INTELIGENCIA ARTIFICIAL

LUIS EDIN IBARGUEN MOSQUERA

JHON ALEJANDRO GIRALDO

ASDRUBAL ANDRES VELEZ

YEISSON ESTIVEN CASTRO

PROYECTO N°1

Carlos Alberto Londoño L.

Docente

Corporación de Estudios Tecnológicos del Norte del Valle

CORPORACIÓN DE ESTUDIOS TECNOLÓGICOS

DEL NORTE DEL VALLE

CARTAGO

TECNOLOGÍA EN SISTEMAS DE INFORMACION

2018

Contenido

[**Detector de bordes Canny cómo contar objetos con OpenCV y Python** 3](#_Toc512271764)

[**El proceso para contar objetos con OpenCV** 4](#_Toc512271765)

[**Convertir la imagen en escala de grises con OpenCV** 6](#_Toc512271766)

[**Filtrado del ruido en una imagen** 7](#_Toc512271767)

[**Filtro Guassiano para la eliminación de ruido en imágenes** 8](#_Toc512271768)

[**Detector de bordes Canny** 11](#_Toc512271769)

[**Detección de bordes con Sobel** 12](#_Toc512271770)

[**Filtrado de bordes mediante la supresión non-maximun** 13](#_Toc512271771)

[**Aplicar umbral por histéresis** 14](#_Toc512271772)

[**Detector de bordes Canny con OpenCV** 14](#_Toc512271773)

[**Buscar los contornos de una imagen** 15](#_Toc512271774)

[**Dibujar los contornos en una imagen con OpenCV** 17](#_Toc512271775)

[**Algoritmo para contar objetos con OpenCV** 17](#_Toc512271776)

[**Importación de librerías y carga de imagen en OpenCV** 18](#_Toc512271777)

[**Conversión a escala de grises y suavizado Gaussiano** 19](#_Toc512271778)

[**Calcular el detector de bordes Canny con OpenCV** 21](#_Toc512271779)

[**Dibujar y detectar contornos** 22](#_Toc512271780)

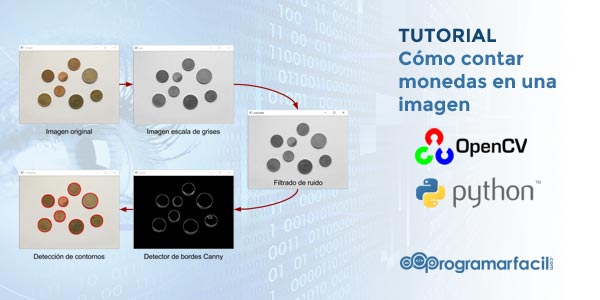
[**Código completo para contar objetos con OpenCV y Python** 24](#_Toc512271781)

[**Conclusión detector de bordes Canny para contar objetos con OpenCV** 27](#_Toc512271782)

**Detector de bordes Canny cómo contar objetos con OpenCV y Python**

En este artículo voy a explicarte cómo puedes **contar objetos en una imagen gracias al detector de bordes Canny con OpenCV y Python**.

La detección de bordes es una técnica muy utilizada que nos permite aislar los objetos y separarlos del fondo. Una vez obtenido los bordes, lo único que nos faltaría es detectar los diferentes contornos para poder contar los objetos.



Poder contar tornillos, tuercas, monedas o cualquier otro objeto es relativamente sencillo gracias a OpenCV. La Visión Artificial es un subcampo de las matemáticas que abarca muchas técnicas del tratamiento digital de imágenes.

No hay un algoritmo perfecto y siempre dependerá de las condiciones en las que se tomó la imagen. Hoy veremos un caso ideal, donde hay un fuerte contraste entre el fondo y los propios objetos.

¿Utilidad? Pues imagínate que necesitas hacer una aplicación que cuente el dinero, que cuente tornillos o que cuente fruta.

