ULPGC
Universidad de
Las Palmas de
Gran Canaria

Escuela de
Ingeniería Informática

Implementación de un detector de bordes a nivel subpíxel para la librería OpenCV

Grado en Ingeniería Informática
Julio 2020

Tutor: Agustín Rafael Trujillo Pino
Alumno: Juan Sebastián Ramírez Artiles

SOLICITUD DE DEFENSA DE TRABAJO DE FIN DE TÍTULO

D/Dª _____Juan Sebastián Ramírez Artiles_____, autor del Trabajo de Fin de Título _____Implementación de un detector de bordes a nivel subpíxel para la librería OpenCV_, correspondiente a la titulación ____Grado en Ingeniería Informática____, en colaboración con la empresa/proyecto (indicar en su caso) _____

S O L I C I T A

que se inicie el procedimiento de defensa del mismo, para lo que se adjunta la documentación requerida, haciendo constar que

[X] se autoriza / [ ] no se autoriza

la grabación en audio de la exposición y turno de preguntas.

Asimismo, con respecto al registro de la propiedad intelectual/industrial del TFT, declara que:

[ ] Se ha iniciado o hay intención de iniciarlo (defensa no pública).
[X] No está previsto.

Y para que así conste firma la presente.

Las Palmas de Gran Canaria, a _10_ de _Julio_ de _2020_.

El estudiante

Fdo.: RAMIREZ ARTILES JUAN SEBASTIAN - 54068376E

Firmado digitalmente por RAMIREZ ARTILES JUAN SEBASTIAN - 54068376E
Fecha: 2020.07.10 14:58:54 +01'00'

A rellenar y firmar **obligatoriamente** por el/los tutor/es

En relación a la presente solicitud, se informa:

TRUJILLO PINO AGUSTIN RAFAEL - 42843425Z

Firmado digitalmente por TRUJILLO PINO AGUSTIN RAFAEL - 42843425Z
Fecha: 2020.07.10 12:28:09 +01'00'

[X] Positivamente

[ ] Negativamente
(la justificación en caso de informe negativo deberá incluirse en el TFT05)

DIRECTOR DE LA ESCUELA DE INGENIERÍA INFORMÁTICA

Índice:

1 Introducción:

1.1 Breve descripción del proyecto y objetivos iniciales:

El proyecto que acordamos desarrollar para este TFG consiste en la implementación de un detector de bordes a nivel subpíxel en un lenguaje de alto nivel. Con anterioridad a este trabajo, el método ya se encontraba implementado en Matlab, por lo tanto, la mayor parte del trabajo consistía en adaptar el código de Matlab al lenguaje objetivo en este TFG.

El artículo que marcará la pauta para el desarrollo del trabajo se encuentra accesible a través de la web de sciencedirect.com. Los códigos originales en Matlab se encuentran en la web de MatWorks. El resultado del proyecto se ha subido a un repositorio de mi cuenta de GitHub de modo que sea accesible a toda la comunidad.

Con el objetivo de tratar las imágenes y poder procesarlas a bajo nivel hemos utilizado la librería OpenCV. Esta librería nos ofrece una gran cantidad de recursos para el tratamiento de imágenes, con la ventaja adicional de ser un proyecto de código libre y de extensa difusión en la comunidad dedicada a la visión por computador.

Como lenguaje de alto nivel para la implementación hemos decidido utilizar C++. La elección de este lenguaje se debe a que es el más adecuado para una posible integración futura en la librería OpenCV. Es por este motivo que se ha diseñado un pequeño programa de consola de modo que se pueda probar el funcionamiento de las distintas aproximaciones al método final. Las aproximaciones al método siguen el proceso de perfeccionamiento, descrito en el artículo.

Uno de los objetivos previos definidos en el proyecto era la creación de una DLL de modo que el método pudiera quedar encapsulado como una librería que pudiese ser utilizada en cualquier lenguaje de programación en la plataforma Windows.

La idea era crear el método en una librería y llamarla desde Python o desde un lenguaje de la familia .Net. Lo que se pretendía con esto era aprovechar la eficiencia de C++ y la versatilidad de los otros lenguajes. A la consecución de este objetivo se le dedicó un tiempo que no se ha visto plasmado en el trabajo final.

1.2 Trabajo previo y evolución del proyecto:

Los inicios de nuestro proyecto plantearon muchas dudas. Teníamos varias cosas claras, íbamos a usar OpenCV, y como opciones principales íbamos a usar C++ o Python o incluso una combinación de ambos lenguajes. Para la combinación de varios lenguajes nos habíamos planteado la creación de DLLs. La opción de encapsular el proyecto en una librería dinámica parecía una opción atractiva que podría mejorar la portabilidad del método, no obstante, esta idea quedó aparcada.

Como primer paso, antes de afrontar el trabajo del detector, nos dedicamos a recopilar información bibliográfica sobre C++, Python, y Java con OpenCV. También probamos el ejemplo "Edge Detection in OpenCV 4.0, A 15 Minutes Tutorial" [L7] en varios lenguajes; concretamente en Java y en C++, además de en Python.

Cuando empezamos a desarrollar el detector intentamos primero implantarlo en java y en C++ simultáneamente. La opción de hacer la implementación en Java no era la preferida, no obstante, esta tecnología es la de mayor uso en los contenidos curriculares de la carrera, y por este motivo nuestra primera toma de contacto fue a través de este lenguaje.

Hubo un momento crítico en nuestro proyecto. El problema consistía en conseguir pintar los bordes en el interior del píxel. Finalmente logramos solucionarlo mediante una opción que tiene OpenCV para el uso de coordenadas en fragmentos de píxel. Otro de los problemas que se nos planteó en el proyecto fue descubrir la forma de aplicar el zoom a la imagen sin escalar toda la imagen sino solo la fracción de imagen que debía abarcar en la ventana.

Para dar forma final al proyecto nos planteamos crear una interfaz sencilla de modo que el usuario pudiera probar el método y variar todos sus parámetros (orden de ajuste, umbral, tipo de método, etc.) sin necesidad de tener que instalar nada.

Como herramienta para dar visibilidad al proyecto nos planteamos publicar un repositorio en internet que hiciera posible compartir nuestro trabajo con toda la comunidad.

1.3 Acerca del proyecto:

Como ya hemos comentado, el proyecto consiste en una implementación en C++ del método desarrollado en el artículo "Accurate subpixel edge location based on partial area effect" escrito por A. Trujillo et al [A1]. El resultado final del proyecto consiste en una aplicación de consola precompilada y en el código fuente que lo genera.

La interacción del usuario con la aplicación es a través de opciones de línea de comandos y de control mediante teclado. Para la visualización de imágenes hace uso de la ventana nativa de OpenCV. El código fuente se compiló desde Visual Studio 2019 Community. Para configurar el entorno de desarrollo se ha diseñado un tutorial que guía paso a paso el proceso de integración de OpenCV en Visual Studio.

Los detalles del funcionamiento de la aplicación se pueden consultar en el punto "Manual de uso de la aplicación".

Tanto los ficheros fuente como el binario se pueden descargar del repositorio de GitHub habilitado para este proyecto.

En los siguientes puntos vamos a explicar resumidamente en qué consiste el método y abordaremos los detalles de su implementación en C++, así como los resultados obtenidos en las pruebas realizadas.

1.4 Herramientas utilizadas:

- OpenCV 4.3.0
- Microsoft Visual Studio 2019 Community
- PyCharm Community 2019.3
- NetBeans IDE 8.2
- StarUML 3.2.2
- JSON nlomann library
- GetOpt for Windows

2 Estado del arte:

Los métodos detección de bordes a nivel subpíxel siguen varias líneas de investigación como son los basados en ajuste, los basados en interpolación y los basados en momentos. El método que hemos implementado se puede clasificar como de ajuste. Como veremos más adelante, nuestro método realiza ajustes de primer y segundo orden, es decir, ajusta a rectas y a parábolas.

Los métodos que usan momentos funcionan calculando centros de 'gravedad', en este caso de intensidades de color, de regiones de píxeles. Tabatabai y Mitchell desarrollaron esta técnica por primera vez en su artículo "Edge Location to Subpixel Values in Digital Imagery" de 1984 [A2]. Como ejemplo de este tipo de métodos destacan los que usan momentos ortogonales Zermike o los que usan los más evolucionados momentos ortogonales Fourier-Mellin (OFMM). Como ejemplo de este último método podemos nombrar el de Bin et al. (2008) [A3].

En los métodos de interpolación se reconstruyen los contornos continuos de la imagen en base a los valores discretos de los píxeles. La reconstrucción se puede hacer horizontalmente, verticalmente o combinando los valores de las dos direcciones. Para calcular los valores discretos de los píxeles se emplean los filtros tradicionales Sobel, Canny o LoG (Laplacian of Gaussian). Estos métodos se pueden clasificar, a su vez, en tres:

- Los que reconstruyen la intensidad de color de la imagen. Como ejemplo de este tipo podemos nombrar el método Hagara, Kulla (2011) [A4].
- Los que reconstruyen los contornos de las derivadas primeras de la imagen. Como ejemplo podemos nombrar el método Fabijanska (2014) [A5].
- Los que usan las derivadas segundas para reconstruir los bordes. Este método ha quedado un poco en desuso. Como ejemplo de este método podemos nombrar el de MacVicar-Whelan, Binford (1991) [A6].

Los métodos de ajuste son muy eficientes cuando conocemos la forma de los bordes que queremos detectar. En estos métodos se usa la imagen como un plano en el que los datos de espacio de los bordes se pueden ajustar mediante splines. El ajuste a parábolas es el más común, no obstante, se puede encontrar otros métodos como el de Zhang, Meng y Li (2018) [A7] que ajusta a elipses o el de Qiucheng Sun et al. (2016) [A8] que ajusta a arcotangentes.

Como observación final podemos destacar que, en la literatura que concierne a la detección de bordes a nivel subpíxel, también existen modelos híbridos que usan características de varios de estos métodos. Un ejemplo de este tipo de detectores mixtos es el desarrollado por Qiucheng Sun et al. (2014) [A9], que se basa en una combinación entre momentos-grises y suavizado de splines.

3 Justificación de las competencias específicas cubiertas:

Nuestro proyecto cubre en mayor o menor medida las competencias FB01, CII06, CP01 y TFG01.

La competencia FB01 se refiere a contenidos relacionados con las asignaturas de análisis matemático, álgebra lineal y métodos estadísticos. Como se podrá comprobar a lo largo de esta memoria, el contenido teórico del método implementado hace uso de cálculos integro-diferenciales y algebraicos. También se usan cálculos estadísticos para la evaluación de los errores de ajuste del método.

El análisis y estudio del algoritmo que se implementó en este proyecto se puede considerar como un contenido que cubre la competencia CII06. Esta competencia se relaciona con el diseño de soluciones a problemas concretos. En nuestro caso el problema a resolver consiste en la detección precisa y eficiente de bordes a nivel subpíxel. En el proceso de diseño del proyecto se tuvo que decidir la idoneidad del lenguaje que finalmente se usó para la implantación del algoritmo. Este último punto también puede formar parte de la competencia CII06.

En la fase de análisis inicial de nuestro proyecto planteamos la posibilidad de hacer uso de computación paralela, ya sea de tipo GPU o de CPU. Nuestro proyecto, actualmente no usa estas técnicas, no obstante, se podría plantear como mejora en caso de que tenga continuidad. Estas técnicas se pueden incluir como contenidos relacionados con la competencia CP01.

Finalmente, podemos considerar todo el proceso de planteamiento, desarrollo y puesta en marcha del proyecto, así como la coordinación con el tutor y la presentación final del mismo, como contenidos relacionadas con la competencia TFG01.

4 Aportaciones:

Nuestro proyecto se concibió con la intención de aportar una nueva herramienta a la comunidad de desarrolladores de aplicaciones relacionada con la visión por computador.

También, con el desarrollo de esta implementación se pretende dar mayor visibilidad al método original. Con esta nueva implementación el usuario interesado simplemente tendrá que usar una ventana de comandos para probar el método, sin que tenga la necesidad de instalar ninguna aplicación adicional.

La implementación del método podrá usarse con fines multidisciplinares, pudiendo ser usada para el procesamiento de imágenes médicas, y de modo general, para cualquier proceso que necesite usar imágenes digitales para su correcto funcionamiento.

Todos los recursos utilizados en el proyecto son herramientas libres, por lo que nuestra implementación puede licenciarse como libre sin mayor problema.

5 Fundamentos del método de detección de bordes:

5.1 Introducción al método:

Este método es una mejora importante de los métodos clásicos de detección de bordes utilizados tradicionalmente, como son los filtros Sobel o los Canny. En este método se consigue detectar no solo los bordes a nivel de píxel, sino que se logra hacer una estimación bastante precisa del trazado de los bordes a nivel subpíxel. Este método permite calcular los datos de posición del trazo dentro del píxel, así como su curvatura y su normal.

