fueron implementadas durante la práctica 6.

ACS(histograma) nos devuelve el array SA con la suma acumulada del histograma desde la posición 1(porque en octave la primera posición no es 0, sino 1) hasta la posición i.

ACProm(histograma) nos devuelve un array M con la suma acumulada ponderada del histograma. En cada bucle, calculamos el valor de la posición anterior de M por el valor anterior del histograma más el valor anterior de M.

function SA= ACS(histo)

SA = zeros(1,256);

SA(1) = histo(1);

for i=2:256

SA(i)=SA(i-1)+histo(i);

endfor

endfunction

function M = ACProm(histo)

M = zeros(1,256);

M(1) = 0;

for i=2:256

M(i)=M(i-1)+(i-1)\*histo(i);

endfor

endfunction

Estas funciones nos serán necesarias para poder calcular, en las diferentes iteraciones de cálculo automático del umbral por el que vamos a binarizar, mu1 y mu2.

En la función mvThreshold(histo) se implementa el algoritmo que permite el cálculo de un umbral adecuado en función de un histograma, su parámetro de entrada. A excepción de en la primera iteración, en la que el umbral será equivalente al cociente de la suma acumulada ponderada entre la suma acumulada, el umbral será la parte entera entre la media de mu1 y mu2. La función terminará su ejecución cuando el umbral, de una iteración a la siguiente, no experimente cambios significativos.

function [umbral, mu1, mu2] = mvThreshold(histo)

SA = ACS(histo);

M = ACProm(histo);

umbral = fix(M(256)/SA(256));

last = 1;

while (last != umbral)

last = umbral;

mu1 = M(umbral)/SA(umbral);

mu2 = (M(256)-M(umbral))/(SA(256)-SA(umbral));

umbral = fix((mu1 + mu2)/2);

endwhile

endfunction

Binarizamos la imagen, ahora, con la función myImage2BW, que recibe como parámetros de entrada la imagen en escala de grises y el umbral que hemos calculado con mvThreshold. De esta forma, todos los píxeles que no superen el umbral serán 0 y, los que lo hagan, 255.

function bwImage = myImage2BW(imagen, threshold)

[rows cols] = size(imagen);

bwImage = zeros(rows, cols);

bwImage = (imagen >= threshold) \* 255;

end

**Paso 3: aplicar máscara para el realzado de bordes**

Ahora que ya tenemos la imagen binaria, termina la ejecución de binarizarImagenColor (imagenOriginal) y, por fin, podemos aplicarle una máscara para realzar los bordes de la silueta de la fruta. Llamamos a la función “prewitt”, ya que nos pareció ligeramente mejor para las imágenes que escogimos que la máscara de Sobel, que fue nuestra otra principal candidata. Explicamos, más adelante, el proceso y los problemas que tuvimos a la hora de implementar esta función.

Para ello, aplicamos las máscaras de aproximación del eje x y de aproximación del eje y, Gy y Gx, respectivamente mediante la función imConvolve.

Para convolucionar una máscara con una imagen monocromática, para cada píxel, centramos la máscara en el píxel seleccionado y su valor resultante será el sumatorio de los productos de los valores que estén en las mismas posiciones. La función imConvolve aplica la máscara que se le pasa como parámetro de entrada a la imagen binarizada, su otro parámetro de entrada.

function R = imConvolve(image, mask)

image = double(image);

mask = double(mask);

[filas, cols] = size(image);

[fmask, colmask] = size(mask);

R = zeros(filas, cols, "single");

incrf = fix(fmask/2);

incrc = fix(colmask/2);

imax = filas - fmask+1;

jmax = cols - colmask+1;

for i=1:imax

for j=1:jmax

R(i+incrf, j+incrc) = aplicaMascara(image, i,j, mask);

end

end

endfunction

Las máscaras, en este caso las de Prewitt, son las siguientes:

maskGy = [-1 -1 -1; 0 0 0; 1 1 1];

maskGx = [-1 0 1; -1 0 1; -1 0 1];

imCovolve, para aplicar la máscara cada píxel, se ayuda de aplicaMáscara(image,n,m,mask), donde n y m marcan la fila y la columna del píxel en el que se centra la máscara a aplicar y mask es la máscara a aplicar.

function val = aplicaMascara(image, n, m, mask)

val = single(0);