**El proceso para contar objetos con OpenCV**

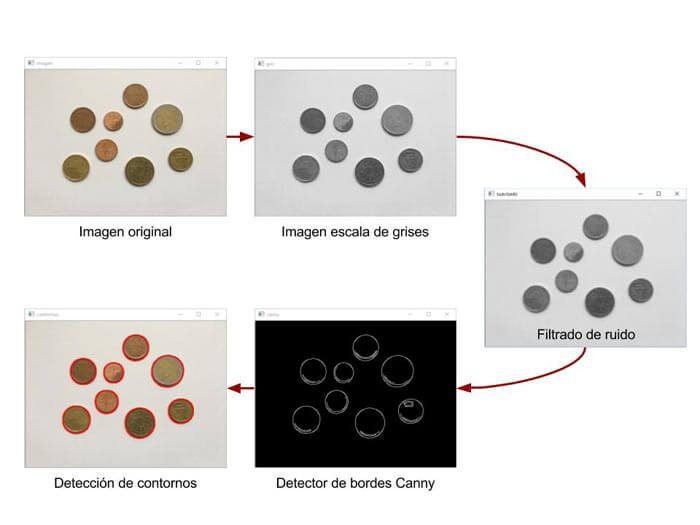
Este proceso involucra varias fases que nos permitirán **aislar los objetos dentro de una imagen**. Vamos a ir viendo el algoritmo para contar objetos con OpenCV con un caso práctico. Vamos a contar las monedas que hay en la siguiente imagen.



En la imagen que vamos a analizar existe un fuerte contraste entre los objetos, las monedas, y el fondo. Esto facilitará mucho a la hora de contar cuántas monedas hay.

El proceso se divide en 5 fases que iremos viendo a lo largo de este tutorial:

1. Convertir la imagen a escala de grises
2. Filtrar la imagen para eliminar el ruido
3. Aplicar el detector de bordes Canny
4. Buscar los contornos dentro de los bordes detectados
5. Dibujar dichos contornos



Cada una de estas fases dará como resultado una imagen que será la entrada de la siguiente fase. Todo el proceso lo haremos con **OpenCV y Python.** Por lo tanto deberás tener instalado estas herramientas en tu ordenador.

**Convertir la imagen en escala de grises con OpenCV**

Una vez que has cargado la imagen, lo primero que tienes que hacer es convertir a escala de grises con OpenCV. Cuando trabajamos con imágenes a color, el coste computacional crece de manera exponencial.

Por ejemplo, si tienes tres componentes de color como en el espacio RGB (R de Red, G de Green y B de Blue) es como si estuvieras trabajando con 3 imágenes diferentes, una para cada componente.

Ya no solo es eso, para aplicar el detector de bordes Canny necesitamos que la imagen esté en escala de grises. En OpenCV hay un método que nos permite cambiar entre espacios de color.

imagenconvertida = cv2.cvtColor(imagen, tipo\_conversion)

Donde

* imagenconvertida: es la imagen resultante en el nuevo espacio de color.
* imagen: es la imagen original. Tenemos que saber que espacio de color utiliza.
* tipo\_conversion: es una constante que indica de que espacio a que espacio vamos a convertir.

El último parámetro, tipo\_conversión, nos permite convertir entre varios espacios de color. Si lo que queremos es convertir entre RGB (en OpenCV se nombra BGR por temas históricos) a escala de grises, el tipo\_conversión es cv2.COLOR\_BGR2GRAY.

La sentencia final quedaría de la siguiente manera:



|  |  |
| --- | --- |
| 1  2 | # Convertimos en escala de grise  gris = cv2.cvtColor(imagen, cv2.COLOR\_BGR2GRAY) |

Puedes ver todos los tipos de conversiones entre los [**diferentes espacios aquí**](http://docs.opencv.org/2.4.8/modules/imgproc/doc/miscellaneous_transformations.html).