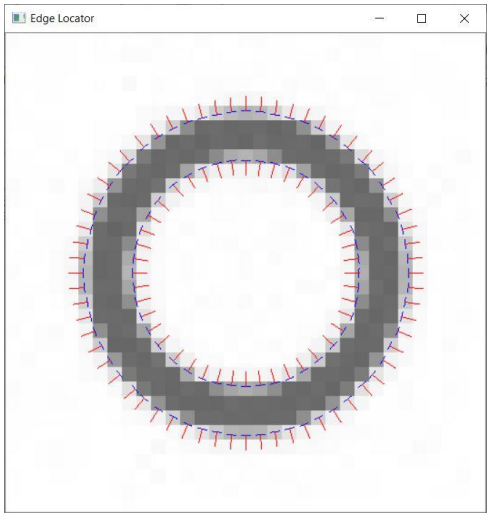


Fig1 *Imagen de Prueba*

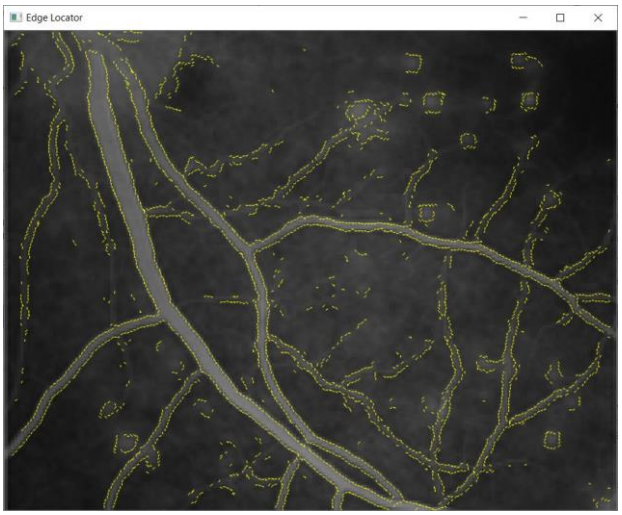


Fig2 *Angio*

En el primer paso del algoritmo se transforma la imagen a escala de grises. Con la imagen ya en gris se pasa a detectar los bordes a nivel píxel, para lo cual se utilizan los filtros Sobel. Con los filtros Sobel obtenemos las derivadas parciales en los ejes X e Y, y con ellas generamos una matriz que almacenará los módulos del gradiente de cada píxel de la imagen. Por último, en este primer paso, se utilizan los módulos del gradiente de cada píxel para detectar todos los píxeles que posean un valor mayor a un cierto umbral y que además sean máximos entre los píxeles de su entorno.

Para detectar los trazos a nivel subpíxel, se ha diseñado un sistema de ventanas que cubre las áreas próximas de cada píxel borde, de modo que, tomando los valores de intensidad de color de los píxeles de la ventana en los extremos más alejados de la porción de borde, podemos realizar los cálculos que finalmente nos permitirán dibujar las líneas continuas a través de cada píxel borde.

5.2 Desarrollo teórico del método:

Podemos dividir la explicación del método en tres niveles de complejidad: una primera aproximación básica en la que se fundamenta la técnica que posteriormente se irá perfeccionando, en la cual se hace uso de ventanas de tamaño fijo; una segunda aproximación en la cual se sigue haciendo uso de ventanas de tamaño fijo pero esta vez se le ha aplicado primero un filtro gaussiano a la imagen; y una tercera aproximación en la que se utilizan ventanas flotantes de tamaño variable para la detección de bordes muy próximos entre sí.

5.2.1 Primera aproximación:

En esta primera aproximación se utilizarán ventanas de tamaño 3x5. Antes de describir cómo funcionaría el método con este tipo de ventana podríamos establecer la fórmula principal de la que derivará por completo el método.

Si suponemos que un píxel resulta atravesado por un borde, podemos estimar la tonalidad final de los lados a partir de la tonalidad del píxel. Si llamamos F al tono de cada píxel, y A y B a los tonos de los lados opuestos por el borde, podemos establecer matemáticamente la relación que existe entre ellos:

$$F(i,j) = B + \frac{A - B}{h^2} P(i,j)$$

Donde la h representa el tamaño del lado del píxel, que en adelante tomará el valor de 1, y P el área que cubre el tono A dentro del píxel. Podemos comprobar fácilmente que cuando P es igual a 1 el tono del píxel tendrá el valor de A, mientras que cuando el valor del área P es igual 0 el tono será B.

Gracias a esta sencilla fórmula podemos construir el método con ventanas estáticas. Como dijimos anteriormente en esta primera aproximación utilizaremos un tamaño de ventana de 3x5. Para ilustrar esta aproximación vamos a suponer que el borde atraviesa la ventana de izquierda a derecha y que su pendiente se encuentra entre 0 y 1. Con esta suposición nos aseguramos de que el borde atraviesa por completo la ventana de lado a lado. Si suponemos, además, que el borde es recto podemos calcular los parámetros de esta recta utilizando la variación de las áreas verticales colindantes de la ventana para plantear un sistema de ecuaciones.

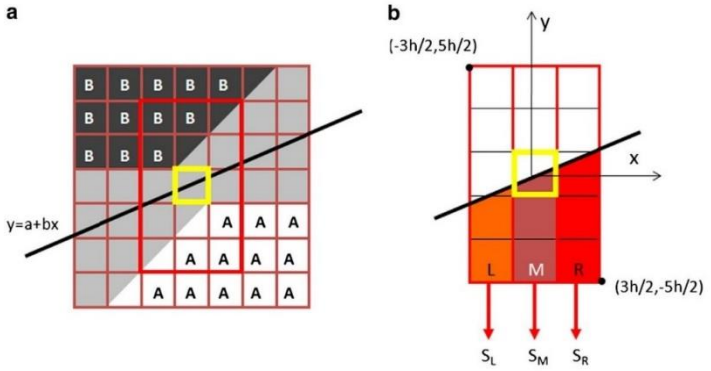


Fig3 Imagen ideal de un borde recto que atraviesa la ventana de lado a lado: a) Píxeles en escala de grises. b) Ventana centrada en el píxel (i, j) usada para calcular los parámetros.

Siguiendo el esquema de la primera fórmula, podemos establecer las fórmulas de las tonalidades acumuladas de cada franja vertical en relación a los dos tonos del borde y al área que abarca la tonalidad A:

$$S_L = \sum_{n=j-2}^{j+2} F_{i-1,n} = 5B + \frac{A-B}{h^2} L \quad (1)$$

$$S_M = \sum_{n=j-2}^{j+2} F_{i,n} = 5B + \frac{A-B}{h^2} M \quad (2)$$

$$S_R = \sum_{n=j-2}^{j+2} F_{i+1,n} = 5B + \frac{A-B}{h^2} R \quad (3)$$

Ahora podemos expresar el área de cada franja como la integral de la recta que atraviesa la franja. Como la recta presenta dos incógnitas (a) y (b) ya podemos plantear un sistema de ecuaciones con las tres anteriores ecuaciones y con las áreas de A expresadas como integrales:

$$L = \int_{-3h/2}^{-h/2} (a + bx + \frac{5}{2}h)\, dx = ah - bh^2 + \frac{5}{2}h^2$$

$$M = \int_{-h/2}^{h/2} (a + bx + \frac{5}{2}h)\, dx = ah + \frac{5}{2}h^2$$

$$R = \int_{h/2}^{3h/2} (a + bx + \frac{5}{2}h)\, dx = ah + bh^2 + \frac{5}{2}h^2$$

Resolviendo el sistema de ecuaciones llegamos a las expresiones de (a) y (b) en función de los tonos A y B y de los valores conocidos de las sumas de los tonos de las tres franjas:

$$a = \frac{2S_M - 5(A+B)}{2(A-B)} h \quad (4) \qquad b = \frac{S_R - S_L}{2(A-B)} \quad (5)$$

Siendo (a) la distancia vertical en la posición central de la ventana.

Para completar esta primera aproximación nos falta determinar qué tonos tomaremos como A y B. Para estimar estos tonos tomamos la media de los tres píxeles de cada esquina opuesta al borde que atraviesa la ventana. Como estamos suponiendo que la pendiente del borde se encuentra entre 0 y 1, deberemos tomar las esquinas superior izquierda e inferior derecha de la ventana, esto es:

$$A = \frac{1}{3}\left(F_{i,j+2} + F_{i+1,j+2} + F_{i+1,j+1}\right) \quad (6)$$

$$B = \frac{1}{3}\left(F_{i-1,j-1} + F_{i-1,j-2} + F_{i,j-2}\right) \quad (7)$$

Con los datos de A, B y de la recta podemos expresar la normal y establecer su signo y su módulo como la diferencia de las tonalidades A y B:

$$N = \frac{A - B}{\sqrt{1 + b^2}}[b, -1]$$

5.2.1.1 *Aproximación cuadrática:*

Como en el caso de la aproximación mediante una recta, la aproximación mediante una parábola sigue la misma mecánica, solo que esta vez hay una variable más. Al ser una parábola la curva que atraviesa la ventana, las integrales de cada franja se calcularán ajustándose a la curva $y = a + bx + cx^2$, y sumando su centro al término independiente, obtenemos:

$$L = \int_{-3h/2}^{-h/2} (a + bx + cx^2 + \frac{5}{2}h)\, dx = ah - bh^2 + \frac{13}{12}ch^3 + \frac{5}{2}h^2$$

$$M = \int_{-h/2}^{h/2} (a + bx + cx^2 + \frac{5}{2}h)\, dx = ah + \frac{1}{12}ch^3 + \frac{5}{2}h^2$$

$$R = \int_{h/2}^{3h/2} (a + bx + cx^2 + \frac{5}{2}h)\, dx = ah + bh^2 + \frac{13}{12}ch^3 + \frac{5}{2}h^2$$

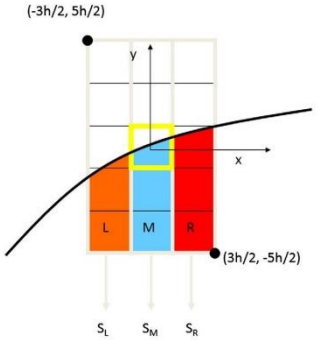


Fig4 Ajuste de segundo orden

Resolviendo el sistema de ecuaciones llegamos al resultado:

$$c = \frac{S_L + S_R - 2S_M}{2(A - B)} \quad (8) \qquad b = \frac{S_R - S_L}{2(A - B)} \quad (9) \qquad a = \frac{2S_M - 5(A + B)}{2(A - B)} - \frac{1}{12}c \quad (10)$$

De lo cual podemos estimar la curvatura en x = 0 como:

$$K = \frac{2c}{(1 + b^2)^{3/2}} \quad (11)$$

5.2.1.2 *Generalización a cualquier valor de pendiente:*

El método tal y como lo hemos explicado hasta ahora se basa en la suposición de que la pendiente de la curva se encuentre entre 0 y 1. Por lo que debemos generalizar el método para que funcione en todas las condiciones posibles. Para generalizar el método podemos distinguir dos situaciones límite. Primero podemos suponer los casos en los que las pendientes del borde estén entre -1 y 1, por lo que la curva resultante se puede detectar usando una ventana vertical de 3x5. El segundo caso, en el que las pendientes de los bordes sean superiores a 1 en valor absoluto, podemos utilizar el mismo método, pero usando esta vez ventanas horizontales de 5x3.

