DETECCIÓN DE BORDES MEDIANTE EL ALGORTIMO DE CANNY

José Francisco Ruiz Zamora

TRATAMIENTO DIGITAL DE IMÁGENES

Contenido

1 Introducción

El documento expuesto a continuación consiste en la explicación e implementación del *algoritmo de Canny* para la detección de bordes en una imagen.

La detección de bordes es una herramienta fundamental en el procesamiento de imágenes y en visión por computadora, particularmente en las áreas de detección y extracción de características, que tiene como objetivo la identificación de puntos en una imagen digital en la que el brillo de la imagen cambia drásticamente o, más formalmente, tiene discontinuidades.

Al aplicar un algoritmo de detección de bordes a una imagen es posible reducir la cantidad de datos a ser procesados, por lo que se puede filtrar la información que puede ser considerada como menos relevante, logrando preservar las propiedades estructurales de la imagen. Si el paso de detección de bordes fue satisfactorio, el paso de interpretar el contenido en la imagen original se puede reducir sustancialmente.

Existen distintos algoritmos para llevar a cabo el proceso de detección de bordes, sin embargo, uno de los más utilizados es el *algoritmo de Canny,* basado en el uso de la primera derivada.

Desarrollado por John F. Canny en 1986 cumple con los tres puntos para ser considerado un algoritmo óptimo:

- Buena detección: Marcar el mayor número real en los bordes de la imagen como sea posible.
- Buena localización: Los bordes deben estar lo más cerca posible del borde de la imagen real.
- Respuesta mínima: El borde de una imagen sólo debe ser marcado una vez, y siempre que sea posible, el ruido de la imagen no debe crear falsos bordes.

2 Idea general

Los bordes de una imagen digital se pueden definir como transiciones entre dos regiones de niveles de gris significativamente distintos. Suministran una valiosa información sobre las fronteras de los objetos y puede ser utilizada para segmentar la imagen, reconocer objetos, etc.

El *algoritmo de Canny* está basado en el uso de la primera derivada, que permite identificar un cambio brusco de intensidad en la imagen. Es decir, para encontrar los bordes de la imagen.

A grandes rasgos el algoritmo basa su funcionamiento en 4 pasos:

- Reducción de ruido: Se aplica un filtro para suavizar la imagen y eliminar el ruido lo máximo posible
- Encontrar intensidad del gradiente: se calcula la magnitud y orientación del vector gradiente en cada píxel.
- Supresión de no máximos: en este paso se logra el adelgazamiento del ancho de los bordes obtenidos con el gradiente, hasta lograr bordes de un píxel de ancho.
- Histéresis de umbral: Se aplica una función de histéresis basada en dos umbrales; con este proceso se pretende reducir la posibilidad de aparición de contornos falsos.

3 Etapas del algoritmo

A continuación se explican los pasos que realiza el programa para conseguir definir los bordes de una imagen siguiendo como estructura general los pasos definidos anteriormente.

3.1 Suavizar la imagen (filtro gaussiano)

Al obtener una imagen inevitablemente contiene ruido. El ruido es la variación aleatoria del brillo o color de la imagen, lo que puede hacer que el algoritmo no funcione correctamente.

Ya que el operador de *Canny* es susceptible a dicho ruido es necesario aplicar un filtro gaussiano antes de su aplicación. El filtro gaussiano difumina la imagen manteniendo la estructura básica de la misma, la cual no se ve afectada por el ruido a un nivel significativo.

Para realizar el suavizado se debe realizar convolución sobre la imagen original con un filtro gaussiano definido por la ecuación:

$$G(x,y) = \frac{1}{2\pi\ \sigma^2} * e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Donde la desviación estándar (**σ**) es clave, ya que al aumentar su valor se reduce el ruido pero se difuminan los bordes y se pierde calidad en la localización.

Es decir, en este punto se tendría que:

1. Calcular el kernel gaussiano (*G*) definido por la ecuación anterior.
2. Aplicar convolución sobre la imagen original con la máscara *G*.

$$J = I * G$$

Siendo **J** la imagen resultante de aplicar convolución, **I** la imagen original y **G** la máscara a utilizar.

3.1.1 Convolución

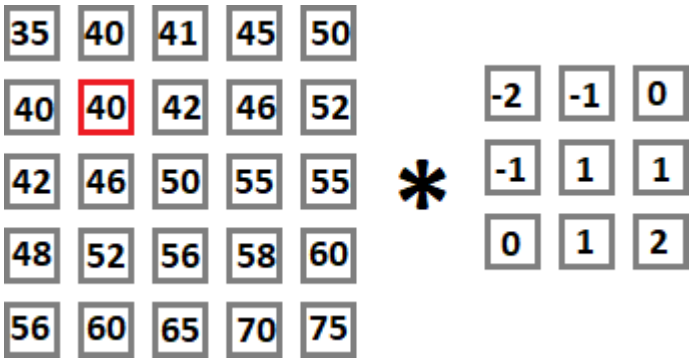
Acabamos de ver que es necesario aplicar convolución con una máscara Gaussiana sobre la imagen original, pero ¿qué es la convolución?

La convolución es el tratamiento de una matriz por otra que se llama “kernel”.

Matemáticamente es una transformación en que el valor del píxel resultante es una combinación lineal de los valores de los píxeles vecinos ya que se realiza sumando los productos obtenidos luego de multiplicar uno a uno los términos de la máscara de convolución con cada uno de los pixeles de la imagen.

Una idea general sobre la convolución sería pensar en pasar la máscara *kernel* para todo píxel de la imagen original, aplicando los coeficientes de la máscara según donde caigan.

Explicado gráficamente, teniendo la matriz de la imagen original definida por la matriz de la izquierda y aplicando convolución con la máscara de la derecha:



Los píxeles de los que coge la información la máscara son los marcados en naranja, donde coincide el centro de ambas matrices. Al utilizarse el centro del *kernel* es recomendable trabajar con matrices de tamaño impar, siendo lo más común tamaño 3 o 5:

Sobre ese píxel en concreto (en rojo) se realizan las operaciones que involucran a los demás píxeles según la máscara (naranja) para crear una nueva imagen donde esa misma posición tiene un valor diferente. Para cada píxel de la matriz original dentro del rango de la matriz *kernel* se multiplican los valores de las posiciones que coinciden. El valor del píxel estudiado será la sumatoria de todas las multiplicaciones anteriores. En este caso es 78 ya que:

$$35 * (-2) + 40 * (-1) + 41 * 0 + 40 * (-1) + 42 * 1 + 42 * 0 + 46 * 1 + 50 * 2 = 78$$

Esta misma acción se realizaría sobre todo los píxeles de la imagen dando como resultada una nueva matriz imagen.

Siendo el resultado:

35	40	41	45	50
40	40	42	46	52
42	46	50	55	55
48	52	56	58	60
56	60	65	70	75

*

-2	-1	0
-1	1	1
0	1	2

=

35	40	41	45	50
40	78	87	94	52
42	98	283	108	55
48	120	125	127	60
56	60	65	70	75

Para aplicar la convolución en los bordes (donde parte de la matriz *kernel* quedaría fuera de los límites de la imagen original) existen distintas opciones:

1. Completar con 0 los valores de alrededor
2. Repetir los valores del borde
3. Completar los valores con la parte simétrica opuesta

Para la implementación se ha decidido ignorar los píxeles en los cuales la matriz *kernel* salga de los límites de la imagen original. Debido a que la máscara de convolución tiene un tamaño generalmente de 3x3 o 5x5 la cantidad de píxeles sin tratar es ínfima en comparación con la cantidad total.

3.2 Calcular gradiente

Se podría decir que el gradiente es el nivel de gris en píxeles consecutivos en una imagen. Cualquier píxel de una imagen puede estar orientado hacia distintas direcciones, esta dirección se puede calcular gracias en primera instancia al gradiente.

De igual modo el algoritmo de *Canny* encuentra los bordes donde hay un cambio brusco de niveles de grises. Estos cambios bruscos también se pueden calcular computacionalmente gracias al gradiente.

Para cada píxel de la imagen se aplica un filtro mediante convolución para determinar cuán grande es el cambio que se produce alrededor del mismo píxel y por tanto poder determinar qué tan probable es que el píxel pertenezca a un borde.