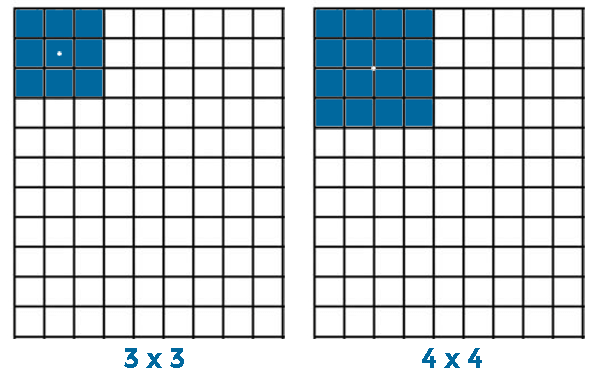
**Filtrado del ruido en una imagen**

Las**imágenes digitales no son perfectas**. El ruido inherente de los propios dispositivos o los efectos contraproducentes por iluminación alteran la realidad.

Es algo que no solo sucede en los entornos visuales. Cualquier señal digital o analógica está expuesta a diferentes [**fuentes de ruido**](https://programarfacil.com/podcast/eliminar-el-ruido-en-arduino/). Debes vivir con ello y, sobre todo, debes saber cómo corregirlo.

En el tratamiento digital de imágenes hay diferentes métodos para eliminar el ruido (promediado, mediana, Gaussiano o bilateral).

Todos utilizan la misma operación matemática, la convolución. Consiste en ir recorriendo píxel a píxel una imagen  con una máscara o kernel de *N x N*. Este tamaño determina el número de píxeles con el que vamos a trabajar.

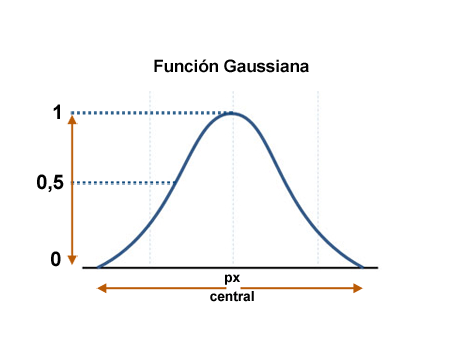


Es muy importante que **el tamaño sea impar para siempre tener un píxel central** que será el píxel en cuestión que estamos tratando.

El objetivo es suavizar la imagen es decir, eliminar los detalles. Puedes verlo como un desenfoque en una cámara fotográfica. Para poder contar los objetos o monedas vamos a aplicar el filtro Gaussiano.

**Filtro Guassiano para la eliminación de ruido en imágenes**

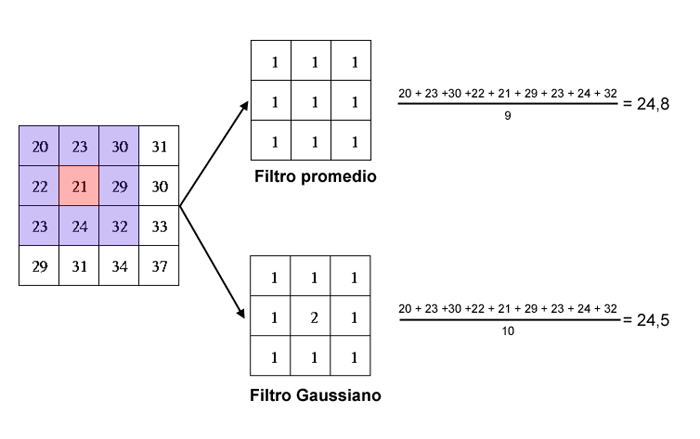
El filtrado Gaussiano es igual que un promediado pero ponderado. Esta ponderación se hace siguiendo la**Campana de Gauss**. Con esto se consigue dar más importancia a los píxeles que están más cerca del centro de los que están más alejados.



Pero **¿por qué utilizar una Campana de Gauss?** No es más que una aproximación de cómo ve el ojo humano, intentando ser lo más natural posible. La función Gaussiana o Distribución Normal se utiliza en muchas áreas de la ciencia.