Dentro de estos dos casos, podemos diferenciar, a su vez, dos casos en los que las pendientes se encuentren entre 0 y 1 o entre -1 y 0. La diferencia entre estos dos casos se presenta cuando se intenta calcular los tonos de las esquinas de las ventanas. Para

solucionar esto, simplemente hacemos uso de una variable (m) que sume o reste la unidad de modo que se permute los extremos hacia la derecha o hacia la izquierda.

$$m = \begin{cases} 1 & f_x(i,j)f_y(i,j) > 0 \\ -1 & f_x(i,j)f_y(i,j) \leq 0 \end{cases} \quad (12)$$

Por lo que ahora los cálculos de los tonos límite serán:

$$A = \frac{1}{3}\left(F_{i,j+2} + F_{i-m,j+2} + F_{i-m,j+1}\right) \quad (13)$$

$$B = \frac{1}{3}\left(F_{i+m,j-1} + F_{i+m,j-2} + F_{i,j-2}\right) \quad (14)$$

También podemos distinguir las situaciones en las que la curva se doble hacia valores de tonos más altos o más bajos. Por ejemplo, supongamos el caso de una circunferencia en la que el tono interior es mayor al tono exterior, en este caso la curvatura deberá ser siempre negativa, por lo que se hace necesaria una generalización como la anterior pero esta vez para la curvatura:

$$K = \frac{2cn}{(1+b^2)^{3/2}} \quad (15)$$

Siendo n en los casos horizontales:

$$n = \begin{cases} 1 & f_y(i,j) > 0 \\ -1 & f_y(i,j) \leq 0 \end{cases} \quad (16H)$$

Y en los casos verticales:

$$n = \begin{cases} 1 & f_x(i,j) > 0 \\ -1 & f_x(i,j) \leq 0 \end{cases} \quad (16V)$$

En el caso de que los bordes sean netamente verticales, es decir, en aquellos casos en los que la pendiente sea mayor en valor absoluto a 1, no hará falta ningún cambio en el algoritmo, solo que ahora se ajustará a la expresión $x = a + by + cy^2$, y la distancia horizontal al centro será esta vez en y = 0 igual a (a). El vector normal no variará, y la expresión para los cálculos de las esquinas será:

$$A = \frac{1}{3}\left(F_{i+2,j} + F_{i+2,j-m} + F_{i+1,j-m}\right) \quad (17)$$

$$B = \frac{1}{3}\left(F_{i-1,j+m} + F_{i-2,j+m} + F_{i-2,j}\right) \quad (18)$$

5.2.1.3 Determinación de los píxeles bordes:

Para que este método funcione es necesario determinar qué píxeles se marcarán como píxeles bordes. Como explicamos en la introducción, tomamos como píxeles candidatos a todos los que el módulo de su gradiente supere un cierto umbral. Además, la anterior condición no es suficiente ya que para que el píxel sea considerado como borde debe ser también un píxel con un valor máximo entre los de su entorno.

Para determinar si un píxel tiene un valor máximo en su entorno, debemos considerar si el píxel constituye un borde vertical u horizontal. Si el píxel es vertical su derivada parcial f_x será mayor a f_y, en el caso horizontal será al contrario. Por lo tanto, será condición suficiente que se cumplan las siguientes desigualdades:

En el caso vertical:

$$\begin{cases} |f_x(i,j)| > |f_y(i,j)| \\ |f_x(i-1,j)| \leq |f_x(i,j)| \geq |f_x(i+1,j)| \end{cases} \quad (19)$$

Y en el caso horizontal:

$$\begin{cases} |f_y(i,j)| > |f_x(i,j)| \\ |f_y(i,j-1)| \leq |f_y(i,j)| \geq |f_y(i,j+1)| \end{cases} \quad (20)$$

5.2.2 Segunda aproximación: Imágenes con ruido:

En los casos en que la imagen que se quiera procesar presente ruido - generalmente toda imagen real lo presenta - tenemos la posibilidad de suavizar los bordes aplicándole primero un filtro gaussiano. El método en este caso no necesita ninguna modificación conceptual, no obstante, sí que produce una alteración en la formulación del sistema de ecuaciones y por lo tanto también en el resultado del mismo.

La matriz gaussiana que utilizaremos será de 3x3 y estará conformada del siguiente modo:

$$K = \begin{pmatrix} a_{11} & a_{01} & a_{11} \\ a_{01} & a_{00} & a_{01} \\ a_{11} & a_{01} & a_{11} \end{pmatrix} \quad (21)$$

La matriz, así mismo, debe cumplir que $a_{00} > a_{01} > a_{11}$ y además $a_{00} + 4a_{01} + 4a_{11} = 1$

Como resultado del proceso de suavizado se produce una alteración el área que abarca los píxeles con valores intermedios entre A y B. Este área es ahora mayor, tal como se puede observar en la siguiente imagen:

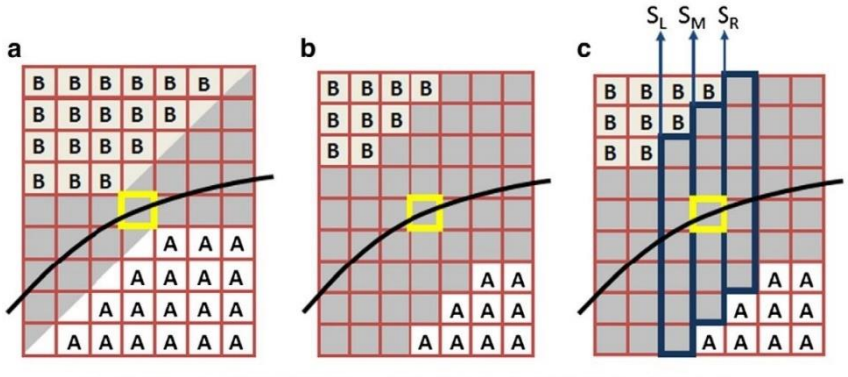


Fig5 Influencia del suavizado: a) Imagen sin suavizado. b) Imagen después del suavizado. c) Nuevo sistema de ventanas.

Si llamamos G a la imagen suavizada, y si suponemos de nuevo el caso ideal en el que la pendiente del borde se encuentra entre -1 y 1, y considerando de nuevo la variable (m) del mismo modo que en caso anterior pero esta vez evaluado sobre las derivadas parciales de G; podemos expresar las sumas de las columnas de la ventana como:

$$S_L = \sum_{k=-3-m}^{3-m} G_{i-1,j+k} \quad (22)$$

$$S_M = \sum_{k=-3}^{3} G_{i,j+k} \quad (23)$$

$$S_R = \sum_{k=-3+m}^{3+m} G_{i+1,j+k} \quad (24)$$

Como se puede observar, ahora el tamaño de las franjas verticales es de 7 y se encuentran desplazadas de modo que se ajusten de un modo óptimo al trazo estimado del borde. Esta nueva configuración de las franjas de las ventanas y los valores modificados de los píxeles de G, da lugar a la solución siguiente del sistema:

$$c = \frac{S_L + S_R - 2S_M}{2(A-B)} \quad (25) \qquad b = m + \frac{S_R - S_L}{2(A-B)} \quad (26) \qquad a = \frac{2S_M - 7(A+B)}{2(A-B)} - \frac{1 + 24a_{01} + 48a_{11}}{12}c \quad (27)$$

Al ser las ventanas distintas también se verá afectada la estimación de los tonos A y B. Los tonos para el caso horizontal son ahora:

$$A = \frac{1}{3}\left(F_{i,j+4} + F_{i-m,j+4} + F_{i-m,j+3}\right) \quad (28)$$

$$B = \frac{1}{3}\left(F_{i+m,j-3} + F_{i+m,j-4} + F_{i,j-4}\right) \quad (29)$$

Para el caso vertical nos quedaría como:

$$A = \frac{1}{3}\left(F_{i+4,j} + F_{i+4,j-m} + F_{i+3,j-m}\right) \quad (30)$$

$$B = \frac{1}{3}\left(F_{i-4,j} + F_{i-4,j+m} + F_{i-3,j+m}\right) \quad (31)$$

5.2.3 Tercera aproximación: Sistema de ventanas flotantes:

En los casos en los que la imagen contenga bordes muy cercanos entre sí, de modo que más de un borde atraviese alguna ventana, se hace necesario adaptar el método, para lo cual se ideó que las ventanas pudieran modificar su tamaño dinámicamente.

Para implementar esta solución se hace uso de tres pares de nuevas variables: (l1, l2), (m1, m2), y (r1, r2). Cada par de variables establecerá los límites de su franja, por lo que las sumas de las franjas se verán alteradas, quedando del siguiente modo:

$$S_L = \sum_{k=l_1}^{l_2} G_{i-1,j+k} \quad (32)$$

$$S_M = \sum_{k=m_1}^{m_2} G_{i,j+k} \quad (33)$$

$$S_R = \sum_{k=r_1}^{r_2} G_{i+1,j+k} \quad (34)$$

Al alterar el cálculo de las sumas acumuladas de los tonos de las franjas, se ve alterado el sistema de ecuaciones, operando una vez más se llega a la nueva solución:

$$c = \frac{S_L + S_R - 2S_M}{2(A-B)} + \frac{A(2m_2 - l_2 - r_2) - B(2m_1 - l_1 - r_1)}{2(A-B)} \quad (35)$$

$$b = \frac{S_R - S_L}{2(A-B)} + \frac{A(l_2 - r_2) - B(l_1 - r_1)}{2(A-B)} \quad (36)$$

$$a = \frac{2S_M - A(1+2m_2) - B(1-2m_1)}{2(A-B)} - \frac{1 + 24a_{01} + 48a_{11}}{12}c \quad (37)$$

Las estimaciones de los tonos A y B se realizarán esta vez tomando un solo valor a cada lado de la franja central. Por lo tanto, la expresión quedaría igual para los casos de pendiente positiva y para los casos de pendiente negativa

$$A = G_{i,j+m_2} \quad (38)$$

$$B = G_{i,j+m_1} \quad (39)$$

Para bordes verticales:

$$\mathrm{A} = \mathrm{G}_{\mathrm{i+m_2,j}} \quad (40)$$

$$\mathrm{B} = \mathrm{G}_{\mathrm{i+m_1,j}} \quad (41)$$

6 Implementación del detector:

6.1 Detalles de implementación:

Las distintas aproximaciones al método se han codificado en clases separadas. Todas ellas heredan su comportamiento común de la clase abstracta *AbstractEdgeLocator*.

La clase que recoge la primera aproximación es *BasicEdgeLocator*. Esta primera implementación es una trascripción del método usando ventanas estáticas de 3x5 y de 5x3 y no hace uso del suavizado de bordes. La segunda aproximación se encuentra implementada en la clase *BasicEdgeLocatorSmoothed*, como sugiere su nombre en esta clase se realiza un suavizado de los bordes antes del cálculo del resto de parámetros. Del mismo modo que en el método anterior utiliza ventanas estáticas, pero de 3x9 o 9x3, debido al efecto de expansión de los bordes descrito anteriormente en la explicación del método. La tercera aproximación se desarrolla en la clase *EdgeLocatorFloatingWindowsSmoothed*. En esta clase ya se hace uso del suavizado de imagen y de las ventanas flotantes. Adicionalmente a estas tres clases se ha desarrollado la clase *EdgeLocatorFloatingWindows* que funciona con ventanas flotantes pero que no utiliza ningún tipo de suavizado previo a los cálculos.

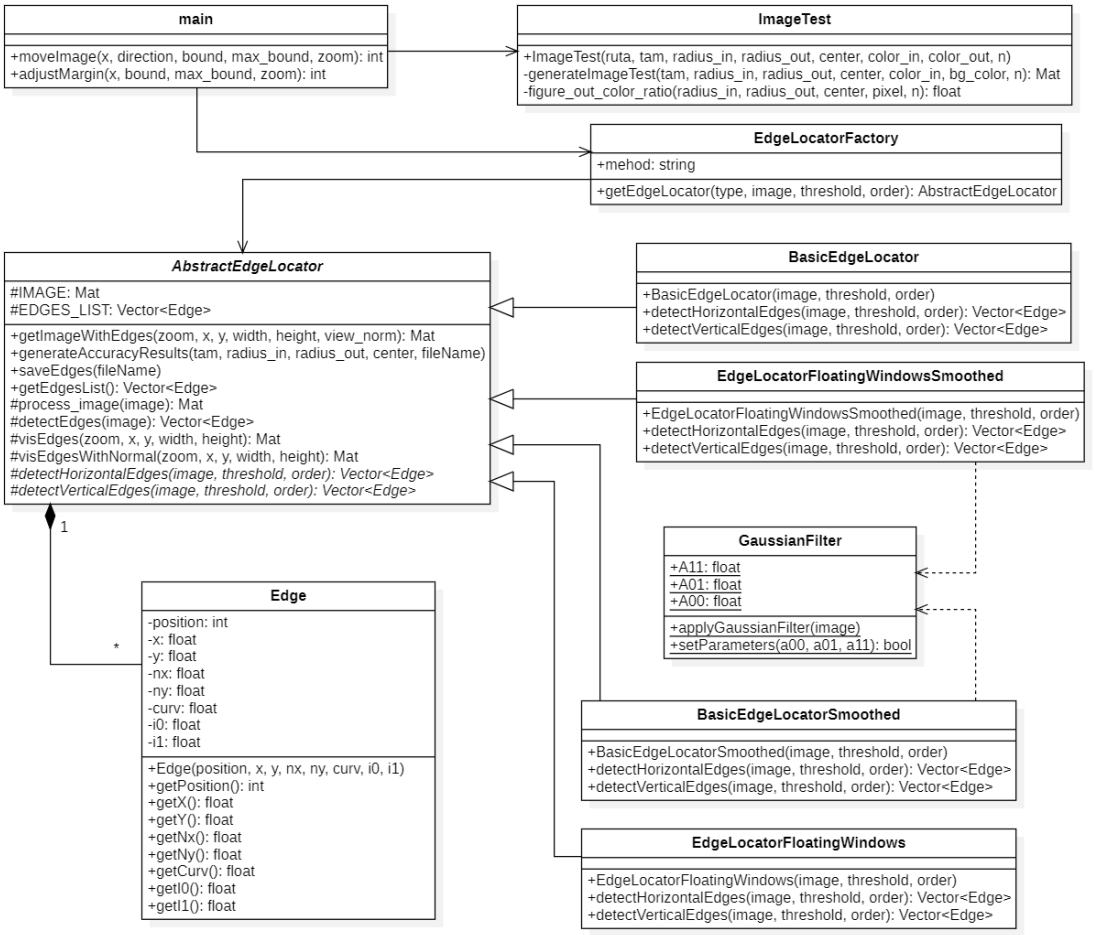


Fig6 Diagrama de clases de la aplicación

El programa, mediante la opción **-m**, permite la selección del método a usar en la detección de bordes. Para lograr esta funcionalidad hemos implementado el patrón *Factory Method* que devuelve la implementación de la clase *AbstractEdgeLocator* seleccionada. La interfaz de la clase *AbstractEdgeLocator* permite a todas sus clases hijas funcionar mediante polimorfismo, lo que aporta flexibilidad al código.