Una vez obtengamos el gradiente podremos obtener la magnitud y orientación del vector gradiente en cada píxel.

3.2.1 Calcular gradiente X y gradiente Y

Para conocer la orientación es necesario realizar el gradiente para las direcciones en el plano X e Y.

Se calcula para cada píxel de la imagen el “peso” que tiene para la dirección X y para la dirección Y, es decir, “cuánto” tiende el pixel a la dirección X y “cuánto” a la dirección Y.

Hay distintos filtros posibles para aplicar, cada uno tiene ventajas e inconvenientes dependiendo de la imagen sobre las que se utilicen.

En el programa se han implementado los filtros de Sobel y Prewitt.

3.2.1.1 Sobel

La máscara de Sobel para el gradiente vertical y horizontal viene definida por las matrices:

$$Jx = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad Jy = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Sobel refuerza el valor del píxel central respecto al algoritmo de Prewitt que se verá a continuación, ya que el valor en dicho punto es 2.

Podemos ver que el eje central es 0 toda la fila o columna dependiendo de la orientación.

Para calcular el valor en dirección del X, el eje vertical es 0, mientras que para el valor Y es 0 el eje horizontal.

Este filtro se supone que es más sensible a los bordes diagonales que el de Prewitt aunque en la práctica la diferencia es mínima.

3.2.1.2 Prewitt

La máscara de Prewitt para el gradiente vertical y horizontal viene definida por las matrices:

$$Jx = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad Jy = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

En el operador de Prewitt se involucran a los vecinos de filas o columnas adyacentes para proporcionar mayor inmunidad al ruido.

Los valores centrales en este filtro son 1, a diferencia del anterior, que eran 2.

Estos dos filtros se pueden formular de forma conjunta con una máscara genérica con valores:

$$Jx = \begin{bmatrix} -1 & 0 & 1 \\ K & 0 & K \\ -1 & 0 & 1 \end{bmatrix} \qquad Jy = \begin{bmatrix} -1 & K & -1 \\ 0 & 0 & 0 \\ 1 & K & 1 \end{bmatrix}$$

Siendo K = 1 para el filtro de Prewitt y K = 2 para Sobel.

Esta máscara genérica facilitará el paso a código del programa.

3.2.2 Calcular magnitud de los bordes

La magnitud es un valor que cuantifica el grado de pertenencia de un píxel al borde. Cuanto mayor sea el valor "más" borde es el píxel.

Una vez se tiene para cada píxel el valor del gradiente, se puede calcular la magnitud de dicho píxel aplicando la distancia Euclídea:

$$|G| = \sqrt{Jx^2 + Jy^2}$$

Siendo **Jx** y **Jy** las matrices resultantes de aplicar convolución con uno de los filtros anteriores.

Cuanto mayor sea el valor resultante significa que el píxel estudiado tiene mayor pertenencia al borde.

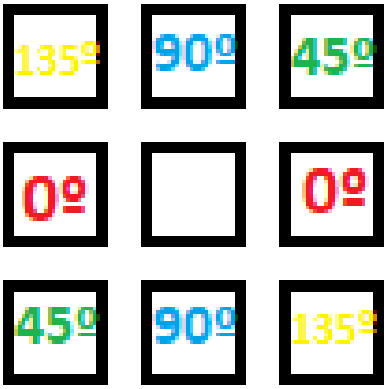
En ocasiones en lugar de utilizar la distancia Euclídea, para simplificar cálculos al computador, se aplica la distancia de Manhattan, definida por:

$$|G| = |Jx| + |Jy|$$

3.2.3 Estimar orientación

Conocida la magnitud se calcula la orientación para cada píxel.

Un píxel tiene 8 posibles direcciones definida por sus 8 vecinos. Para conocer la dirección de cada píxel primero se estima la orientación entre 4 posibles que representan a 0º, 45º, 90º y 135º respecto al eje horizontal:



La orientación de un píxel viene definida por la fórmula:

$$eo(i,j) = \arctan\left(\frac{Jy}{Jx}\right)$$

3.2.3.1.1 Estimar dirección según orientación

Una vez obtenida la orientación para cada píxel de la imagen hay que asociar dicho valor a una de las 4 posibles opciones. Para ello se calcula el ángulo al que se encuentre más cerca.

Es decir, si un píxel tiene un valor de, por ejemplo, 94º, se le asigna el ángulo 90º ya que es el más cercano que tiene.

La dirección no implica sentido, si un píxel tiene dirección 0º no se conoce si el sentido es hacia la izquierda o la derecha, ese valor se calculará más adelante según el peso de cada uno de los píxeles involucrados.

3.3 Supresión no máximos

El objetivo de este paso obtener bordes de 1 pixel de grosor al considerar únicamente los píxeles cuya magnitud es máxima en bordes gruesos y descartar aquellos cuyas magnitudes no alcancen ese máximo definido por los píxeles vecinos según la orientación o por peso con los vecinos cercanos.

3.3.1 Según orientación

En este punto se comprueba para cada píxel su magnitud con la magnitud de ambos vecinos según la dirección calculada anteriormente. Si el píxel en cuestión tiene una magnitud menor que al menos uno de sus dos vecinos, la matriz de no máximos tendrá un valor de 0 (suprimir), de lo contrario el valor será el de la magnitud de dicho gradiente (igualar).

3.3.2 Según cuadrantes [1]

Otro método implementado para crear la matriz de no máximos y que en la práctica ha dado resultados más precisos consiste en, a partir de una imagen conociendo el gradiente asignar pesos a los píxeles y comparar con sus vecinos según cuadrantes, determinando así la dirección que tendrá el borde y posteriormente igualar o suprimir.

Este método se ha implementado para recorrer cada píxel de la imagen comparando el valor de la matriz gradiente tanto X como Y. El punto mayor de entre estos dos será hacia donde tiende el píxel; si en un punto (m, n) la matriz GradienteX(m,n) > GradienteY(m,n) ese píxel tiende hacia X. De lo contrario hacia Y.

Si el punto tiende hacia X el peso del píxel actual es el valor del gradienteY/gradienteX; una vez calculado se guardan los vecinos en dirección horizontal, es decir, izquierda y derecha. De igual modo, si los valores de los gradientes tienden a la misma dirección (tienen el mismo signo -positivo o negativo-) se guardan los vecinos que se corresponden con la posición arriba e izquierda y abajo derecha del píxel que se estudia. Por el contrario, si no tienen el mismo signo los vecinos a comprobar serán los opuestos.

De otra forma, si tiende hacia Y se aplica la misma idea, calculando el peso como gradienteX/gradienteY y guardando primero los vecinos pertenecientes al eje vertical y después los otros dos vecinos resultantes según la dirección de ambos gradientes.

Para saber si los gradientes tienen o no el mismo signo se pueden multiplicar ambos valores y si el resultado es positivo (mayor que cero) ambos tendrán el mismo signo, de lo contrario, el valor resultante será negativo, menor que cero.

Una vez calculado esto se tiene el peso del píxel actual y los valores de las 4 posiciones a comprobar.

Para dicha comprobación se realiza una interpolación con los pares de posiciones correspondientes. Es decir, por ejemplo, si el gradiente de un píxel tiene a Y (los valores fijos son los vecinos de arriba y abajo), la interpolación se realizará del vecino de arriba con su vecino de izquierda o derecha y el vecino de abajo con su propio vecino de izquierda o derecha.

De ese modo se compara el valor del píxel actual con el valor de los 4 posibles vecinos, ya que primero cada par de vecinos se estudia entre ellos y el resultado se compara con el píxel local a estudiar.