Por ejemplo, la altura de la población o el coeficiente intelectual se puede representar a a través de la Distribución Normal. Gracias a su forma acampanada y simétrica, hace que los elementos que están más cerca del centro sean los más comunes. Por el contrario, los elementos más lejanos serán los más raros o diferentes.

Para poder aplicarlo en una imagen debemos hacerlo en dos dimensiones. Esto lo hacemos a través de una máscara o kernel de convolución.



Como puedes comprobar en la imagen anterior, se da más importancia al píxel central que a los píxeles que están alrededor de éste.

En OpenCV tenemos un método que nos ayuda a realizar el filtrado Gaussiano.



|  |  |
| --- | --- |
| 1 | gaussiana = cv2.GaussianBlur(imagen, (n, n), σ) |

Donde

* gaussiana: es la imagen desenfocada resultante.
* imagen: es la imagen original, la que queremos suavizar o desenfocar.
* n X n: es el tamaño del kernel o máscarade convolución. Recuerda que debe ser impar.
* σ: sigma (σ) representa la desviación estándar en el eje X es decir, la anchura de la campana de Gauss. Si ponemos un 0, OpenCV se encargará automáticamente de calcular ese valor para el kernel o máscara que hemos elegido. Esta es la opción aconsejable.

Si aplicamos el filtro Gaussiano con un kernel de 5×5 el código sería el siguiente.



|  |  |
| --- | --- |
| 1  2 | # Aplicar suavizado Gaussiano  gaussiana = cv2.GaussianBlur(gris, (5,5), 0) |

**Detector de bordes Canny**

El proceso para detectar bordes con Canny se divide en 3 pasos.

* Detección de bordes con Sobel
* Supresión de píxeles fuera del borde
* Aplicar umbral por histéresis

Uno de los grandes inconvenientes de los algoritmos de la visión artificial es la parametrización. Muchas técnicas **requieren establecer parámetros o condiciones iniciales** según cada situación. En muchas ocasiones esos parámetros son exclusivos para una determinada iluminación o perspectiva.

Esto dificulta mucho a la hora de buscar una solución única para diferentes situaciones y por lo tanto, algo que funciona correctamente en unas condiciones de iluminación no tiene porque funcionar en otras circunstancias.

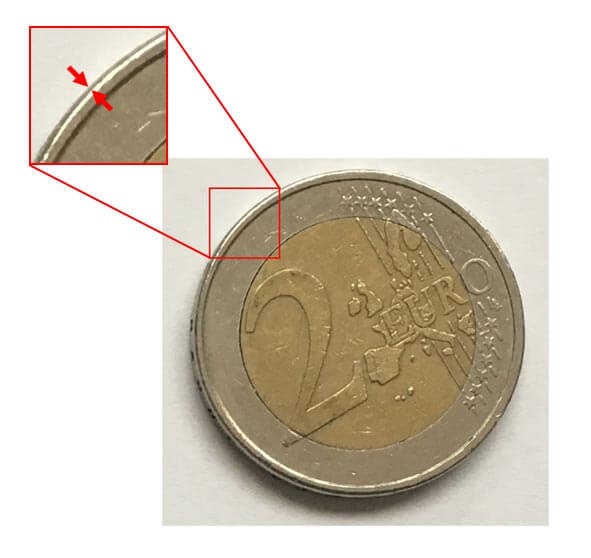


Normalmente, este tipo de algoritmos se entrenan para buscar la mejor solución para múltiples casos. A todo esto se le llama **Machine Learning**. Aunque esto se salga fuera del tema de este artículo, debes ser consciente de la dificultad que todo esto supone.

**Detección de bordes con Sobel**

El detector de bordes **Sobel se basa en el cálculo de la primera derivada**. Esta operación matemática mide las **evoluciones y los cambios de una variable**. Básicamente se centra en detectar cambios de intensidad.

Cuando hablamos de bordes en una imagen, hablamos de los píxeles donde hay un cambio de intensidad. En imágenes como la que estamos utilizando para contar las monedas, hay un fuerte contraste entre el fondo y las monedas.