[filas, cols] = size(mask);

for k=1:filas

for l=1:cols

val = val + image(n+k-1, m+l-1) \* mask(k,l);

endfor

endfor

Tras aplicar estas máscaras, es necesario reajustar los valores de la imagen, ya que la máscara de prewitt no está normalizada, es decir, los valores obtenidos de nivel de gris para los píxeles no se ajustan al rango [0,255]. Para el reajuste, utilizamos imRemap, que tiene como parámetro de entrada la imagen a ajustar y, como salida, la imagen ajustada.

function out = imRemap(image)

maxval = max(max(image));

minval = min(min(image));

[filas,cols]=size(image);

for i=1:filas

for j=1:cols

image(i,j) = (image(i,j)-minval)\*255/(maxval-minval);

endfor

endfor

out = uint8(image);

endfunction

En esta función de reajuste, calculamos el mínimo y el máximo valor para determinar el rango. Reajustamos un píxel restándole a su valor el mínimo, dividiéndolo entre un gradiente (máximo menos mínimo) y, para que se diferencien bien los valores, multiplicando por 255.

La función prewitt(imagenBinaria), ayudándose de estas funciones anteriormente explicadas, quedaría simplificada a las siguientes líneas:

function bordes= prewitt(IN)

maskGy = [-1 -1 -1; 0 0 0; 1 1 1];

maskGx = [-1 0 1; -1 0 1; -1 0 1];

R = imConvolve(IN, maskGy);

maximo= max(max(R));

RR = imRemap(R);

bordes = imConvolve(RR, maskGx);

RR = imRemap(bordes);

bordes= RR;

figure();

imshow(bordes);

endfunction

Sin embargo, no realizamos el código bien de primera mano. Hubo bastantes fallos que nos hcieron dar bastantes vueltas.

Interfaz de usuario gráfica

Descripción generada automáticamentePor ejemplo, pusimos un condicional para que sólo se reajustara la imagen en caso de cumplirse este. Sin embargo, ni tenía la sintaxis adecuada ni se daba el caso en el que no se usara, por esto de que no está normalizada. Como nunca entraba en la condición de reajuste, siempre nos devolvía una imagen con rangos erróneos. Al principio no nos dimos cuenta porque octave expresaba en la imagen del resultado los números negativos y los que pasaban de 255 como blancos. Como no daba error, tardamos bastante en darnos cuenta de que el fallo se hallaba en que no estaba reajustando. De hecho, al principio intentamos aplicar la máscara de Sobel y cambiamos a la de Prewitt, pensando que esta nos daría mejor resultado. Fue en posteriores cálculos donde nos dimos cuenta de que algo no terminaba de encajar y, remontando, llegamos al fallo. Es bastante curioso cómo, en algunas imágenes (sin reajustar nada), varía tanto el resultado de Prewitt a Sobel.

Mostramos la imagen ‘filaplatanos.jpg’,

su imagen binarizada y la imagen

resultante de aplicar Prewitt y Sobel

respectivamente sin el reajuste

adecuado.

Interfaz de usuario gráfica

Descripción generada automáticamente

Finalmente, al solución fue eliminar el

condicional y reajustar siempre y, además,

utilizar Prewitt en lugar de Sobel

apesar de que, estando en el rango

correcto, las diferencias

se volvieron mínimas.

**Paso 4: detección de bordes de la fruta , coloreado de estos en la imagen original y perímetro:**

Llamamos, finalmente, a la función deteccionBordes2(imagenOriginal, imagenBordes), en la que imagenBordes es la imagen con los bordes realzados obtenido con Prewitt.

El resultado es la imagen original, pero con los bordes detectados de la fruta coloreados de azul.

function [OUT, perimetro]= deteccionBordes2(imgOriginal, imgBordes)

[filas, cols, canales] = size(imgOriginal);

ngFondo=imgBordes(1,1); Esta línea sirve para determinar el color de fondo de la imagen. Esta idea se basa en que, al ser la imagen de los bordes de un nivel de gris liso, en general, el color de las esquinas coincidirá con el color del fondo.

perimetro=0;

for i=1:filas

for j=1:cols

if imgBordes(i,j)==ngFondo Si el píxel en la posición i,j de los bordes coincide que es fondo, se mantienen los píxeles de la imagen original.