El proceso de detección de los bordes sigue en todos los casos un mismo esquema, esto es: primero se pasa la imagen a escala de grises, el segundo paso sería el suavizado de la imagen, siempre que se haya elegido esa opción. Con la imagen ya monocroma se pasaría a calcular las derivadas parciales y con ellas el gradiente, posteriormente se llama al método de detección de bordes horizontales, seguidamente al método de detección de bordes verticales, y por último se llama al método para visualizar los bordes calculados en el proceso. En los pasos de detección de bordes se irá guardando los resultados en una estructura dinámica *vector* que almacena objetos *Edge*.

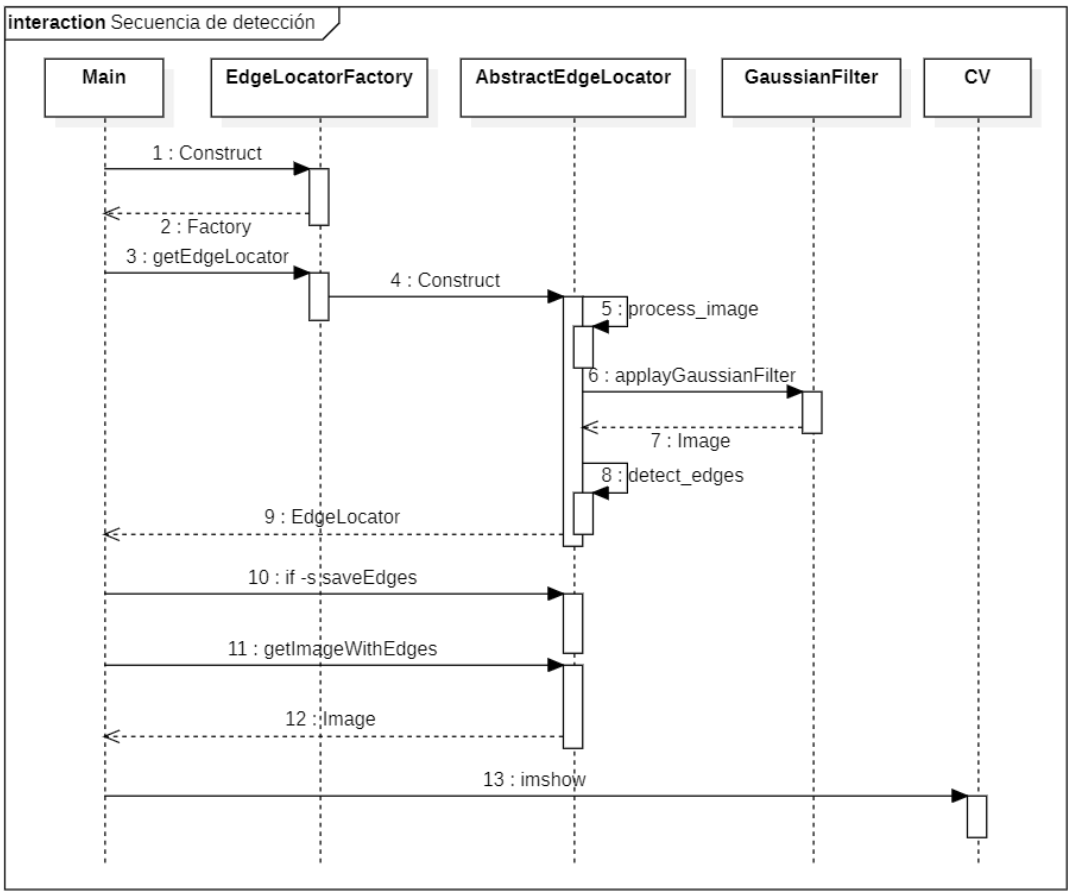


Fig7 Diagrama de secuencia del proceso de detección.

Una vez que se han realizado los cálculos de los bordes y si se ha activado la opción de guardar los resultados generados, se creará un fichero JSON en la carpeta *jsonData*. El fichero tendrá el mismo nombre que el fichero de entrada, pero con una extensión ".json". Los campos de cada registro del fichero JSON serán los mismos que los de los atributos de la clase *Edge*.

Los objetos *Edge* son la unidad mínima de información del resultado de la ejecución. La estructura del objeto *Edge* es la siguiente:

- position: Almacena la posición del píxel borde dentro de la matriz de bytes de la imagen.
- x: Representa la posición horizontal de izquierda a derecha en coordenadas cartesianas del borde.
- y: Representa la posición vertical de arriba a abajo en coordenadas cartesianas del borde.
- nx: Representa la componente x del vector normal al borde normalizado.
- ny: Representa la componente y del vector normal al borde normalizado.
- curv: Almacena la curvatura del borde.
- i0: Almacena la intensidad de color menor del borde.
- i1: Almacena la intensidad de color mayor del borde.

6.1.1 Consideraciones sobre la conversión de código Matlab a C++:

En el proceso de adaptación del código Matlab a código C++ hay que tener en cuenta que los arrays y las matrices en Matlab se almacenan en base 1, mientras que en C++ se almacenan en base 0. Otra consideración importante a tener en cuenta es que las matrices en Matlab se organizan en Column-Major, mientras que en C++ se organizan internamente en Row-Major. Estas diferencias notables entre Matlab y C++ obligan a un esfuerzo minucioso en el manejo de los índices de las matrices.

Otra diferencia entre Matlab y C++ consiste en la forma de tratar las imágenes. Matlab de modo automático realiza los cálculos matriciales paralelizados, mientras que en C++ las imágenes son arrays de bits que deben tratarse mediante dobles bucles for. Estos bucles se procesan de modo secuencial en un coprocesador de la CPU. Para conseguir el mismo efecto que en Matlab deberíamos usar OpenMP, si quisiéramos hacer paralelismo en CPU, o CUDA u otra librería de paralelismo en GPU para paralelizar en la tarjeta gráfica.

6.2 Descripción de las clases implementadas:

6.2.1 La clase AbstractEdgeLocator:

Esta clase será la clase padre de la que heredarán las clases que implementan las distintas aproximaciones al método.

La clase usa tan solo dos atributos: IMAGE que es de tipo cv::Mat, que almacena la imagen de entrada en escala de grises; y EDGES_LIST, de tipo std::vector<Edge>, que almacena todos los píxeles bordes detectados junto con los datos calculados para cada píxel. Estos atributos tienen visibilidad protegida de modo que pueden ser utilizados por las clases hijas, pero impidiendo al mismo tiempo que puedan ser accedidos desde fuera del objeto.

6.2.1.1 getImageWithEdges (público):

El método público que permite acceder a la imagen, o porción de imagen, ya con sus bordes dibujados sobre ella es "*cv::Mat getImageWithEdges(int zoom, int x, int y, int width, int height, bool view_norm=false)*". El método devolverá la imagen que se presentará en la ventana.

Los parámetros x e y determinan la posición del borde superior izquierdo de la imagen una vez aplicado el zoom sobre esta.

Los parámetros width y height establecen el tamaño real de la porción de imagen que se visualizará en la ventana. Si la imagen es menor al tamaño máximo de la ventana width y height tendrán el tamaño de la imagen, y por lo tanto el tamaño de la ventana se reducirá. En el caso de que el tamaño de la imagen sobrepase el máximo del ancho o alto de la ventana los parámetros width o height o los dos se establecerán al tamaño máximo de esta.

El parámetro opcional view_norm permite indicar al método el modo en que se deben dibujar los bordes. Existen dos posibilidades, que se dibujen con líneas amarillas y sin la normal, o que se dibujen los bordes de azul y las normales de cada borde en rojo. Si view_norm es true se llamará al método visEdgesWithNormal y si es false se llamará al método visEdges.

6.2.1.2 getEdgesList (público):

El método "*std::vector<Edge> getEdgesList()*" es el accesor del atributo EDGES_LIST actualmente no tiene uso, pero podría ser de utilidad para usos futuros.

6.2.1.3 saveEdges (público):

El método "*void saveEdges(std::string fileName)*" sirve para guardar los datos contenidos en EDGES_LIST en un fichero JSON. Cada registro del fichero JSON estará conformado por los atributos de los objetos EDGE de cada registro del vector EDGES_LIST.

6.2.1.4 generateAccuracyResults (público):

El método "*void generateAccuracyResults(cv::Size tam, float radius_in, float radius_out, cv::Point2f center, std::string fileName)*" se encarga de generar el fichero con los resultados estadísticos de la ejecución del programa en modo de prueba. Los cálculos que se almacenarán serán:

- El error cuadrático medio de la posición del borde del anillo externo.
- La media de la distancia al centro de los píxeles bordes del anillo externo.
- La desviación estándar de la anterior medición.

- El error cuadrático medio de la posición del borde del anillo interno.
- La media de la distancia al centro de los píxeles bordes del anillo interno.
- La desviación estándar de la anterior medición.

- El error cuadrático medio de la curvatura del anillo externo.
- La media del radio de curvatura el anillo externo.
- La desviación estándar de la medición anterior.

- El error cuadrático medio de la curvatura del anillo interno.
- La media del radio de curvatura el anillo interno.
- La desviación estándar de la medición anterior.

- El error de proyección medio del vector normal de los píxeles borde.

6.2.1.5 process_image (protegido):

El método "*cv::Mat process_image(cv::Mat& image)*" es para uso interno de las clases hijas. Mediante este método se transforma la imagen de entrada en una imagen en escala de grises y con sus píxeles de tipo CV_32F, es decir de float de 32 bits. Esto es necesario ya que utilizaremos los valores de los píxeles para hacer cálculos con decimales. Decidimos utilizar float de 32 bits ya que ofrece mejor rendimiento tanto en uso de memoria como en velocidad de procesamiento que double de 64 bits.

6.2.1.6 detectEdges (protegido):

El método "*std::vector<Edge> detectEdges(cv::Mat& image, float threshold=20, int order=2)*" es de uso interno para los objetos de clases hijas, es decir, para las implementaciones de la clase *AbstractEdgeLocator*. Se encarga de la detección de los bordes y de calcular todos los datos relativos a estos. Devuelve la lista de bordes que se almacenará en el atributo EDGES_LIST.

Dentro de este método se llama a los métodos detectHorizontalEdges y detectVerticalEdges que serán implementados por las clases hijas y que se usarán para hacer polimorfismo. Los valores devueltos por estos dos métodos se concatenarán para conformar el resultado final que se devolverá.

6.2.1.7 visEdges (protegido):

El método "*cv::Mat visEdges(int zoom, int x, int y, int width, int height)*" también es de uso interno en las clases hijas y se encarga de devolver una porción de imagen junto con sus bordes ya dibujados sobre ella. Esta porción de imagen será la que se visualizará por la ventana. Este método dibuja los bordes en amarillo y sin los vectores normales.

6.2.1.8 visEdgesWithNormal (protegido):

El método "*cv::Mat visEdgesWithNormal(int zoom, int x, int y, int width, int height)*" también es de uso interno en las clases hijas y al igual que el anterior se encarga de calcular una porción de la imagen y de dibujar los bordes sobre ella. Esta vez los bordes serán de color azul y se dibujará la normal de cada borde de color rojo.

6.2.1.9 Métodos virtuales (protegidos):

En esta clase se declaran los métodos detectHorizontalEdges y detectVerticalEdges como virtuales puros que confieren a la clase su característica de abstracta. Estos métodos servirán para que las clases hijas implementen las distintas aproximaciones al método descritas en el desarrollo teórico. Estos métodos serán de uso interno en las clases hijas, de ahí que sean protegidos, y permitirán el uso de polimorfismo sobre los objetos de las distintas clases hijas de *AbstractEdgeDetector*.