Donde se aplica la fórmula:

$$f = peso * valorVariable + (1 - peso) * valorFijo$$

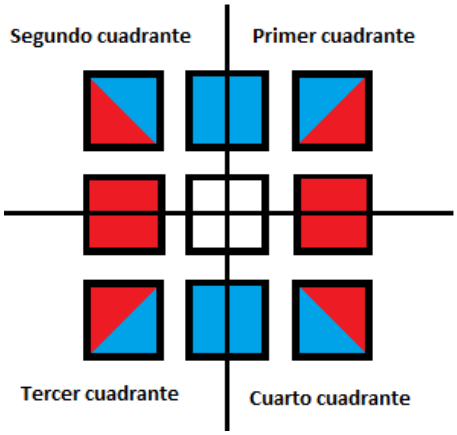
Siendo **valorFijo** el valor de la posición que corresponde con la dirección del gradiente (gradienteX arriba y abajo, gradienteY izquierda y derecha) calculado a partir de la división de ambos gradientes y **valorVariable** el peso de los vecinos según el signo de ambos gradientes calculado por la comparación mayor o menor que cero de la multiplicación.

Esta fórmula se calcula para ambas direcciones.

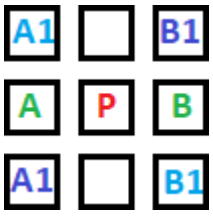
Si el valor del píxel actual es mayor que ambos valores calculados, entonces se considera como máximo local (igualar), de lo contrario se descarta (suprimir).

Este método es similar al anterior visto según orientación, pero en la práctica ha dado mejor resultado.

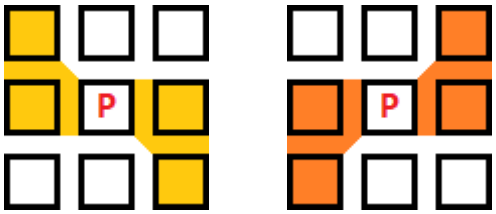
Gráficamente se puede ver de forma que según el peso del gradiente se actuará sobre uno de los cuatro cuadrantes posibles, siendo el color rojo los valores fijos para el eje X y el azul las posiciones fijas para el eje Y. Las posiciones variables se muestran con ambos colores:



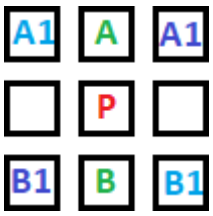
Así pues para la aplicación suponemos que tenemos el píxel **P**. Si su gradiente tiende hacia X, los valores fijos serán **A** y **B** (0º) y los valores variables serán uno de los dos pares **A1-B1** (45º o 135º) definidos por la multiplicación entre ellos si es mayor o menor que 0.



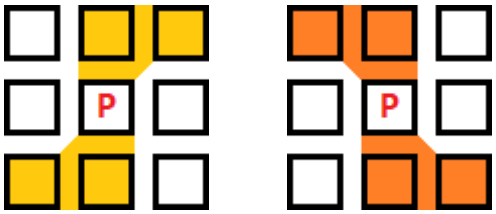
Es decir, se tienen estas dos posibles opciones (cuadrante 2/4 eje X o cuadrante 1/3 eje X):



Mientras que si su gradiente tiende al eje Y, los valores fijos serán los vecinos de arriba y abajo (90º) y los variables también cambiarán de la siguiente forma (45º o 135º):



Teniendo las dos posibles opciones (cuadrante1/3 eje Y o cuadrante 2/4 eje Y):



Se puede ver como de este modo, según el gradiente, se comprueban las 4 posiciones más posibles a formar parte del borde, mientras que siguiendo la orientación solo se estudiarían dos vecinos, lo cual requiere menos cálculos pero es menos preciso.

3.4 Histéresis de umbral a la supresión no máxima

La imagen obtenida en el paso anterior suele contener máximos locales creados por el ruido.

Una solución para eliminar dicho ruido es la histéresis del umbral. Esto consiste en tomar dos umbrales **m1** y **m2** siendo **m1** < **m2**. Para cada punto de la imagen se debe localizar el siguiente punto de borde no explorado que sea mayor a **m2**. A partir de dicho punto seguir las cadenas de máximos locales (valores superiores a **m1**) conectados con el píxel en cuestión.

De este modo se recorre en la imagen a partir de un punto localizado como borde (mayor que **m2**) todos los píxeles contiguos, que serán marcados también como bordes si superan el umbral **m1**.

Para seguir la cadena a partir de un punto localizado como máximo se han programado dos opciones, según orientación de los bordes y comprobando todos los vecinos de dicho punto.

3.4.1 Cadena de mínimos y máximos según orientación

Esta opción aprovecha la estimación de los bordes calculada anteriormente para, a partir de un punto localizado como máximo buscar únicamente en los píxeles conectados en ambas direcciones perpendiculares a la normal del borde según una orientación determinada.

De esta forma de 8 posibles direcciones solo es necesario estudiar 2 lo que ahorra tiempo de computación, pero puede perderse información en el proceso.

3.4.2 Cadena de mínimos y máximos de todos los vecinos

En esta segunda opción, una vez que se encuentra un punto máximo localizado como borde (mayor que **m2**) se busca entre los 8 vecinos un píxel superior a **m1** para seguir la cadena.

Con esta opción se busca entre todos los píxeles vecinos por lo que la probabilidad de errar es menor a costa de un mayor tiempo de ejecución.

3.4.3 Unión de bordes

Este paso se realiza únicamente tras el método basado en orientación, donde una vez terminado el paso anterior se realiza otra pasada por la imagen incorporando como bordes finales aquellos bordes débiles que están conectados a bordes fuertes en un rango de 3x3.

4 Implementación

A continuación se muestra el proceso llevado a cabo para implementar todos los pasos teóricos explicados anteriormente utilizando el lenguaje de programación c++.

El único requisito es utilizar una imagen en escala de grises (8 bits) como entrada.

4.1 Idea general del funcionamiento

El programa consta de dos partes diferenciables. La primera mantiene interacción con el usuario para pedir valores de entrada. En la segunda parte el programa de forma autónoma realiza los pasos iterativamente hasta formar la imagen resultante.

En la medida de los posible se han guardado los valores referentes a la matriz en variables de tipo `C_Matrix::ElementT` para evitar lo máximo posible pérdida de información, utilizando los tipos nativos `int`, `double` solo en casos necesarios o cuando no se trataba con un valor de matriz.

4.2 Métodos y variables

Se ha basado el programa en utilización de métodos para extraer del método principal procesos iterativos. Siendo estos:

`int main(int argc, char **argv)` donde se inicializa el programa y se interacciona con el usuario. Una vez introducidos los datos se inicia el tratamiento de la imagen llamando al método `programa`

`void programa(char entrada[], char salida[])` es donde empieza el tratamiento de la imagen. Las cadenas por parámetro son el nombre de la imagen original y la imagen de salida. Desde aquí se realizarán las acciones o llamadas necesarias para aplicar el algoritmo correctamente.

`C_Matrix convolucion(C_Matrix m1, C_Matrix m2)` realiza una convolución sobre la matriz `m1` usando la máscara `m2` devolviendo una matriz convolucionada.

`int direccion_cercana(C_Matrix::ElementT f)` comprueba a qué ángulo se acerca más (0º, 45º, 90º o 135º) el valor de entrada `f`.

`void crear_matriz_nomax_orientacion(int i, int j)` crea la matriz de no máximos igualando o suprimiendo según la orientación para cada píxel de entrada `i`, `j`, que corresponden con la posición (fila, columna) del píxel a estudiar

`void crear_matriz_nomax(int i, int j)` forma la matriz de no máximos igualando o suprimiendo según el método de gradiente y contiguos, donde las entradas `i`, `j` corresponden con la posición (fila, columna) del píxel a estudiar.

`void seguir_cadena_m2(int i, int j)` crea la imagen final binaria siguiendo el umbral predefinido, accediendo recursivamente a las posiciones donde se cumplen las condiciones de umbral. Los valores de entrada son la posición (fila, columna) de la imagen a tratar.

`void seguir_cadena_orientacion(int i, int j)` crea la imagen final binaria siguiendo los valores del umbral, comprobando únicamente los vecinos según la orientación y accediendo a los que cumplan las condiciones. Los valores de entrada son la posición (fila, columna) de la imagen a tratar.

`void juntar_contornos(int i, int j)` añade como borde aquellos contornos débiles en contacto con un píxel catalogado como borde fuerte.

Para simplificar llamadas y retornos de los métodos se han utilizado generalmente atributos globales. Así pues, por ejemplo, no es necesario pasar el valor de los umbrales del método `main(…)` a `programa(…)` y a su vez a `seguir_cadena_m2(…)` y a `seguir_cadena_orientacion(…)`.