OUT(i,j,1:canales)= imgOriginal(i,j,1:canales);

Else Si no, se resaltan en azul

perimetro++;

OUT(i,j,1:2) = 0;

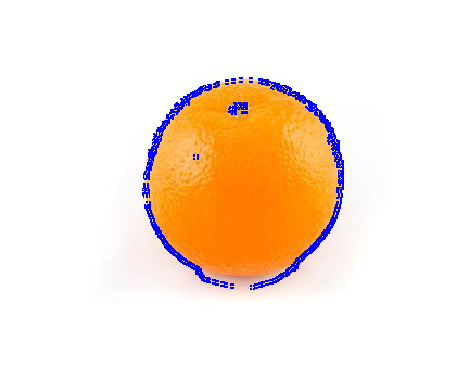
OUT(i,j,3) = 255;

endif

endfor

endfor

endfunction

Si ha funcionado bien, la función nos devolverá la imagen con los bordes pintados de azul y el tamaño del perímetro de la fruta o verdura, calculado a partir de los bordes. Una imagen resultado podría ser la siguiente:

‘NaranjaRedonda.png’ tras el proceso de

Cálculo de bordes.

En un principio, planteamos el código de detectar los bordes de una manera distinta, pero que resultó ser problemático.

Este código fallido es el archivo “deteccionBordes.m”. La idea que teníamos con este primer intento era detectar el color de los bordes y pintar aquellos píxeles de la imagen original que tuvieran dicho color en la imagen de los bordes, pero no acertamos cómo ponderar el nivel de gris de los bordes de manera adecuada. Decidimos que era más fácil detectar el fondo y sustraerlo.

# Funcionalidad: Detección de los píxeles que ocupa la fruta y su área

**Objetivo:** El algoritmo detectará los píxeles que conforman el cuerpo de la fruta.

El desarrollo de esta funcionalidad fue paralelo al desarrollo de la detección de bordes, pero contando con un proceso mucho más simple que este, ya que tan sólo necesitamos binarizar la imagen original para poder aplicar nuestra función deteccionFruta, que pinta de azul, sobre la imagen original, los píxeles que conforman el cuerpo de la fruta.

Binarizamos la imagen con la función binarizarImagenColor, tal como hemos explicado en la detección de bordes.

function OUT = deteccionFruta(imgOriginal, imgBinarizada)

[filas, cols, canales] = size(imgOriginal);

for i=1:filas

for j=1:cols

if imgBinarizada(i,j) == 0 Al binarizar la imagen, se quedarán con valor 0 los píxeles correspondientes a la silueta de la fruta, que serán los que pintemos de azul sobre la imagen original.

OUT(i,j,1:2) = 0;

OUT(i,j,3) = 255;

else

OUT(i,j,1:canales) = imgOriginal(i,j,1:canales);

endif

endfor

endfor

endfunction

Con areaAlimento, contamos el tamaño en píxeles que ocupa la fruta o verdura en la imagen. Para ello, partimos de la imagen binarizada y contamos los píxeles con valor 0 (color negro, que representa el alimento).

function area = areaAlimento(imgBinarizada)

[filas, cols, canales] = size(imgBinarizada);

area = 0;

for i=1:filas

for j=1:cols

if imgBinarizada(i,j) == 0

area++;

endif

endfor

endfor

endfunction

# Funcionalidad: Detección del color de la fruta

**Objetivo:** El código detectará el color de la fruta o verdura presente en la imagen.

Para esta funcionalidad, hemos implementado la función color (imgOriginal, imgBinarizada), siendo imgOriginal la imagen original e imgBinarizada la imagen binarizada de imgOriginal.

La idea de este algoritmo es, guardar las posiciones en las que se encuentra el alimento en dos vectores (vectorI y vectorJ), coger 4 pixeles aleatorio de ellos y mirar que colores se repiten, siendo este el color de la fruta. Este método es estadístico, es decir, es posible que muestre una tonalidad del color de la fruta poco adecuada o incluso llegar a dar un color diferente (como puede ser el caso de una naranja, si al binarizar se cuentan las hojas, existe la posibilidad que salga verde). Pero dicha probabilidad es lo suficientemente baja, de hecho, en todas las pruebas que hemos realizado ha mostrado el color oportuno.