6.2.2 La clase BasicEdgeLocator:

Esta clase extiende de modo público a la clase *AbstractEdgeLocator* de modo que todos los métodos de la clase padre conservan su visibilidad en *BasicEdgeLocator*.

La primera aproximación del método se implementa en esta clase. En esta primera aproximación se hace uso de ventanas de 3x5 en los casos en los que los bordes detectados sean horizontales, y de 5x3 en los casos de bordes verticales.

6.2.2.1 El constructor (público):

El constructor simplemente necesita dos líneas para funcionar. En la primera se llama al método *process_image* que procesa la imagen para transformarla en monocroma y para cambiar el tipo de dato de los píxeles a float de 32 bits, el resultado de la llamada se almacenará en el atributo IMAGE.

En la segunda línea se llama al método *detectEdges* que devuelve los datos de los bordes en un vector y que almacenarán en el atributo EDGES_LIST. Estos dos métodos no se volverán a llamar en el ciclo de vida del objeto por lo que los objetos de esta clase se pueden considerar inmutables.

En general todas las clases que heredan de *AbstractEdgeLocator* producen objetos inmutables.

6.2.2.2 detectHorizontalEdges (privado):

El método std::vector<Edge> detectHorizontalEdges(cv::Mat &image, float threshold, int order) sobrescribe al método virtual de igual nombre de la clase padre. Hay que destacar que la visibilidad del método se modificó de protegida a privada. Lo hacemos así debido a que ya no es necesario que sea protegido ya que no se espera que se creen clases hijas de esta.

En el listado siguiente se puede observar los pasos del algoritmo de la primera aproximación:

```
std::vector<int> edges;

--Expresión (20)--
for (int i = 2; i < rows - 2; i++) {
   for (int j = 1; j < cols - 1; j++) {
       if (modGradData[i*cols+j] > threshold
       && absFyData[i*cols+j] >= absFxData[i*cols+j]
       && absFyData[i*cols+j] >= absFyData[(i-1)*cols+j]
       && absFyData[i*cols+j] >= absFyData[(i+1)*cols+j]) {
            edges.push_back(i * cols + j);
       }
   }
}

[…]
```

```
int k = 0;

for (int edge : edges) {

   float AA, BB;

   --Expresión (12)--
   int m;
   if (FxData[edge] * FyData[edge] >= 0) {
       m = 1;
   }
   else {
       m = -1;
   }

   --Expresiones (13, 14)--
   AA = (imageData[edge + 2 * cols]
       + imageData[edge + 2 * cols - m]
       + imageData[edge + cols - m]) / 3;

   BB = (imageData[edge - cols + m]
       + imageData[edge - 2 * cols + m]
       + imageData[edge - 2 * cols]) / 3;

   float SL = 0, SM = 0, SR = 0;

   --Expresiones (1, 2, 3)--
   for (int n = 0; n < 5; n++) {
       SL += imageData[edge - 2 * cols + n * cols - 1];
       SM += imageData[edge - 2 * cols + n * cols];
       SR += imageData[edge - 2 * cols + n * cols + 1];
   }

   float den = 2 * (AA - BB);

   --Expresiones (8, 9, 10)--
   if (order == 2) {
       c[k] = (SL + SR - 2 * SM) / den;
   }
   else {
       c[k] = 0;
   }

   b[k] = (SR - SL) / den;
   a[k] = (2 * SM - 5 * (AA + BB)) / den - c[k] / 12;

   A[k] = AA;
   B[k] = BB;

   k++;
}
```

6.2.2.3 detectVerticalEdges (privado):

Este método es simétrico al anterior solo que ahora las columnas son filas y las filas son columnas. Se puede decir que hemos girado el sistema de referencia 90º en sentido contrario a las agujas de reloj.

6.2.3 La clase BasicEdgeLocatorSmoothed:

Esta clase, al igual que la anterior y al resto de clases que implementan las distintas aproximaciones del método, hereda de la clase AbstractEdgeLocator. En esta clase se implementa la segunda aproximación del método, la cual consiste en hacer un suavizado mediante la aplicación de un filtro Gaussiano a la imagen de entrada y posteriormente usar ventanas estáticas para detectar los bordes.

6.2.3.1 El constructor (público):

Este constructor tiene una línea más, en la que se llama al método estático *applayGaussianFilter* de la clase *GaussianFilter* antes de llamar al método *detectEdges*. Al igual que en la anterior clase, los objetos generados por esta son inmutables.

6.2.3.2 detectHorizontalEdges (privado):

Como en el caso de la anterior clase, presentamos un listado en el que detallamos la correlación entre las líneas de código y las expresiones matemáticas del método.

```
int k = 0;

for (int edge : edges) {

      float AA, BB;

      --Expresión (12)--
      int m;
      if (FxData[edge] * FyData[edge] >= 0) {
                  m = 1;
      }
      else {
            m = -1;
      }

      --Expresiones (28, 29)--
      AA = (imageData[edge + 4 * cols]
            + imageData[edge + 4 * cols - m]
            + imageData[edge + 3 * cols - m]) / 3;

      BB = (imageData[edge - 3 * cols + m]
            + imageData[edge - 4 * cols + m]
            + imageData[edge - 4 * cols]) / 3;

      float SL = 0, SM = 0, SR = 0;

      --Expresiones (22, 23, 24)--
      for (int n = 0; n < 7; n++) {
            SL += imageData[edge - (3-m) * cols + n * cols - 1];
            SM += imageData[edge - 3 * cols + n * cols];
            SR += imageData[edge - (3+m) * cols + n * cols + 1];
      }

      --Expresiones (25, 26, 27)--
      float den = 2 * (AA - BB);

      if (order == 2) {
```

```
            c[k] = (SL + SR - 2 * SM) / den;
      }
      else {
            c[k] = 0;
      }

      b[k] = m + (SR - SL) / den;
      a[k] = (2*SM - 7*(AA+BB)) / den - (1+24*GaussianFilter::A01 +
48*GaussianFilter::A11) * c[k] / 12;

      A[k] = AA;
      B[k] = BB;

      k++;
}
```

6.2.3.3 detectVerticalEdges (privado):

Al igual que en la clase anterior la detección de bordes verticales es una simple rotación del eje de coordenadas y, por lo tanto, se pueden aplicar las mismas fórmulas pero intercambiando filas por columnas.

6.2.4 La clase EdgeLocatorFloatingWindows:

Al igual que las dos anteriores clases, esta clase es una implementación de la clase abstracta *AbstractEdgeLocator*. En esta clase el método usa ventanas flotantes, pero sin suavizado previo.

6.2.4.1 El constructor (público):

El constructor está formado por las mismas dos líneas que las del constructor de *BasicEdgeLocator*. Del mismo modo que en las dos clases anteriores, los objetos creados de esta clase son objetos inmutables.

6.2.4.2 detectHorizontalEdges (privado):

Podemos echar un vistazo al listado para ver el funcionamiento de las ventanas flotantes.

```
int l1, l2, r1, r2, minl1, maxl2, minr1, maxr2;
int k = 0;
for (int edge : edges) {

      // Se inicializan los límites dependiendo de si la pendiente
es positiva o negativa
      int m1 = -1;
      int m2 = 1;

      if (FxData[edge] * FyData[edge] >= 0) {
            l1 = 0;
            r2 = 0;
            minl1 = -2;
            maxr2 = 2;
            l2 = 1;
            r1 = -1;
            maxl2 = 4;
            minr1 = -4;
      }
      else {
            l1 = -1;
            r2 = 1;
            minl1 = -4;
            maxr2 = 4;
            l2 = 0;
            r1 = 0;
            maxl2 = 2;
            minr1 = -2;
      }
      // Si la pendiente es completamente horizontal se inicializan
las variables con valores más adecuados.
      if (absFxData[edge] < 1) {
            l1 = -1;
            l2 = 1;
            r1 = -1;
            r2 = 1;
      }
```

```
      // Se recorren los límites para ampliarlos al máximo.
      while (l1 > minl1 && absFyData[edge-1 + l1*cols] >=
absFyData[edge-1 + (l1-1)*cols]) {
            l1--;
      }
      while (l2 < maxl2 && absFyData[edge-1 + l2*cols] >=
absFyData[edge-1 + (l2+1)*cols]) {
            l2++;
      }
      while (m1 > -3 && absFyData[Edge + m1*cols] >= absFyData[edge
+ (m1-1)*cols]) {
            m1--;
      }
      while (m2 < 3 && absFyData[edge + m2*cols] >= absFyData[edge +
(m2+1)*cols]) {
            m2++;
      }
      while (r1 > minr1 && absFyData[edge+1 + r1*cols] >=
absFyData[edge+1 + (r1-1)*cols]) {
            r1--;
      }
      while (r2 < maxr2 && absFyData[edge+1 + r2*cols] >=
absFyData[edge+1 + (r2+1)*cols]) {
            r2++;
      }

      --Expresiones (40, 41)--
      float AA, BB;
      AA = imageData[edge + m2 * cols];
      BB = imageData[edge + m1 * cols];

      --Expresiones (32, 33, 34)--
      float SL = 0, SM = 0, SR = 0;
      for (int n = l1; n <= l2; n++) {
            SL = SL + imageData[edge - 1 + n * cols];
      }

      for (int n = m1; n <= m2; n++) {
            SM = SM + imageData[edge + n * cols];
      }
      for (int n = r1; n <= r2; n++) {
            SR = SR + imageData[edge + 1 + n * cols];
      }
      --Expresiones (35, 36, 37)--
      float den = 2 * (AA - BB);
      if (order == 2) {
            c[k] = (SL + SR - 2 * SM + AA * (2 * m2 - l2 - r2) - BB
* (2 * m1 - l1 - r1)) / den;
      }
      else {
            c[k] = 0;
      }
      b[k] = (SR - SL + AA * (l2 - r2) - BB * (l1 - r1)) / den;
      a[k] = (2*SM - AA*(1 + 2*m2) - BB*(1 - 2*m1))/den - c[k]/12;
      A[k] = AA;
      B[k] = BB;
      k++;
}
```

6.2.4.3 detectVerticalEdges (privado):

Al igual que en las anteriores dos clases la detección de bordes verticales es una simple rotación del caso horizontal, por lo que los cálculos se mantienen invariantes, siempre teniendo en cuenta que hay que invertir filas por columnas.

6.2.5 La clase EdgeLocatorFloatingWindowsSmoothed:

Esta clase es igual que la anterior, solo que en el constructor se llama al método *applyGaussianFilter* que realiza un suavizado de la imagen mediante un filtro gaussiano antes de hacer los cálculos de la detección de bordes.

6.2.6 La clase EdgeLocatorFactory:

Esta clase contiene el método que fabrica los objetos que extienden a la clase *AbstractEdgeLocator*. No contiene constructor y está compuesta únicamente por el "factory method" getEdgeLocator.

6.2.6.1 getEdgeLocator (público):

El método "*AbstractEdgeLocator* getEdgeLocator(int type, cv::Mat &src, float threshold = 20, int order = 2)*" se encarga de generar la instancia concreta de las cuatro clases hijas de *AbstractEdgeLocator*.

El parámetro type se usará para saber qué clase se debe instanciar: el valor 0 crea una instancia de *EdgeLocatorFloatingWindowsSmoothed*; el valor 1 crea una instancia de EdgeLocatorFloatingWindows; el valor 2 crea una instancia de BasicEdgeLocatorSmoothed; y el valor 4 una instancia de la clase BasicEdgeLocator. En caso de que se le pase cualquier otro valor, aborta el programa.

6.2.7 La clase Edge:

Es la clase que se usará para la creación de los objetos que almacenarán la información relativa a cada píxel borde. Sus atributos son de solo lectura y tomarán su valor en el momento de su instanciación, por lo tanto, los objetos de esta clase serán inmutables.

Los atributos de la clase son:

- **position**: posición lineal dentro de la imagen, valor entero.
- **x** e **y**: posición cartesiana del borde dentro de la imagen, valores reales.
- **nx** y **ny**: componentes del vector normal normalizado, valores reales.
- **curv**: valor de la curvatura del borde, valor real.
- **i0** e **i1**: valores de intensidad de los tonos a ambos lados del borde, valor real.

Todos estos atributos tienen sus métodos accesores públicos.

6.2.8 La clase ImageTest:

Esta clase se usará para generar una imagen de test en el modo de prueba. La imagen consistirá en un anillo de tamaño y grosor variable. La imagen se calculará en base a los parámetros pasados en el constructor. El anillo se colocará centrado en el centro de la imagen.