Gran parte de las matrices a utilizar en el proceso basan su tamaño en el de la imagen original a procesar, es por eso que estas matrices se declaran globales y una vez conocido el tamaño deseado se hace uso del método `Resize(imagen.FirstRow(), imagen.LastRow(), imagen.FirstCol(), imagen.LastCol(), 255)` donde `imagen` es la matriz que contiene la imagen original para así poder tener lo mismos índices de acceso tanto para filas como para columnas.

4.3 Introducción de datos por pantalla

Para una mayor flexibilidad a la hora de usar el programa se ha permitido al usuario introducir valores por pantalla. Estos valores son:

- Nombre imagen de entrada *
- Nombre imagen de salida *
- Valor de sigma
- Tamaño del kernel gaussiano (menor que el tamaño de la imagen original) **
- Valor de umbral máximo
- Valor de umbral mínimo (menor que el máximo)
- Máscara para el gradiente (Prewitt o Sobel)
- Rutina de ejecución (Orientación o comprobar 8 vecinos)

*Es importante no olvidar el tipo de archivo (p.ej. archivo.bmp)

**Recomendable valores impares, generalmente tamaño 3 o 5.

4.4 Cálculo Kernel Gaussiano

El kernel gaussiano permite suavizar la imagen y será el primer paso del algoritmo. Para calcularlo se ha implementado la ecuación presentada anteriormente, cogiendo los valores del tamaño de la matriz y de sigma introducidos por el usuario.

Con un bucle anidado se recorre la matriz entera para cada fila y cada columna añadiendo los valores correspondientes a cada posición definidos por la ecuación:

```
(1 / ((2 * M_PI)*pow(sigma,2)))*exp(-((pow(i,2)+pow(j,2)) / (2*pow(sigma, 2))));
```

Así para cada posición del kernel se añade el valor resultante de la ecuación.

4.5 Convolución

El proceso de convolución ha sido implementado en un método de manera que los parámetros de entrada son la matriz a convolucionar y la máscara a aplicar; el retorno es la matriz ya convolucionada.

Un punto clave de la implementación ha sido obviar los píxeles de la imagen donde la máscara estaría fuera de rango, de manera que dichos píxeles tienen el valor original. Para realizarlo se ha declarado una variable *center* que realiza una división entera sobre el tamaño del kernel, por lo que se consigue el tamaño y así se puede conocer el número de píxeles que se deben dejar de margen.

Seguidamente se ejecuta un bucle que recorre la matriz a convolucionar. Para cada valor se comprueba si está dentro del margen anteriormente calculado, donde la matriz se saldría de los límites. En caso afirmativo se mantiene el valor del píxel original y salta a la siguiente iteración

En caso de que no se cumplan las condiciones entra en ejecución otro bucle anidado del tamaño del kernel gaussiano, en el cual se van sumando los valores de las posiciones correspondientes tanto de la imagen original como del kernel.

Una vez terminado este último bucle se actualiza el píxel sobre el que se ha ejecutado la convolución con la sumatoria de los valores calculados.

A grandes rasgos el algoritmo se basa en:

1. Conocer el número de píxel a dejar de margen
2. Recorrer imagen original
3. Dejar los píxeles dentro del margen (próximos a los bordes) con el valor original
4. Convolucionar los píxeles fuera del margen que quedaría fuera del rango.
   Siendo el último paso un bucle anidado únicamente del tamaño de la máscara con el siguiente comportamiento:

   ```
   sumatoria_convolucion +=
          m1(i + k - center - m2.FirstRow(), j + l - center - m2.FirstCol())*
          m2(k, l);
   ```

   Dentro del cual se resta el valor *center* anteriormente calculado para recorrer todas las posiciones del kernel.
   Al ser el bucle una ejecución con valores 1..*n,* siendo *n* el tamaño del kernel, restar el centro del *kernel* es dividir *n/2* con lo se consigue que la ejecución en lugar de ser

desde un píxel *n* hasta un píxel *m* se realiza desde *-n/2 … n/2* lo que deja al píxel a estudiar en el centro.
Se aplica tanto a filas como a columnas.

La primera llamada al método en el programa es para convolucionar la imagen original con el kernel gaussiano y poder proceder a calcular el gradiente.

4.6 Gradiente ejes X e Y

Con el suavizado de la imagen aplicando convolución con la máscara gaussiana creada anteriormente sobre la imagen original se procede a calcular el gradiente para los ejes X e Y. Para no modificar la imagen original se ha creado una matriz auxiliar `C_Matrix matriz_J` de igual tamaño que la imagen original para guardar la imagen suavizada.

Seguidamente se realiza una nueva convolución tanto para el eje X como Y con una máscara elegida por el usuario.

Según la máscara elegida la matriz que la contiene tendrá unos valores u otros, por lo que es necesario indicar dichos valores. Las máscaras posibles para utilizar son de Sobel y Prewitt.

Como se explicó en la parte teórica, ambas máscaras tienen un patrón de construcción que se puede aprovechar para utilizar una variable `k` a la que asignarle el valor 1 o 2, según la máscara elegida.

Una vez terminadas las convoluciones para los ejes X e Y se han generado dos matrices que contienen ambas convoluciones, `matriz_Jx` para el eje X y `matriz_Jy` para el eje Y.

4.7 Magnitud de los bordes

Disponiendo de los valores del gradiente para ambos ejes se calcula la magnitud de los bordes para cada píxel. Esto quiere decir que será necesario un bucle anidado del tamaño de la imagen original donde en cada píxel se realiza la distancia Euclídea.

```
matriz_es(i, j) = sqrt((pow(matriz_Jx(i, j), 2) + (pow(matriz_Jy(i, j), 2))));
```

Siendo `matriz_es` la matriz donde guardar los resultados.

4.8 Método1: Siguiendo orientación

A partir de este punto el programa se bifurca según la opción introducida por el usuario.

El método 1 se basa en comprobar los dos vecinos que se encuentran en las posiciones referentes a la orientación.

Para ello el primer paso es estimar dicha orientación.

4.8.1 Estimar orientación de bordes

Para estimar la orientación se hace uso de la arcotangente entre gradientes.

Se recorre en un bucle anidado el tamaño de la imagen original y para cada píxel se añade a una matriz auxiliar el valor calculado.

En el programa se ha llamado a la matriz donde guardar los resultados `matriz_eo()` y se ha codificado la ecuación de la siguiente manera:

```
matriz_eo(i, j) = atan(matriz_Jy(i, j) / matriz_Jx(i, j));
```

4.8.2 Estimar dirección según la orientación

Conocida la orientación se tienen que redondear los valores a los ángulos 0º, 45º, 90º, 135º.

Con un nuevo bucle anidado se guardan en `matriz_direccion` los valores devueltos por el método `direccion_cercana(C_Matrix::ElementT f)`

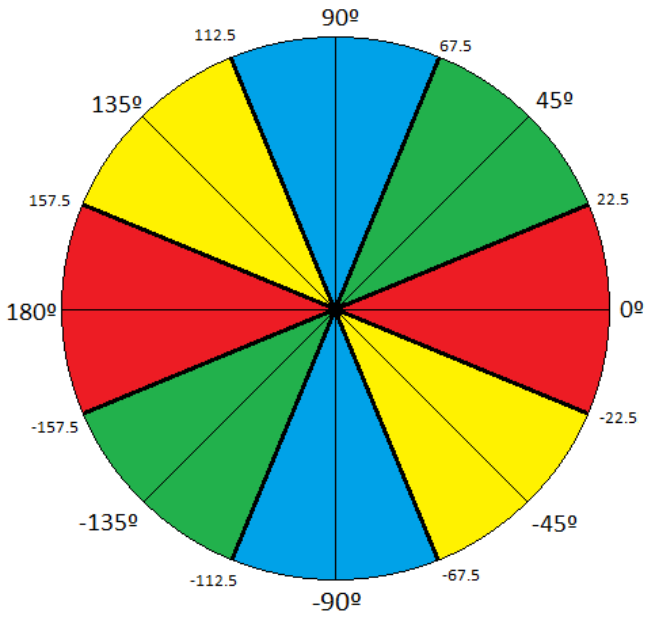
La llamada al método se realiza:

```
matriz_direccion(i, j) = direccion_cercana(matriz_eo(i, j));
```

Así para cada elemento de la matriz de orientación se calcula el ángulo al que correspnde.