El objetivo es poder detectar este borde a través de la primera derivada.

**Filtrado de bordes mediante la supresión non-maximun**

El objetivo en esta fase es poder quedarnos con aquellos bordes que cumplan cierta condición. En el detector de bordes Canny serán aquellos que tengan como grosor 1.

La supresión non-maximun es una técnica que permite adelgazar los bordes basándose en el gradiente.

**Aplicar umbral por histéresis**

La **umbralización de imágenes** nos permite también **segmentar una imagen en sus diferentes objetos**. Básicamente consiste en determinar un umbral por el cual se decide si un píxel forma parte del fondo o forma parte de un objeto.

En el detector de bordes Canny este es el último paso. Al contrario que el umbral simple, el umbral por histéresis se centra en establecer dos umbrales, uno máximo y otro mínimo.

Esto te ayudará a determinar si un píxel forma parte de un borde o no. Pueden darse 3 casos:

* Si el valor del píxel es mayor que el umbral máximo, el píxel se considera parte del borde.
* Un píxel se considera que no es borde si su valor es menor que el umbral mínimo,
* Si está entre el máximo y el mínimo, será parte del borde si está conectado con un píxel que forma ya parte del borde.

**Detector de bordes Canny con OpenCV**

Todo este proceso puede ser extremadamente difícil de implementar por ti mismo. Gracias a **OpenCV todo esto resulta relativamente sencillo y práctico**.

Esta librería nos aporta métodos y funciones fáciles de implementar. Si además lo unimos a la sencillez de hacerlo con un lenguaje como Python, las técnicas de visión artificial y tratamiento de imágenes están al alcance de todo el mundo.

El método para calcular los bordes con el método Canny es el siguiente.



|  |  |
| --- | --- |
| 1 | canny = cv2.Canny(imagen, umbral\_minimo, umbral\_maximo) |

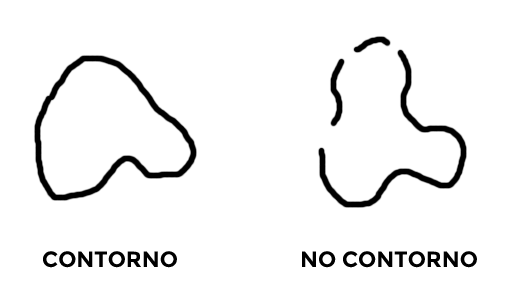
Donde:

* canny: es la imagen resultante. Aparecerán los bordes detectados tras el proceso.
* imagen: es la imagen original.
* umbral\_minimo: es el umbral mínimo en la umbralización por histéresis
* umbral\_maximo: es el umbral máximo en la umbralización por histéresis

Como ya he comentado, el umbral mínimo y el máximo dependerá de cada situación. Debes experimentar con la escena que estás analizando y elegir los más adecuados.

**Buscar los contornos de una imagen**

Hay que saber la diferencia que existe entre un borde y un contorno. Los bordes, como hemos visto anteriormente, son cambios de intensidad pronunciados. Sin embargo, **un contorno es una curva de puntos sin huecos ni saltos** es decir, tiene un principio y el final de la curva termina en ese principio.



El objetivo de esta fase es analizar todos los bordes detectados y comprobar si son contornos o no. En OpenCV lo podemos hacer con el siguiente método.



|  |  |
| --- | --- |
| 1 | (imagen, contornos, jerarquia) = cv2.findContours(imagenbinarizada, modo\_contorno, metodo\_aproximacion) |

Donde:

* Resultado: obtenemos 3 valores como resultados, el interesante para nosotros es contornos.
  + imagen: la imagen después de aplicar la detección de contornos. Es modificada y por lo tanto no podremos utilizarla.
  + contornos: es una lista de Python con todos los contornos que ha encontrado. Luego veremos cómo dibujar estos contornos en una imagen.
  + jerarquía: la jerarquía de contornos.
* imagenbinarizada: es la imagen donde hemos detectado los bordes o umbralizado. Mucho ojo, este método modifica esta imagen así que es conveniente que sea una copia. Como norma general, la imagen binarizada debe ser con el fondo negro y los objetos a ser buscados deben ser blancos.
* modo\_contorno: es un parámetro interno del algoritmo que indica el tipo de contorno que queremos. Puede tomar diferentes valores RETR\_EXTERNAL, RETR\_LIST, RETR\_COMP y RETR\_TREE. Nos centraremos en el primero, RETR\_EXTERNAL que obtiene el contorno externo de un objeto.
* metodo\_aproximacion: este parámetro indica cómo queremos aproximar el contorno. Básicamente le decimos si queremos almacenar todos los puntos del contorno. Imagínate que tienes una línea recta ¿para qué quieres almacenar todos los puntos? con dos valdría. A lo mejor ahora no le encuentras sentido pero cuando se analizan muchos objetos en imágenes grandes, tendrás que llevar mucho cuidado con el tiempo de procesado y con la memoria. Puede tomar dos valores CHAIN\_APPROX\_NONE que toma todos los puntos y CHAIN\_APPROX\_SIMPLE,  que elimina todos los puntos redundantes.

**Dibujar los contornos en una imagen con OpenCV**

Por último nos queda dibujar los contornos que hemos encontrado en la imagen. Esto lo haremos a través de otro método de OpenCV.



|  |  |
| --- | --- |
| 1 | cv2.drawContours (imagen, lista\_contornos, numero\_contornos, color\_RGB, grosor) |

Donde:

* imagen: es la imagen donde vamos a dibujar los contornos.
* lista\_contornos: es la lista de Python con los contornos.
* numero\_contornos: el número de contornos que queremos dibujar si son todos pasar -1.
* color\_RGB: es un array o tupla con el color RGB del contorno.
* grosor: grosor de la línea a dibujar.

Y con esto ya tendríamos el algoritmo completo. Ahora vamos a unir todas las piezas y ver cómo quedaría el código completo.

**Algoritmo para contar objetos con OpenCV**

Una vez que tenemos claro lo que tenemos que hacer, vamos a unir todas las piezas y componer el código que nos permitirá contar las monedas o cualquier otro objeto dentro de una imagen.

Final del formulario

**Importación de librerías y carga de imagen en OpenCV**



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | import numpy as np  import cv2    # Cargamos la imagen  original = cv2.imread("imagenes/monedas.jpg")  cv2.imshow("original", original) |

Lo primero es importar las librerías NumPy y OpenCV. Luego, cargamos la imagen monedas. Te la dejo a continuación para que puedas hacer las pruebas con ella.



En mi caso he guardado el archivo *detector-canny.py* y he creado una carpeta en el mismo directorio para guardar la imagen (*imagenes/monedas.jpg*).

La última línea de código muestra la imagen original.

**Conversión a escala de grises y suavizado Gaussiano**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | # Convertimos a escala de grises  gris = cv2.cvtColor(original, cv2.COLOR\_BGR2GRAY)    # Aplicar suavizado Gaussiano  gauss = cv2.GaussianBlur(gris, (5,5), 0)    cv2.imshow("suavizado", gauss) |

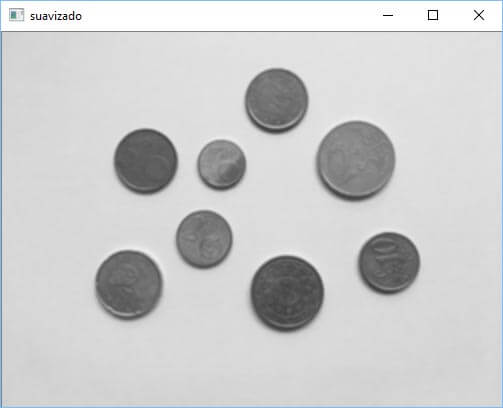
Para detectar los bordes con Canny, tenemos que partir de una imagen en escala de grises. El método cv2.cvtColor convertirá la imagen cargada en color a una imagen en escala de grises.