6.2.8.1 El constructor (público):

Para comprender mejor el constructor presentamos un listado del mismo.

```
ImageTest(std::string ruta, cv::Size tam, float radius_in, float radius_out, cv::Point2f center, cv::Vec3f color_in, cv::Vec3f color_out, int n) {

      cv::Mat image = generateImageTest(tam, radius_in, radius_out, center, color_in, color_out, n);

      cv::imwrite(ruta, image);
}
```

El cuerpo del constructor solo tiene dos líneas. En la primera se genera una imagen que se usará en el modo de prueba, y en la segunda se escribe la imagen en el disco.

El parámetro tam contiene las medidas de ancho y alto de la imagen que se va a generar. El parámetro radius_in contiene el valor en número de píxeles del radio interno del anillo. El parámetro radius_out contiene el valor en número de píxeles del radio externo del anillo. Color_in y color_out almacenarán el color del anillo y del fondo. El parámetro n se utilizará para calcular los tonos de los píxeles conforme a una cuadrícula de nxn.

6.2.8.2 El método generateImageTest (privado):

El método "*cv::Mat generateImageTest(cv::Size tam, float radius_in, float radius_out, cv::Point2f center, cv::Vec3f color_in, cv::Vec3f bg_color, int n)*" se encargará de la generación de la imagen. Este método inicializa una imagen al valor de color del fondo, luego recorre cada píxel de la nueva imagen y calcula si los cuatro vértices del píxel caen dentro o fuera del anillo. Si caen dentro pinta el píxel del color color_in y si caen fuera no hace nada. En el caso de los vértices del píxel caigan dentro y fuera se deberá calcular el tono final de este, para esto se llamará al método "*figure_out_color_ratio(float radius_in, float radius_out, cv::Point2f center, cv::Point2f pixel, int n)*" que devolverá el porcentaje de color interno que se debe aplicar al tono final que se asignará al píxel.

6.2.8.3 El método figure_out_color_ratio (público):

El método "*float figure_out_color_ratio(float radius_in, float radius_out, cv::Point2f center, cv::Point2f pixel, int n)*" calcula el porcentaje de color interno que corresponde a un pixel que se encuentre en la frontera. Para realizar el cálculo se recorre una rejilla de nxn posiciones internas del píxel. Si la posición concreta que se esté evaluando cae

dentro del anillo, se sumará a un contador que finalmente se dividirá por el total de muestras de la rejilla, y ese será el porcentaje de color interno del píxel.

Para el cálculo del tono final, simplemente hay que hacer una suma ponderada del tono externo y el interno, es decir:

*color = color_in * ratio + bg_color * (1. - ratio);*

6.2.9 La clase GaussianFilter:

Esta clase se usará para suavizar las imágenes. No tiene constructor y solo posee dos métodos: applyGaussianFilter y setParameters. La matriz tendrá la forma de la expresión (21), y deberá cumplir los criterios $a_{00} > a_{01} > a_{11}$ y además $a_{00} + 4a_{01} + 4a_{11} = 1$

6.2.9.1 El método applyGaussianFilter (público):

El método "static cv::Mat applyGaussianFilter(cv::Mat& image)" será el encargado de suavizar los bordes de la imagen. Usará una matriz gaussiana de 3x3 e irá recorriendo cada píxel de la imagen, menos los márgenes de la imagen a los que no se aplicará filtro, e irá calculando el nuevo valor del píxel según la convolución del filtro en la imagen.

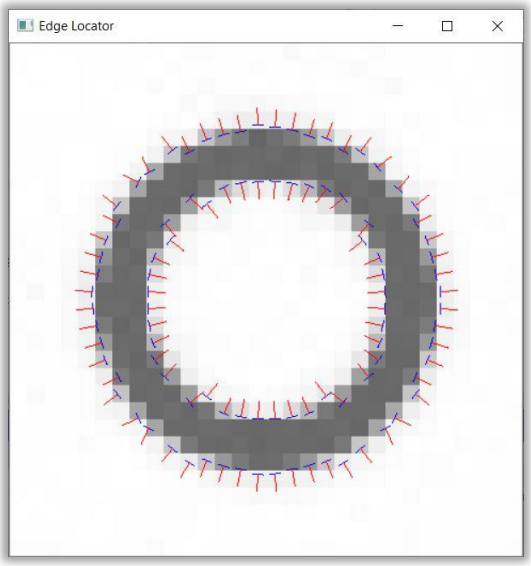
6.2.9.2 El método setParameters (público):

Se puede usar para inicializar la matriz con valores que concuerden con las restricciones de una matriz gaussiana.

7 Resultados:

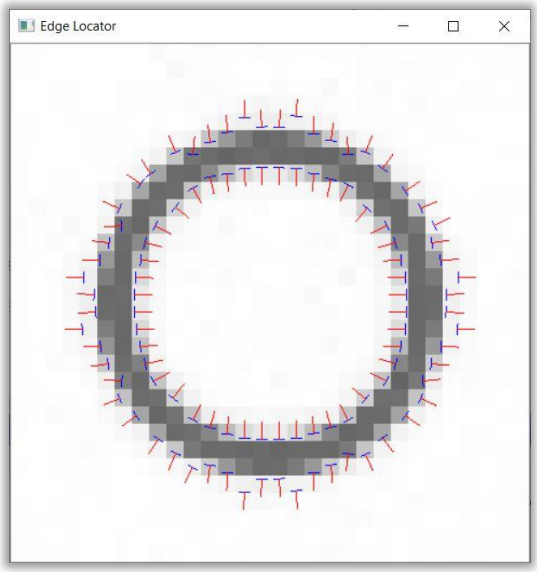
7.1 Método de ventanas estáticas sin suavizado:

7.1.1 Anillo de 7 píxeles de radio interno y 10 de radio externo:



Circunferencia externa	
Radio real	10
MSE posición	0.0399097
Radio medio	10.1008
STD radio	0.174033
MSE curvatura	0.00547511
Radio curvatura medio	9.66278
STD curvatura	0.0745805
Circunferencia interna	
Radio real	7
MSE posición	0.081445
Radio medio	7.09409
STD radio	0.272861
MSE curvatura	0.0792468
Radio curvatura medio	7.54188
STD curvatura	0.0588269
Vector normal	
Error de proyección	0.00760822

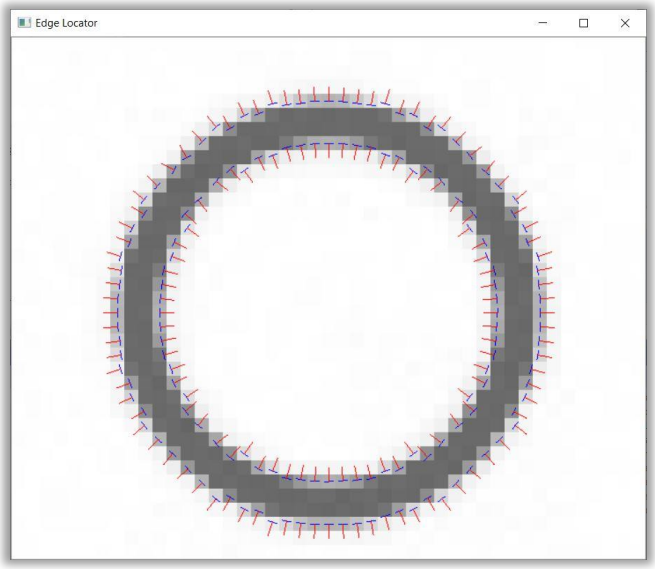
7.1.2 Anillo de 8 píxeles de radio interno y 10 de radio externo:



Circunferencia externa	
Radio real	10
MSE posición	0.173123
Radio medio	10.2461
STD radio	0.338513
MSE curvatura	0.096154
Radio curvatura medio	3.84037
STD curvatura	0.267785
Circunferencia interna	
Radio real	8
MSE posición	0.00910267
Radio medio	7.96808
STD radio	0.0909505
MSE curvatura	0.0434103
Radio curvatura medio	37.1308
STD curvatura	0.144221
Vector normal	
Error de proyección	0.0220626

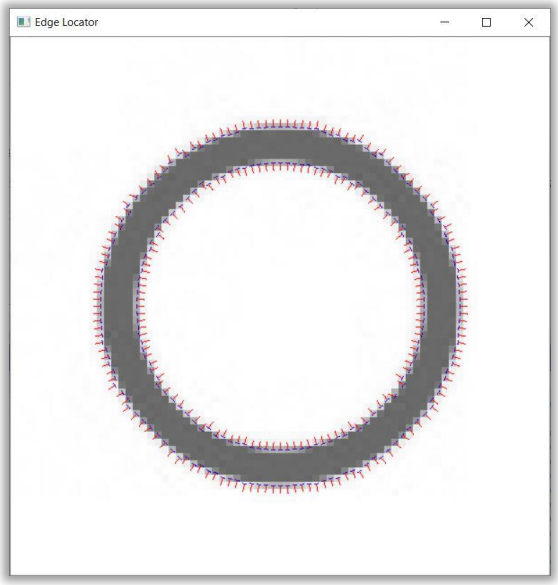
Se puede observar que al reducir el ancho del anillo, este método produce alteraciones muy grandes en el cálculo del radio de curvatura.

7.1.3 Anillo de 12 píxeles de radio interno y 15 de radio externo:



Circunferencia externa	
Radio real	15
MSE posición	0.0327376
Radio medio	15.0777
STD radio	0.164408
MSE curvatura	0.00348526
Radio curvatura medio	13.8585
STD curvatura	0.0591332
Circunferencia interna	
Radio real	12
MSE posición	0.0416652
Radio medio	12.0554
STD radio	0.197919
MSE curvatura	0.0259142
Radio curvatura medio	14.3531
STD curvatura	0.0504106
Vector normal	
Error de proyección	0.00395624

7.1.4 Anillo de 20 píxeles de radio interno y 25 de radio externo:



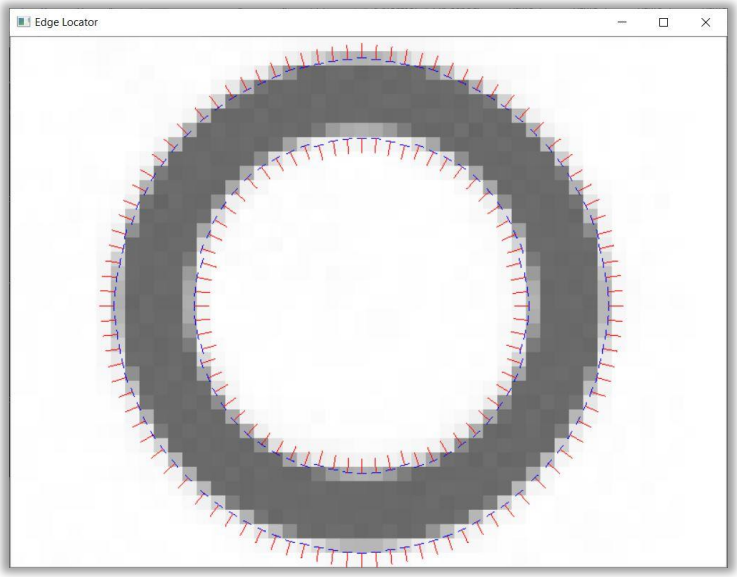
Circunferencia externa	
Radio real	25
MSE posición	0.0357384
Radio medio	25.0188
STD radio	0.188781
MSE curvatura	0.00269449
Radio curvatura medio	24.2429
STD curvatura	0.0520798
Circunferencia interna	
Radio real	20
MSE posición	0.0257277
Radio medio	20.01
STD radio	0.16112
MSE curvatura	0.0109608
Radio curvatura medio	20.8422
STD curvatura	0.037054
Vector normal	
Error de proyección	0.00394728

En este caso el método funciona muy bien con grosor de 5 píxeles en el anillo.

7.2 Método de ventanas estáticas con suavizado:

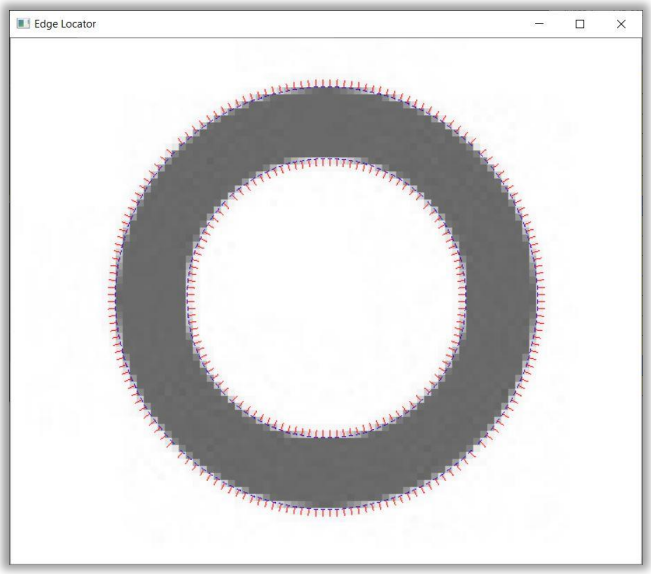
Con este método no se detectan bien los anillos con menos de 4 píxeles de ancho, por este motivo no mostramos resultados de medidas 7x10, 8x10, y 12x15, en su lugar mostramos los resultados de un anillo de 12x17.