Dentro del método `direccion_cercana` se calcula el ángulo y se realiza una estructura de control para delimitar la posición del valor según su cercanía.

Para determinar la cercanía se han elegido los puntos medios entre cada ángulo contiguo quedando los 360º divididos tal que:



Perteneciendo al grupo de:

- 0º: Ángulos comprendidos entre 22.5º y -22.5º o entre de 157.5º y -157.5º
  `(angulo < 22.5 && angulo > -22.5) || (angulo > 157.5 && angulo < -157.5)`
- 45º : Ángulos comprendidos entre 22.5º y 67.5º o entre -112.5º y -157.5º
  `(angulo > 22.5 && angulo < 67.5) || (angulo < -112.5 && angulo > -157.5)`
- 90º: Ángulos comprendidos entre 67.5º y 112.5º o entre -6.75º y -112.5º
  `(angulo > 67.5 && angulo < 112.5) || (angulo < -67.5 && angulo > -112.5)`
- 135º: Ángulos comprendidos entre 112.5º y 157.5º o entre -22.5º y -67.5º
  `(angulo > 112.5 && angulo < 157.5) || (angulo < -22.5 && angulo > -67.5)`

De esta manera todos los ángulos quedan reconocidos por su dirección más cercana.

4.8.3 Matriz no máximos según orientación

En este punto el objetivo es igualar o suprimir píxeles; si la magnitud de un píxel es menor que uno de sus dos vecinos según la orientación se suprime, la matriz de no máximos pasa a tener valor 0, de lo contrario se iguala.

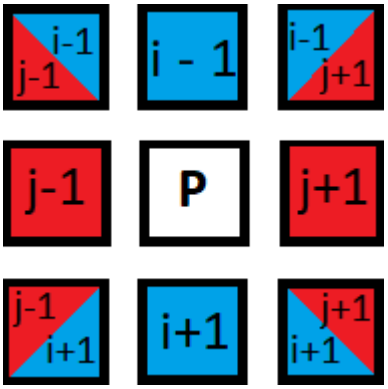
Para ello el primer paso es conocer la dirección que tiene el píxel a estudiar. Se puede acceder fácilmente al valor ya que se ha calculado anteriormente y se ha almacenado en la matriz `matriz_direccion()`, para ello se guarda el valor en una variable que será 0, 45, 90 o 135:

```
int direccion = matriz_direccion(i, j);
```

Conocida la dirección se comprueba con sus vecinos correspondientes.

Cada dirección tiene que comprobar sus vecinos correspondientes. Las filas están definidas por la variable `i` y las columnas por `j`, por lo que se puede acceder a todas las posiciones con la combinación de valores de ±1 y ±0 para los valores de filas y columnas.

Para verlo más claro se puede observar los valores de fila y columna necesarios para acceder a cada vecino a partir de un píxel **P**.



Sabiendo como acceder a los vecinos se realiza una estructura de control en la que para cada orientación se comprueba las posiciones correspondientes.

0º: Vecinos de izquierda y derecha; misma fila y una columna más o una menos

```
(matriz_es(i, j) < matriz_es(i, j - 1) || matriz_es(i, j) < matriz_es(i, j + 1))
```

90º: Vecinos de arriba ya abajo; misma columna una fila superior o inferior

```
(matriz_es(i, j) < matriz_es(i - 1, j) || matriz_es(i, j) < matriz_es(i + 1, j))
```

45º: Vecino de arriba derecha y abajo izquierda; hay que variar filas y columnas

```
(matriz_es(i, j) < matriz_es(i - 1, j + 1) ||
 matriz_es(i, j) < matriz_es(i + 1, j - 1))
```

135º: Vecino de arriba izquierda y abajo derecha; hay que variar filas y columnas

```
(matriz_es(i, j) < matriz_es(i - 1, j - 1) ||
 matriz_es(i, j) < matriz_es(i + 1, j + 1))
```

Si cualquiera de estas igualdades es cierta, el píxel se suprime: `matriz_nomax(i, j) = 0;`

De lo contrario se iguala: `matriz_nomax(i, j) = matriz_es(i, j);`

4.8.4 Histéresis según umbral de máximos y mínimos según orientación

Una vez formada la matriz de no máximos se recorre entera y para cada píxel se comprueba si su valor está por encima del umbral máximo. En caso afirmativo se llama al método `seguir_cadena_orientacion(int i, int j)` a partir del cual se sigue la cadena de valores por encima del umbral mínimo marcándolos como borde.

4.8.4.1.1 Seguir cadena de mínimos y máximos según orientación

Al entrar al método se comprueba la matriz de visitados y se actualiza para no iterar dos veces en la misma posición. De igual manera se actualiza el punto actual como borde, ya que si ha entrado es porque supera los valores mínimos según el umbral.

```
if (matriz_visitados(i, j) == 1)return;  //Píxel ya estudiado

        matriz_visitados(i, j) == 1;      //Visitado

   matriz_umbral(i, j) = 255;        //Marcado como borde
```

La cadena tiene que seguir según la dirección a la que tiende, por lo que se utiliza la matriz donde estaban almacenados los valores de la dirección y se comprueban los dos vecinos según esa dirección.

Se ha seguido la misma idea para acceder al vecino en concreto que para crear la matriz de no máximos. Partiendo de un píxel, los vecinos se encuentran en una posición de ±1 y ±0 para las filas y las columnas.

Se utilizan 4 variables auxiliares que se actualizan según los vecinos a estudiar y se llama al método recursivamente con esas dos nuevas posiciones como parámetros de entrada.

Así pues, por ejemplo, teniendo las variables declaradas:

```
int aux_x1, aux_y1, aux_x2, aux_y2;
```

Si el píxel tiene una dirección de 0º se actualizarán los valores para el vecino de la izquierda (i+0, j-1) y el vecino de la derecha (i+0, j+1) tal que:

```
aux_x1 = 0; aux_x2 = 0; aux_y1 = -1; aux_y2 = 1;
```

Antes de llamarse recursivamente es necesario hacer una comprobación para saber si el punto supera el umbral mínimo, ya que la implementación supone que una vez que entra al método es porque ha superado el umbral y por tanto se considera borde.

```
if (matriz_nomax(i + aux_x1, j + aux_y1) >= u_min)

if (matriz_nomax(i + aux_x2, j + aux_y2) >= u_min)
```

Terminada la comprobación para cada vecino puede llamarse recursivamente entrando en las dos nuevas posiciones:

```
seguir_cadena_orientacion(i + aux_x1, j + aux_y1)
seguir_cadena_orientacion(i + aux_x2, j + aux_y2)
```

Una vez finalizada la iteración se tiene una matriz binaria (0,255) donde se marcan los bordes encontrados por el algoritmo.

4.8.5 Unión de bordes

Tras una primera pasada y finalizado el seguimiento de la cadena por valores mayores que los umbrales predefinidos se realiza otra iteración sobre todos los píxeles de la imagen buscando aquellos que tengan un valor mayor o igual que el umbral mínimo. Para cada uno de estos píxeles se realiza una búsqueda en un margen de 3x3, es decir, sobre los píxeles vecinos buscando un píxel por encima del umbral. En caso de encontrarlo el píxel se considera borde.

Para la implementación se itera sobre todos los píxeles y se llama al método `juntar_contornos(int i, int j)` si el píxel actual es mayor que el umbral mínimo. Una vez dentro del método se ha generado un bucle que itere sobre la matriz de tamaño 3x3 en búsqueda de un borde.

Si alguno de estos píxeles en el rango 3x3 es mayor o igual que el umbral mínimo se añade el píxel como borde

```
if (matriz_nomax(i + k, j + l) >= u_min) matriz_umbral(i, j) = 255;
```

La iteración se realiza en el rango (-1 .. 1) tanto para filas como para columnas para acceder a todas las posiciones posibles definidas por los 8 vecinos. Así pues si estamos estudiando un pixel **P**, el valor inicial será tanto para filas como para columnas (-1,-1) y el valor final (+1,+1), pasando entre medias por todas las posiciones posibles. En el caso de la posición (0,0) se continua la iteración pues esa posición es la del píxel actual y no es necesario estudiarla.