Luego aplicamos el suavizado Gaussiano para eliminar el ruido. El método *cv2.GaussianBlur()* admite 3 parámetros.

El primero es la imagen que queremos suavizar en nuestro caso la imagen en escala de grises.

El segundo parámetro es el kernel o máscara de convolución. Recuerda que debe ser impar. Comenzamos con un tamaño de 5 x 5. Cuanto mayor sea el kernel mayor será el suavizado y menos detalles tendrá la imagen.

El último parámetro es sigma (σ) y representa la desviación estándar en el eje X es decir, la anchura de la campana Gaussiana, por defecto lo vamos a dejar a 0. Esto hace que el método ponga el valor más adecuado de forma automática.



**Calcular el detector de bordes Canny con OpenCV**



|  |  |
| --- | --- |
| 1  2  3  4 | # Detectamos los bordes con Canny  canny = cv2.Canny(gauss, 50, 150)    cv2.imshow("canny", canny) |

Una vez que ya has suavizado la imagen, ya puedes proceder a la eliminación de los bordes con la supresión non-maximun y aplicar el umbral por histéresis.

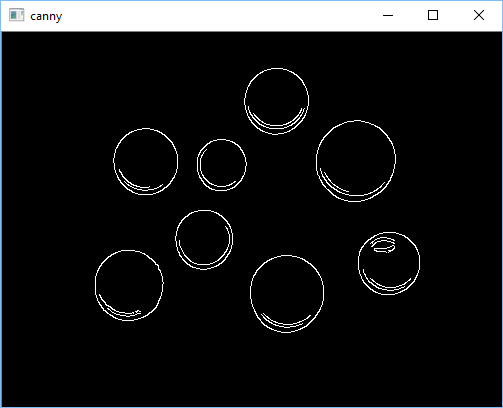
Estos dos procesos se hacen a través del método *cv2.Canny*. Admite 3 parámetros.

El primero es la imagen donde queremos detectar los bordes. Esta imagen es la que ya hemos suavizado en el paso anterior.

El segundo parámetro será el umbral mínimo y el tercer parámetro será el umbral máximo. Debes buscar los más adecuados para tu imagen aunque, como verás posteriormente, no importa que se detecten más bordes siempre y cuando no sean contornos es decir, curvas cerradas.

Un valor de 50 para el mínimo y 150 para el máximo puede ser un buen comienzo. Intenta subir el máximo para eliminar esos bordes inecesarios.

Por último, volvemos a mostrar la imagen con los bordes detectados. El resultado sería el siguiente.



**Dibujar y detectar contornos**

Una vez que tenemos los bordes, hay que decidir cuales de ellos forman contornos y cuales no.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | # Buscamos los contornos  (\_, contornos,\_) = cv2.findContours(canny.copy(), cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)    # Mostramos el número de monedas por consola  print("He encontrado {} objetos".format(len(contornos)))    cv2.drawContours(original,contornos,-1,(0,0,255), 2)  cv2.imshow("contornos", original)    cv2.waitKey(0) |

Con el método *cv2.findContours()* se detectan los contornos. Solo detectaremos los externos (cv2.RETR\_EXTERNAL) y se hará una aproximación para eliminar los píxeles del contorno redundantes (cv2.CHAIN\_APPROX\_SIMPLE).

Este método nos devuelve una lista de contornos además de dos variables que no vamos a utilizar, la imagen y la jerarquía de bordes. Con solo saber su tamaño, sabremos cuantas monedas ha encontrado. Toda esta información la mostramos por consola.

Por último, pintamos los contornos sobre la imagen original con el método *cv2.drawContours()*. Mostramos el resultado y esperamos a que se pulse una tecla con el método cv2.waitKey(0).