7.2.1 Anillo de 12 píxeles de radio interno y 17 píxeles de radio externo:



Circunferencia externa	
Radio real	17
MSE posición	0.0207971
Radio medio	17.1311
STD radio	0.0604452
MSE curvatura	0.00306095
Radio curvatura medio	10.688
STD curvatura	0.0432859
Circunferencia interna	
Radio real	12
MSE posición	0.0679868
Radio medio	11.7899
STD radio	0.155611
MSE curvatura	0.0139167
Radio curvatura medio	56.4102
STD curvatura	0.061308
Vector normal	
Error de proyección	0.000161742

7.2.2 Anillo de 20 píxeles de radio interno y 30 de radio externo:

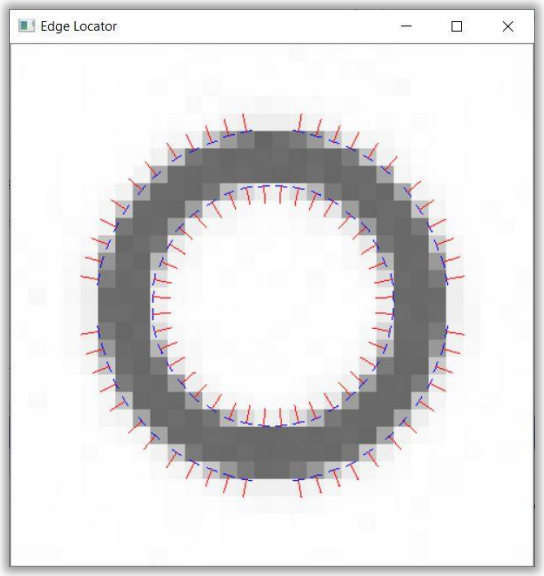


Circunferencia externa	
Radio real	30
MSE posición	0.00759904
Radio medio	30.0804
STD radio	0.033755
MSE curvatura	0.000403978
Radio curvatura medio	27.7828
STD curvatura	0.0199819
Circunferencia interna	
Radio real	20
MSE posición	0.00753508
Radio medio	19.9181
STD radio	0.0289241
MSE curvatura	0.00972809
Radio curvatura medio	21.2376
STD curvatura	0.0174655
Vector normal	
Error de proyección	6.94986e-05

Los datos de radio de curvatura para el caso de grosor de 5 píxeles presentan bastante desviación, no obstante, para un grosor de 10 píxeles el método funciona bien.

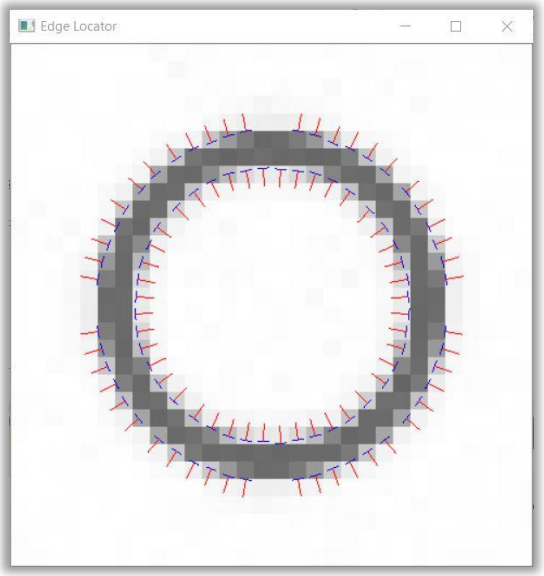
7.3 Método de ventanas flotantes sin suavizado:

7.3.1 Anillo de 7 píxeles de radio interno y 10 de radio externo:



Circunferencia externa	
Radio real	10
MSE posición	0.00728068
Radio medio	10.0762
STD radio	0.03889
MSE curvatura	0.00263702
Radio curvatura medio	10.7222
STD curvatura	0.0514469
Circunferencia interna	
Radio real	7
MSE posición	0.00819901
Radio medio	6.91743
STD radio	0.0376346
MSE curvatura	0.0850348
Radio curvatura medio	6.85464
STD curvatura	0.0412862
Vector normal	
Error de proyección	0.000246857

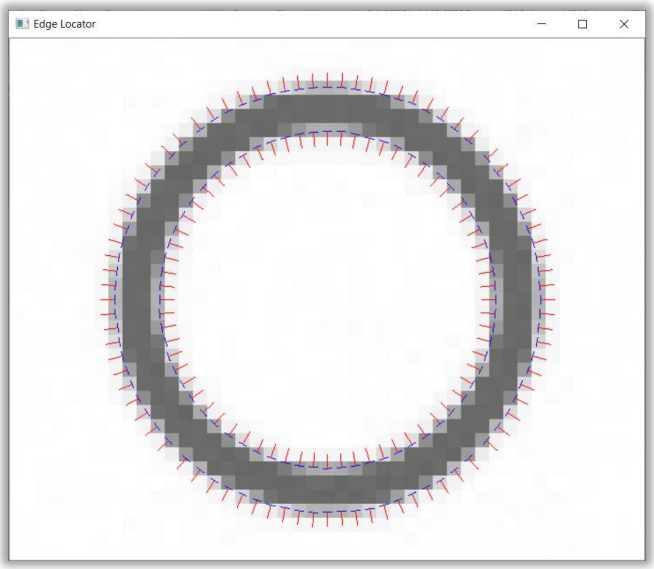
7.3.2 Anillo de 8 píxeles de radio interno y 10 de radio externo:



Circunferencia externa	
Radio real	10
MSE posición	0.00639671
Radio medio	10.0692
STD radio	0.0406004
MSE curvatura	0.00352013
Radio curvatura medio	10.6386
STD curvatura	0.0596509
Circunferencia interna	
Radio real	8
MSE posición	0.00967477
Radio medio	7.91095
STD radio	0.042249
MSE curvatura	0.0659969
Radio curvatura medio	7.99075
STD curvatura	0.0591966
Vector normal	
Error de proyección	0.000285272

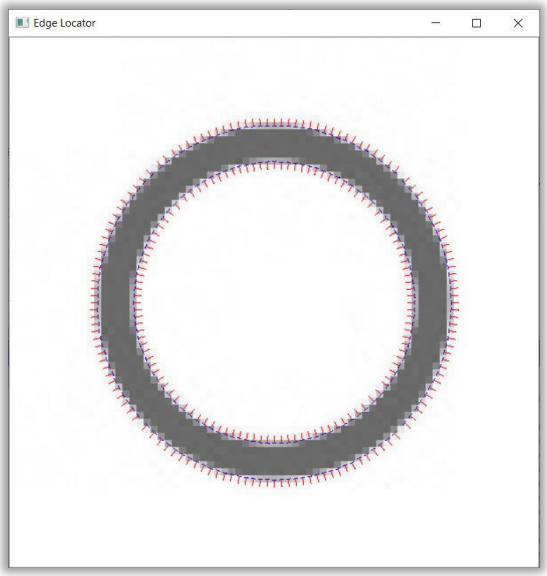
Con este método ya se consiguen valores casi óptimos, siempre en situaciones ideales. Para casos reales se hace necesario aplicar primero un filtro gaussiano.

7.3.3 Anillo de 12 píxeles de radio interno y 15 de radio externo:



Circunferencia externa	
Radio real	15
MSE posición	0.0062512
Radio medio	15.0708
STD radio	0.0354421
MSE curvatura	0.00207855
Radio curvatura medio	14.8464
STD curvatura	0.0458597
Circunferencia interna	
Radio real	12
MSE posición	0.00905097
Radio medio	11.9097
STD radio	0.0302156
MSE curvatura	0.0298741
Radio curvatura medio	11.8405
STD curvatura	0.0417919
Vector normal	
Error de proyección	0.000125987

7.3.4 Anillo de 20 píxeles de radio interno y 25 de radio externo:

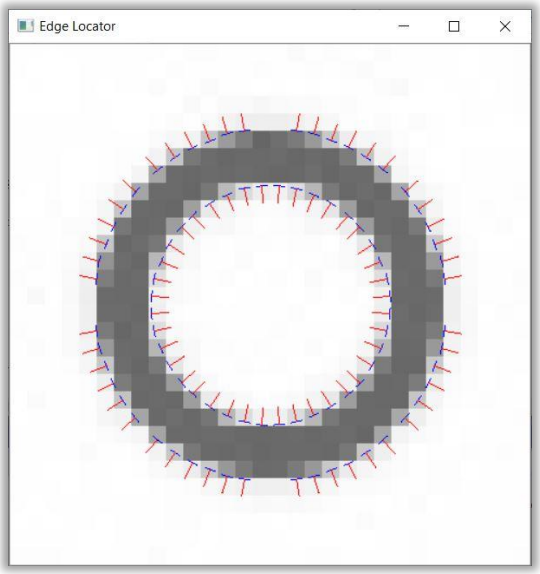


Circunferencia externa	
Radio real	25
MSE posición	0.00921803
Radio medio	25.0853
STD radio	0.0442325
MSE curvatura	0.00269979
Radio curvatura medio	23.7028
STD curvatura	0.0520997
Circunferencia interna	
Radio real	20
MSE posición	0.00890191
Radio medio	19.9132
STD radio	0.0371768
MSE curvatura	0.0115974
Radio curvatura medio	19.7334
STD curvatura	0.0384058
Vector normal	
Error de proyección	0.000150618

En los casos de grosores mayores a 4 píxeles no se ven grandes diferencias con los métodos básicos.

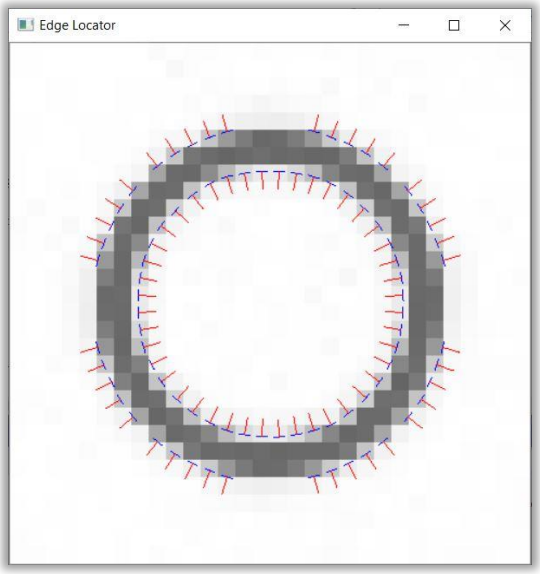
7.4 Método de ventanas flotantes y con suavizado:

7.4.1 Anillo de 7 píxeles de radio interno y 10 de radio externo:



Circunferencia externa	
Radio real	10
MSE posición	0.01919
Radio medio	10.1359
STD radio	0.0269897
MSE curvatura	0.000883358
Radio curvatura medio	10.6481
STD curvatura	0.0293994
Circunferencia interna	
Radio real	7
MSE posición	0.0249351
Radio medio	6.84659
STD radio	0.0378804
MSE curvatura	0.0849224
Radio curvatura medio	6.81886
STD curvatura	0.0336921
Vector normal	
Error de proyección	0.000348532

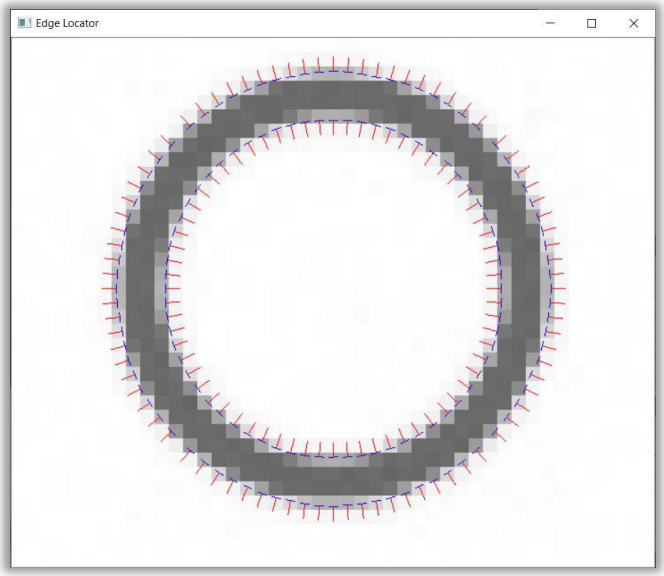
7.4.2 Anillo de 8 píxeles de radio interno y 10 de radio externo:



Circunferencia externa	
Radio real	10
MSE posición	0.0588459
Radio medio	10.2415
STD radio	0.0227216
MSE curvatura	0.00125319
Radio curvatura medio	11.0192
STD curvatura	0.0346062
Circunferencia interna	
Radio real	8
MSE posición	0.0927028
Radio medio	7.69784
STD radio	0.0378853
MSE curvatura	0.0659066
Radio curvatura medio	7.7158
STD curvatura	0.0332948
Vector normal	
Error de proyección	0.000269687

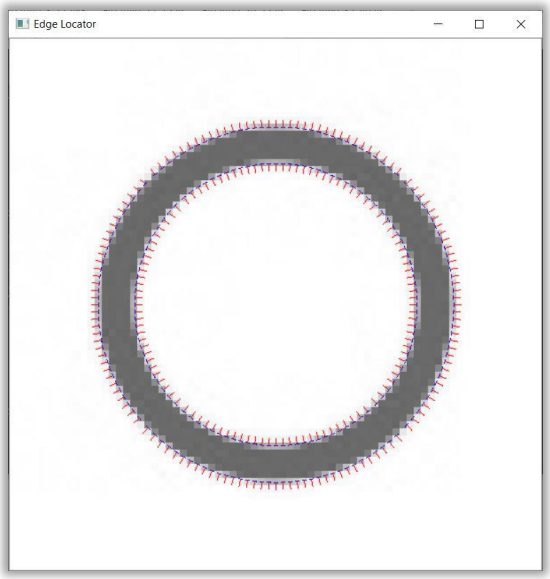
Al aplicar el filtro gaussiano, y siempre teniendo en cuenta la situación ideal de la imagen, se obtienen resultados peores que los de ventanas flotantes sin suavizado.