4.9 Método2: Cuadrantes

Otro método implementado ha sido comprobando 4 vecinos en lugar de únicamente 2 según cuadrantes. El último punto en común entre ambos métodos es la creación de la matriz de magnitud para los bordes `matriz_es`, por lo que hasta este punto también se tiene las matrices del gradiente `matriz_Jx` y `matriz_Jy`.

Por lo tanto el siguiente paso, al igual que el método anterior, es calcular la matriz de no máximos, pero en este caso de una manera distinta.

4.9.1 Matriz no máximos según cuadrantes

Dentro del método el primer paso es calcular la tendencia según el gradiente, si es hacia el eje X o el eje Y. Para ello se comparan ambos valores en valor absoluto ya que el píxel puede tender a valores negativos (por ej. -90º).

Una vez conocida la dirección se aplica la teoría explicada anteriormente calculando el peso del píxel a comparar dividiendo los valores de los gradientes.

Sabiendo el peso del píxel central se comparan los vecinos “fijos” con sus contiguos. Para el eje Y sus vecinos fijos son el de arriba y el de abajo mientras que para el eje X los de izquierda y derecha.

Esos vecinos fijos a su vez son comparados con su par de vecinos “variables” en función de la dirección; positiva multiplicación entre gradientes mayor que 0 o negativa menor que 0.

Una vez que se tienen los vecinos con los que se compara el píxel se interpolan entre ellos:

```
aux1 = peso*v1 + (1 - peso)*v2;
aux2 = peso*v3 + (1 - peso)*v4;
```

Siendo `peso` la división entre gradientes.

En las variables `aux1` y `aux2` se almacena el valor a comparar con el píxel original a estudiar, por lo que se compara la posición actual con ambas variables auxiliares para suprimir o igualar en la matriz de no máximos.

```
if (aux >= aux1 && aux >= aux2) {
    matriz_nomax(i, j) = aux;
}else if (aux < aux1 || aux < aux2) {
     matriz_nomax(i, j) = 0;}
```

Donde `aux` es `matriz_es(i, j)` , es decir, el valor de la magnitud del borde actual.

4.9.2 Histéresis según umbral de máximos y mínimos siguiendo los 8 vecinos

Con la matriz de no máximos creada se itera sobre cada píxel comprobando si supera el umbral máximo predefinido para ser considerado como borde, de ser así se llama al método `seguir_cadena_m2(int i, int j)` en el que se sigue recursivamente la cadena de valores por encima del umbral mínimo.

4.9.2.1.1 Seguir cadena de mínimos y máximos siguiendo los 8 vecinos

La idea y estructura de este método es similar al método de seguir la cadena por orientación.

En primer lugar se actualiza la matriz de visitados y se marca el punto actual como borde. Seguidamente se inicia un bucle para recorrer todos los vecinos.

Para una fácil ejecución se han guardado todo par de vecinos en dos vectores para acceder iterativamente a todos los vecinos; esto es:

Valores fila:	1	1	0	-1	-1	-1	0	1
Valores columna:	0	1	1	1	0	-1	-1	-1
Par de valores sumados:	1+0	1+1	0+1	-1+1	-1+0	-1-1	0-1	1-1

Los valores finales se corresponden con las posiciones de los 8 vecinos de un píxel cualquiera.

Para acceder a cada uno de los vecinos se itera en un bucle 8 veces (una para cada vecino) en el que se almacena el peso del vecino en cuestión según la matriz de no máximos calculada anteriormente.

```
valor = matriz_nomax(i + pos_x[k], j + pos_y[k]);
```

Seguidamente se compara dicho valor con el umbral mínimo. En caso de que sea mayor o igual se llama iterativamente al método con la nueva posición, comprobando así todos los vecinos del píxel en cuestión.

```
if (valor >= u_min) seguir_cadena_m2(i + pos_x[k], j + pos_y[k]);
```

5 Resultados finales

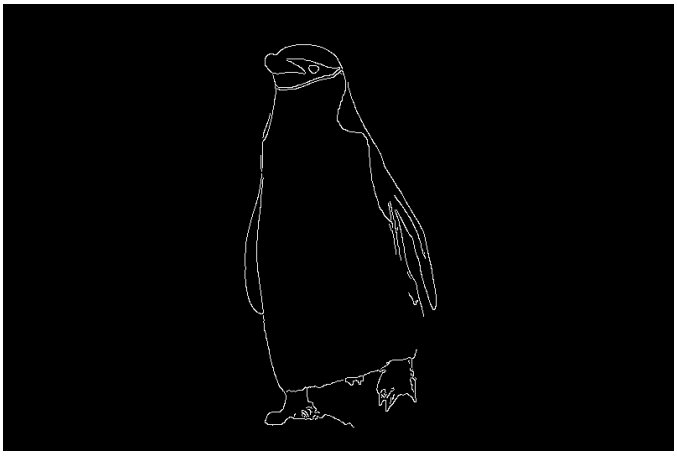
Terminado el proceso de implementación se pueden obtener los resultados a partir de una imagen de entrada en escala de grises.

A continuación se exponen diferentes salidas según distintos imágenes y parámetros de entrada. Se ha intentado coger valores límite para mostrar las diferencias para cada opción, filtro o umbral introducido por el usuario ya que dos imágenes iguales tendrán distinta salida aplicando el mismo algoritmo si los valores de entrada son dispares.

Todas las imágenes que se muestran a continuación se encuentran adjuntas a tamaño original.

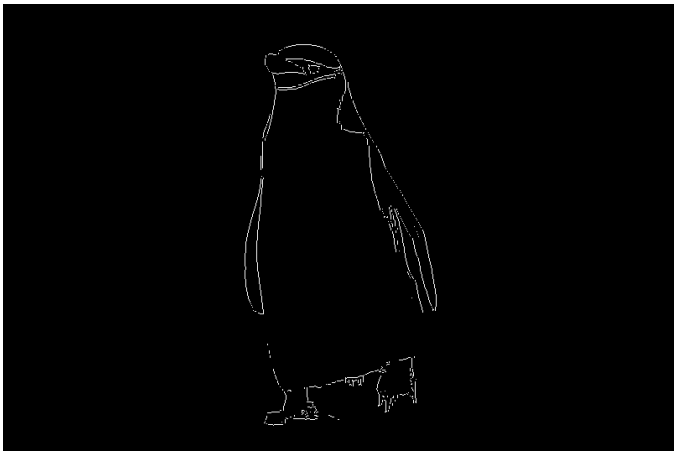
Imagen: **penguin.bmp**

penguin.bmp



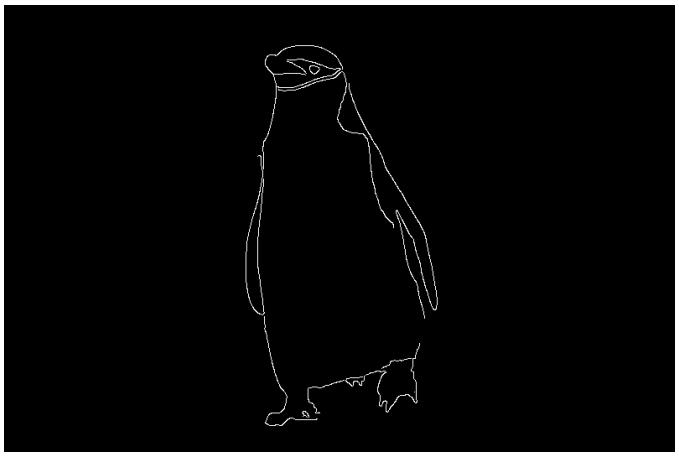
penguin1.bmp

σ: 1.5
Kernel Gauss: 5
Umbral máx: 140
Umbral min: 80
Máscara: Sobel
Método: 8 vecinos