La salida será una imagen con el color de la fruta.

function color = color(imgOriginal, imgBinarizada)

[filas, cols, canales] = size(imgBinarizada);

color = imgOriginal;

tam = areaAlimento(imgBinarizada); El tamaño de los vectores será el numero de pixeles que compone a la fruta, es decir, lo que hemos denominado el área del alimento.

vectorI = zeros(tam); Inicializamos los vectores que guardan la posición de la fruta a 0

vectorJ = zeros(tam);

ind = 0;

for i=1:filas En este bucle, recorremos la imagen binarizada para cada vez que estemos en una posición donde se encuentra la fruta, salvarla en los vectores aumentado su respectivo índice.

for j=1:cols

if imgBinarizada(i,j) == 0

ind++;

vectorI(ind) = i;

vectorJ(ind) = j;

endif

endfor

endfor

n1 = fix(rand()\*tam); Generamos 4 números aleatorios entre 0 y tam y nos quedamos con la parte entera, estos números indicarán una posición aleatoria de los pixeles que componen la fruta

n2 = fix(rand()\*tam);

n3 = fix(rand()\*tam);

n4 = fix(rand()\*tam);

n = [n1, n2, n3, n4];

colorRepe = n1;

for k = 1:4 Con este bucle, desarrollamos un algoritmo de comparación entre los 4 números, (y que funcionaría igual se añadieran más números) guardando en colorRepe el último color que se repita, en caso de no repetirse ninguno saldría con n1

aux = k+1;

for l=aux:size(n)(2)

if(imgOriginal(vectorI(n(k)), vectorJ(n(k)),:) == imgOriginal(vectorI(n(l)), vectorJ(n(l)),:)) Aquí es donde se comparan los colores extraídos aleatoriamente

colorRepe = k;

endif

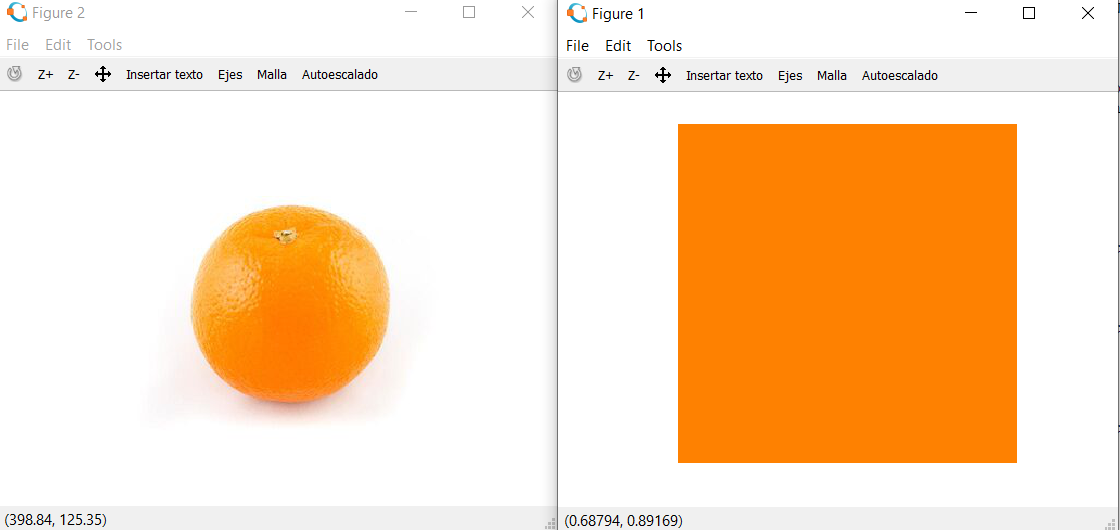
endfor

endfor

color = imgOriginal(vectorI(n(colorRepe)), vectorJ(n(colorRepe)),:); Por últimos creamos una imagen con el color de la fruta como fondo

endfunction

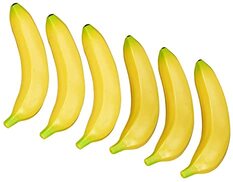
Ejemplo del resultado



Imágenes de entrada:

Para las entradas de los algoritmos hemos escogido imágenes que, una vez en blanco y negro, puedan ser lo más bimodales posibles, debido a que el proceso de binarización es usado para prácticamente todas las funcionalidades. También hemos tenido que redimensionar las imágenes a más baja resolución, ya que los algoritmos tardaban demasiado por el tamaño de la entrada. Aquí dejamos algunos ejemplos de las posibles imágenes de entrada:

Repositorio:

A lo largo del desarrollo hemos usado GitHub, les dejamos en enlace al repositorio:

<https://github.com/Franbr31/EntregablePSM>