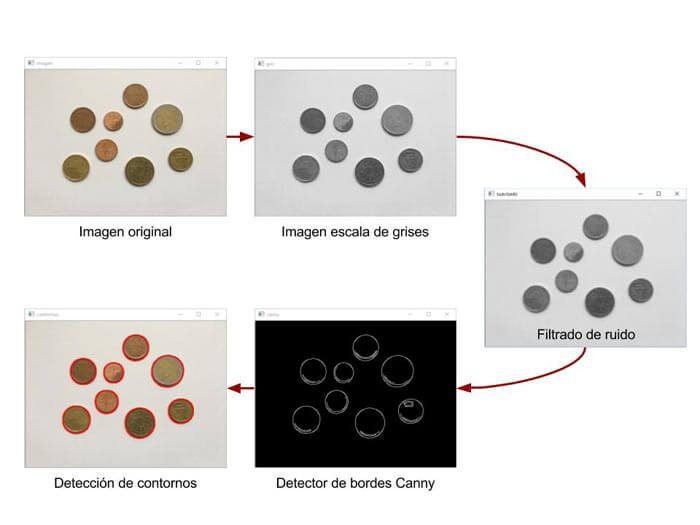
**Código completo para contar objetos con OpenCV y Python**

Como resumen de lo visto hasta ahora, te dejo a continuación todo el código. Puedes copiarlo y pegarlo en tu ordenador para probarlo.

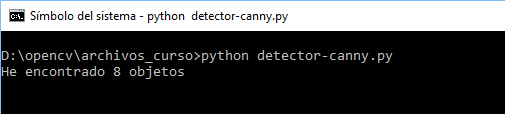


|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | import numpy as np  import cv2    # Cargamos la imagen  original = cv2.imread("imagenes/monedas.jpg")  cv2.imshow("original", original)    # Convertimos a escala de grises  gris = cv2.cvtColor(original, cv2.COLOR\_BGR2GRAY)    # Aplicar suavizado Gaussiano  gauss = cv2.GaussianBlur(gris, (5,5), 0)    cv2.imshow("suavizado", gauss)    # Detectamos los bordes con Canny  canny = cv2.Canny(gauss, 50, 150)    cv2.imshow("canny", canny)    # Buscamos los contornos  (\_, contornos,\_) = cv2.findContours(canny.copy(), cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)    # Mostramos el número de monedas por consola  print("He encontrado {} objetos".format(len(contornos)))    cv2.drawContours(original,contornos,-1,(0,0,255), 2)  cv2.imshow("contornos", original)    cv2.waitKey(0) |

Si ejecutas el código anterior con la imagen de las monedas el resultado es el siguiente.



Por consola obtendrás el valor de 8 objetos detectados.



**Conclusión detector de bordes Canny para contar objetos con OpenCV**

Hemos visto un ejemplo de cómo contar monedas, pero esto se puede utilizar para contar cualquier tipo de objeto. Cuando vayas probando con diferentes imágenes, comprobarás que en muchos casos no es suficientemente preciso.

Una fase previa que se podría incorporar es hacer que el contraste entre los objetos y el fondo se realce, pero eso lo dejo para otro tutorial. De momento, siempre que tengas un alto contraste entre el objeto y el fondo funcionará.

El algoritmo de contar objetos con el detector de bordes Canny es un ejemplo de cómo aplicar diferentes técnicas de visión artificial para conseguir un objetivo.

# Bibliografía

animusproject. (s.f.). *wix*. Obtenido de animusproject.wix.com/web/apps/blog/opencv-i

*cimat*. (s.f.). Obtenido de http://www.cimat.mx/~alram/VC/ramirez\_segmObjDetecc.pdf

google. (s.f.). *google*. Obtenido de https://plus.google.com/collection/whJrQE

*programarfacil*. (s.f.). Obtenido de https://programarfacil.com/blog/vision-artificial/detector-de-bordes-canny-opencv/

*riune*. (s.f.). Obtenido de https://riunet.upv.es/bitstream/handle/10251/83514/Agust%C3%AD%20-%20Contar%20objetos%20en%20una%20imagen%20mediante%20t%C3%A9cnicas%20de%20Visi%C3%B3n%20por%20Computador.pdf?sequence=1&isAllowed=y