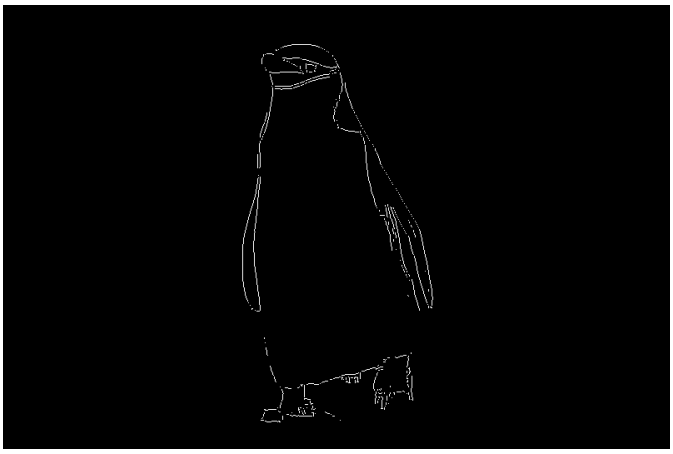
penguin2.bmp

σ: 1.5
Kernel Gauss: 5
Umbral máx: 140
Umbral min: 80
Máscara: Sobel
Método: Orientación



penguin3.bmp

σ: 2.5
Kernel Gauss: 5
Umbral máx: 180
Umbral min: 40
Máscara: Prewitt
Método: 8 vecinos



penguin4.bmp

σ: 4.5
Kernel Gauss: 3
Umbral máx: 120
Umbral min: 60
Máscara: Prewitt
Método: Orientacion

En estas imágenes el horizonte, pese a juntar el blanco de la nieve con un nivel de gris más oscuro del fondo no es captado como borde por el algoritmo ya que los niveles de grises de ambas posiciones se van igualando paulatinamente y al aplicar el suavizado de la imagen con el kernel gaussiano difumina el fondo de tal manera que no hay un salto de niveles de grises tan alto como para ser considerado borde.

Imagen: **panda.bmp**

panda.bmp

panda1.bmp

σ: 1.5
Kernel Gauss: 5
Umbral máx: 140
Umbral min: 60
Máscara: Prewitt
Método: 8 vecinos

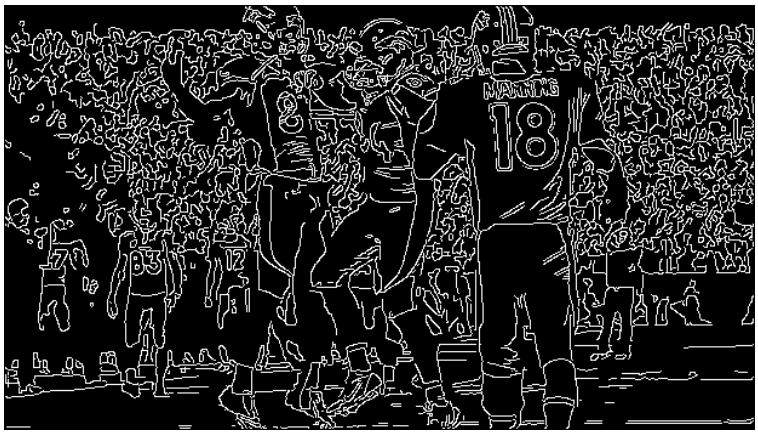
panda2.bmp

σ: 4
Kernel Gauss: 3
Umbral máx: 200
Umbral min: 100
Máscara: Prewitt
Método: 8 vecinos

Los valores de entrada de estas imágenes de ejemplo son dispares. La primera imagen tiene valores más bajos tanto de sigma como de umbrales, lo que permite obtener más detalles, mientras que la segunda imagen pese a ser más restrictiva en los umbrales mantiene la estructura y se distingue el animal, aunque con menos detalles.

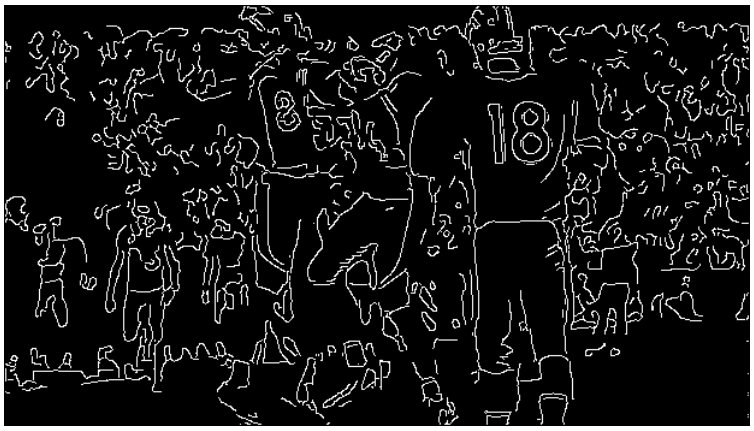
Imagen: **football.bmp**

football.bmp



football1.bmp

σ: 1.5
Kernel Gauss: 5
Umbral máx: 180
Umbral min: 120
Máscara: Sobel
Método: 8 vecinos



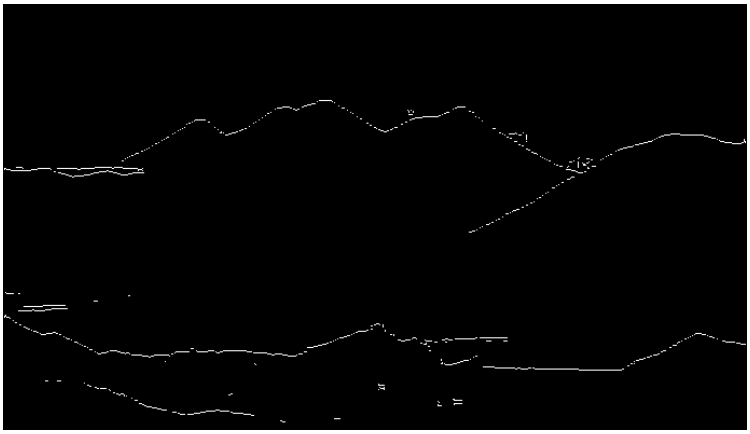
football2.bmp

σ: 5
Kernel Gauss: 5
Umbral máx: 180
Umbral min: 120
Máscara: Sobel
Método: 8 vecinos

En este ejemplo se ha variado el valor de sigma manteniendo los demás valores iguales para ver la diferencia entre ambas salidas. Como se ha explicado anteriormente, cuanto mayor es el nivel de la desviación estándar más ruido se elimina de la imagen, que a la vez se "nubla". Es por eso que en estas imágenes el público (al fondo) pierde contraste, y por tanto bordes, ya que al difuminarse quedan los píxeles más unificados o más similares, mientras que los jugadores que se encuentras más cerca se siguen distinguiendo junto a sus números.

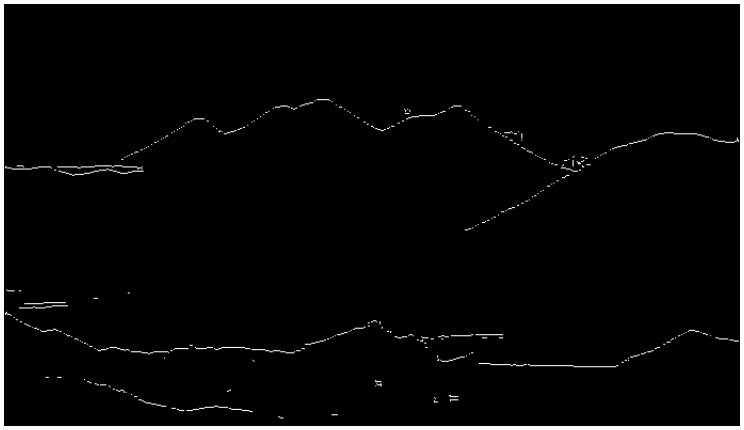
Imagen: **mountains.bmp**

mountains.bmp



mountainsPrewitt.bmp

σ: 1.5
Kernel Gauss: 5
Umbral máx: 100
Umbral min: 40
Máscara: Prewitt
Método: Orientación



mountainsSobel.bmp

σ: 1.5
Kernel Gauss: 5
Umbral máx: 100
Umbral min: 40
Máscara: Sobel
Método: Orientación

En teoría el filtro de Sobel es más sensible a los bordes diagonales. Para estas imágenes se han utilizado los mismos valores de entrada excepto la máscara y la diferencia es inapreciable.

Imagen: **matricula.bmp**

matricula.bmp

matriculaOrientacion.bmp

matriculaOrientacion	**matricula8vecinos**
σ: 1.5	σ: 1.5
Kernel Gauss: 5	Kernel Gauss: 5
Umbral máx: 140	Umbral máx: 140
Umbral min: 100	Umbral min: 100
Máscara: Prewitt	Máscara: Prewitt
Método: Orientación	Método: 8 vecinos

matricula8vecinos.bmp

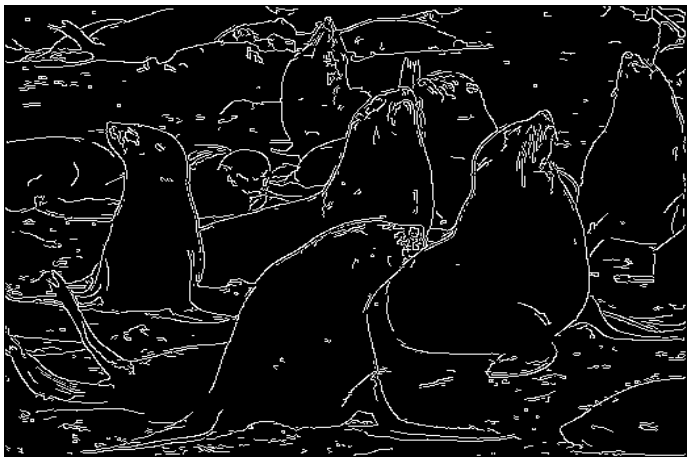
Como era de esperar el método de cuadrantes y comparación de los 8 vecinos muestra bordes más nítidos a costa de mayor tiempo de ejecución. En este ejemplo cualquiera de los dos métodos podría ser útil ya que los números y letras están bien definidas con cualquiera de ellos.

Imagen: **seals.bmp**

seals.bmp

seals1.bmp

σ: 3
Kernel Gauss: 5
Umbral máx: 140
Umbral min: 80
Máscara: Sobel
Método: 8 vecinos



seals2.bmp

σ: 0.1
Kernel Gauss: 5
Umbral máx: 140
Umbral min: 80
Máscara: Sobel
Método: 8 vecinos

En este ejemplo variando únicamente la desviación estándar tenemos más o menos contraste tanto para los detalles faciales como con el fondo de la imagen. La ausencia de kernel muestra más detalles y un valor excesivo para la desviación estándar se traduce a pérdida de información. Dependiendo de la utilidad pueden ser necesarios los detalles o no. También se adjunta una imagen (**seals3.bmp**) con un punto medio para la desviación estándar σ = 1.5

6 Conclusión

Teniendo en cuenta el incremento de utilización de la visión artificial, el reconocimiento de bordes es un paso fundamental y computacionalmente posible como se muestra en este documento donde se ha presentado la aplicación teórica y práctica de detección de bordes mediante el algoritmo de Canny, en el cual su mayor virtud, la versatilidad, puede ser a su vez un defecto ya que se puede aplicar una configuración diferente para cada tipo de imagen a tratar, como se ha mostrado en los ejemplos finales, pudiendo obtener distintos resultados según la finalidad del usuario para cualquier tipo de imagen pero todo esto a expensas de interacción humana y por tanto perder la automatización completa.

A diferencia de otros métodos más simples basados únicamente en filtros como Sobel o Prewitt, el *algoritmo de Canny* añade más pasos para un mejor resultado a costa de más tiempo de ejecución. Uno de esos pasos es la eliminación del ruido de la imagen, donde una mala elección de la desviación estándar puede ocasionar que se difuminen los bordes y por tanto perder información para calcular orientación y peso.

Se debe tener en cuenta que según los valores de entrada, el algoritmo genera una salida diferente, una misma imagen procesada dos veces tendrá resultados diferentes si los parámetros no son iguales. De igual modo, unos mismos parámetros pueden dar un resultado satisfactorio sobre una imagen y no sobre otra. Es por eso que la interacción humana se hace necesaria.

Aún así los resultados finales obtenidos son muy satisfactorios con cualquiera de los dos métodos implementados, siendo mejor el método basado en división por cuadrantes, interpolación y recorrido de los ocho vecinos, aunque ambos dieron resultados notorios como se ha mostrado en la sección anterior.

Por lo visto y expuesto, el *Algoritmo de Canny* es un algoritmo potente con resultados superlativos por lo que se considera uno de los mejores métodos para la detección de bordes ya que también es un algoritmo sencillo de implementar. Sin embargo no se puede afirmar que sea el mejor ya que se deben probar distintos algoritmos sobre las imágenes que se quieran procesar para conocer cuál de ellos se adapta mejor al problema.

7 Bibliografía

[1] Wang, X., & Jin, J. Q. (2007, October). *An edge detection algorithm based on improved Canny operator*. In *Intelligent Systems Design and Applications, 2007. ISDA 2007. Seventh International Conference on* (pp. 623-628). IEEE. Disponible en: https://ieeexplore.ieee.org/document/4389677

[2] REBAZA, Jorge Valverde. *Detección de bordes mediante el algoritmo de Canny*. *Escuela Académico Profesional di Informática. Universidad Nacional de Trujillo*, 2007 [fecha de consulta mayo 2018]. Disponible en : https://www.researchgate.net/publication/267240432_Deteccion_de_bordes_mediante_el_algoritmo_de_Canny.pdf

[3] ORTIZ, C. A. *Metodología para la construcción de indicadores morfodinámicos a través del uso de cámaras de vídeo. Caso de aplicación: Playa de La Magdalena. (Cantabria, España)*. Universidad Nacional de Colombia, Facultad de Minas. 2009 [fecha de consulta mayo 2018]. Disponible en: http://www.bdigital.unal.edu.co/5258/4/Cap.4a.pdf (páginas 7-10)

[4] ZIOU, Djemel, et al. *Edge detection techniques-an overview*. *Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii*, 1998, vol. 8, p. 537-559. [fecha de consulta mayo 2018] Disponible en: https://pdfs.semanticscholar.org/587a/acc01a4c33f0fe7fb172f5db785f40522b57.pdf

[5] AZEEZULLAH, S. I. & Huynh D. A. *Canny Edge Detection*. Massey University of New Zealand [fecha de consulta mayo 2018]. Disponible en: http://www.massey.ac.nz/~mjjohnso/notes/59731/presentations/img_proc.PDF

[6] KIM, D. *Sobel Operator and Canny Edge Detector*. Michigan State University, 2013 [fecha de consulta mayo 2018]. Disponible en: https://www.egr.msu.edu/classes/ece480/capstone/fall13/group04/docs/danapp.pdf

[7] KALRA, Prem K. *Canny Edge Detection*. Indian Institute of Technology Delhi, 2009 [fecha de consulta mayo 2018]. Disponible en: http://www.cse.iitd.ernet.in/~pkalra/col783/canny.pdf

[8] MUÑOZ, P. J. *Segmentación de imágenes*, Universidad de Málaga [fecha de consulta mayo 2018]. Disponible en: http://www.lcc.uma.es/~munozp/documentos/procesamiento_de_imagenes/temas/pi_cap6.pdf (páginas 10-12)

[9] LIANG, J. *Canny Edge Detection*, [fecha de consulta mayo 2018]. Disponible en: http://justin-liang.com/tutorials/canny/

[10] SINHA, U.*The Canny Edge Detector*, [fecha de consulta mayo 2018]. Disponible en: http://aishack.in/tutorials/canny-edge-detector/

[11] LAZEBNIK, S. *Edge detection*. University of Illinois at Urbana-Champaign, Department of Computer Science, 2016 [fecha de consulta mayo 2018]. Disponible en: http://slazebni.cs.illinois.edu/spring16/lec07_edge.pdf

[12] MEER, P. *Canny Edge Detection*, Rutgers, The State University of New Jersey, [fecha de consulta mayo 2018]. Disponible en: http://rci.rutgers.edu/~meer/GRAD561/ADD/cannyedge.pdf

https://github.com/jfrz38/CannyEdgesAlgorithm