7.4.3 Anillo de 12 píxeles de radio interno y 15 de radio externo:



Circunferencia externa	
Radio real	15
MSE posición	0.0217195
Radio medio	15.1432
STD radio	0.0352458
MSE curvatura	0.000971711
Radio curvatura medio	15.2678
STD curvatura	0.0313374
Circunferencia interna	
Radio real	12
MSE posición	0.0339506
Radio medio	11.8181
STD radio	0.0293305
MSE curvatura	0.0294286
Radio curvatura medio	11.7592
STD curvatura	0.0330955
Vector normal	
Error de proyección	0.00015377

7.4.4 Anillo de 20 píxeles de radio interno y 25 de radio externo:



Circunferencia externa	
Radio real	25
MSE posición	0.0066097
Radio medio	25.0763
STD radio	0.0280858
MSE curvatura	0.000528655
Radio curvatura medio	24.7076
STD curvatura	0.0230702
Circunferencia interna	
Radio real	20
MSE posición	0.0089969
Radio medio	19.9079
STD radio	0.0227462
MSE curvatura	0.0105185
Radio curvatura medio	19.659
STD curvatura	0.0186375
Vector normal	
Error de proyección	7.55366e-05

Al igual que en el caso anterior al incrementar el grosor del anillo, este método no ofrece grandes mejoras a los métodos básicos. No obstante, en situaciones de imágenes con ruido, los métodos básicos no logran la efectividad de este método.

7.5 Capturas de pantalla de imágenes reales:

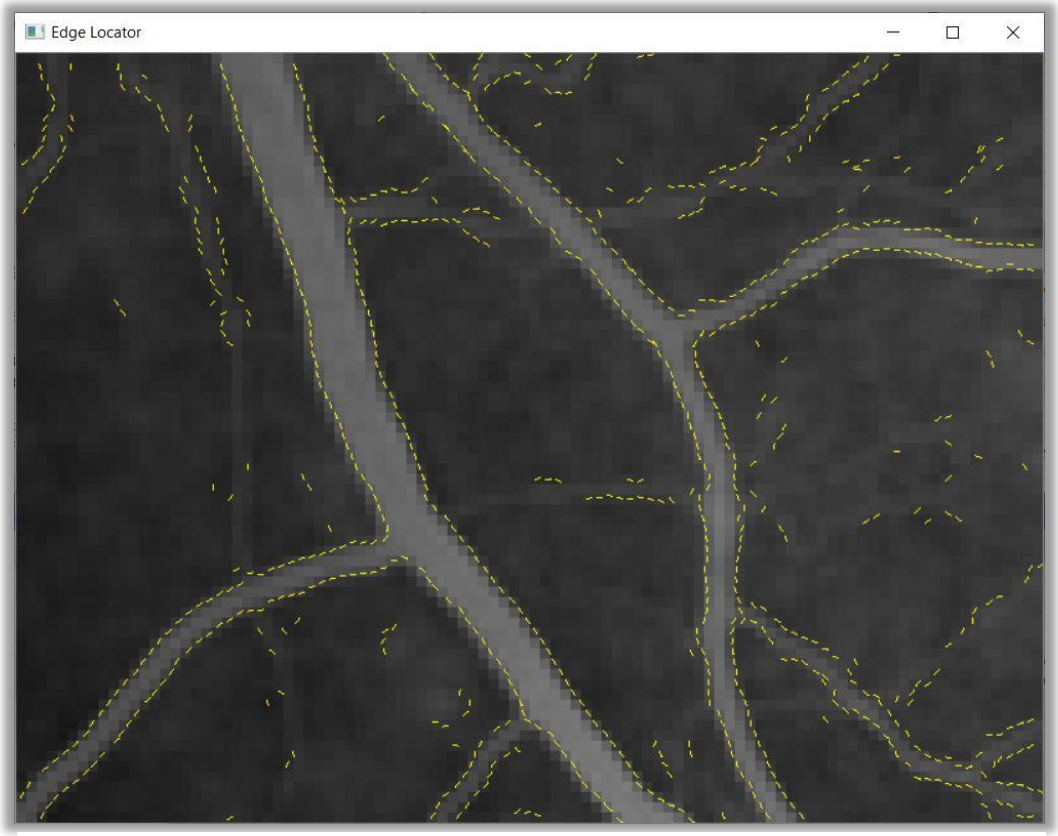


Fig8 Imagen angio

Fig9 Imagen angio

Fig10 Imagen reloj

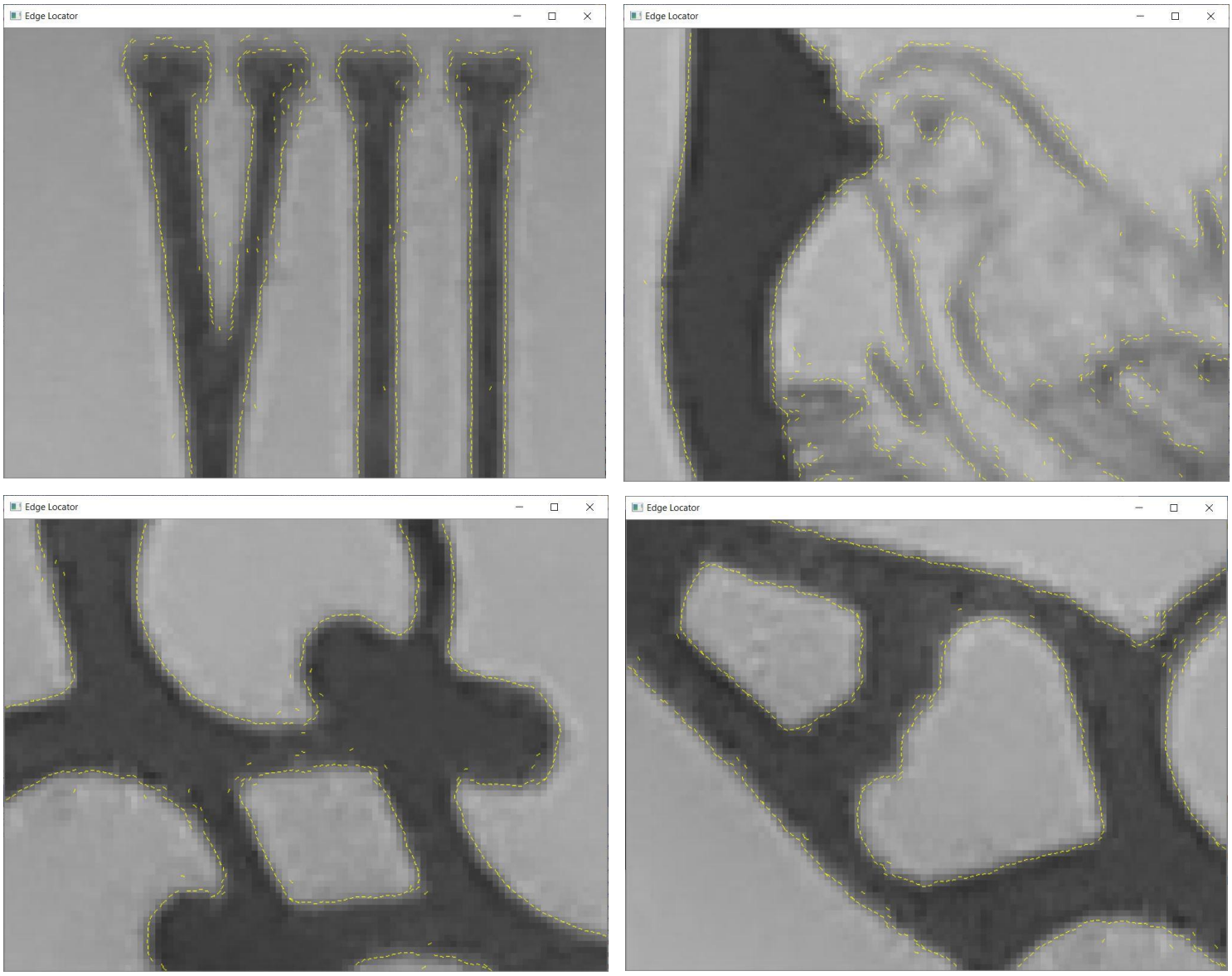


Fig11 Imágenes reloj

Se puede observar, que tanto en las imágenes de la angio como las del reloj, la detección de bordes funciona a un nivel de precisión bastante alto. El algoritmo que se usó para estas pruebas es el de ventanas flotantes con suavizado. Hay que destacar que la versión original del algoritmo es capaz de mejorar estos resultados mediante el uso de varias iteraciones de suavizado antes de la detección.

8 Manual de uso de la aplicación:

Para probar la aplicación puede descargar los ficheros fuente y compilarlos o usar el fichero binario contenido en la carpeta comprimida *EdgeLocator_1.1.zip*. Al descomprimirlo verá que existe una estructura de carpetas. En la carpeta *images* deberá situar los ficheros de imagen que desee procesar. Las carpetas *jsonData* y *accuracyResults* sirven para guardar resultados de la ejecución. En la carpeta *jsonData* se guardarán en formato JSON los parámetros de los bordes calculados en la ejecución, siempre y cuando se haya activado la opción **-s** en el comando. La carpeta *accuracyResults* se usará para guardar las estadísticas de la ejecución en modo test.

El comando admite una serie de parámetros que alteran su funcionamiento. El formato del comando es:

EdgeLocator.exe -f (fileName | "test" [-e (10-60) -i (8-58)]) [-o (1-2)] [-t (10-255)] [-m (0-3)] [-n] [-s]

Un ejemplo podría ser:

EdgeLocator.exe -f angio2.png -t 20

Otro ejemplo para la imagen de test sería:

EdgeLocator.exe -f test -e 11 -i 8 -t 10 -n

Donde la opción **-f** puede ser el nombre del fichero de entrada que debe existir en la carpeta *images*, o a la palabra "test" que activa la imagen de prueba y que admite las opciones **-e** para fijar el radio exterior del anillo, y **-i** para fijar el radio interior del anillo. Las opciones **-e** y **-i** no tienen ningún efecto cuando se procesa una imagen real. La opción **-t** corresponde al umbral del gradiente. La opción **-o** al orden de ajuste: 1 para ajustar a rectas, y 2 para ajustar a parábolas. La opción **-m** permite seleccionar la versión del método a usar: 0 ventanas flotantes con suavizado; 1 ventanas flotantes sin suavizado; 2 ventanas estáticas con suavizado; y 3 ventanas estáticas sin suavizado. La opción **-n** activa la visualización de los vectores normales. Por último, la opción **-s** permite volcar todos los parámetros de bordes calculados en el método a un fichero JSON. El fichero se almacenará en la carpeta *jsonData* y su nombre será el mismo que el de la imagen de entrada, pero su extensión será ahora ".json".

Los valores por defecto para los parámetros opcionales son:

- m: 0
- o: 2
- t: 20
- i: 20
- e: 25

El programa en ejecución permite la interacción con una serie de controles de teclado. Las teclas de dirección sirven para moverse por la imagen, mientras que la tecla (u) aumenta el zoom, la tecla (d) reduce el zoom, y la tecla (q) aborta el programa.

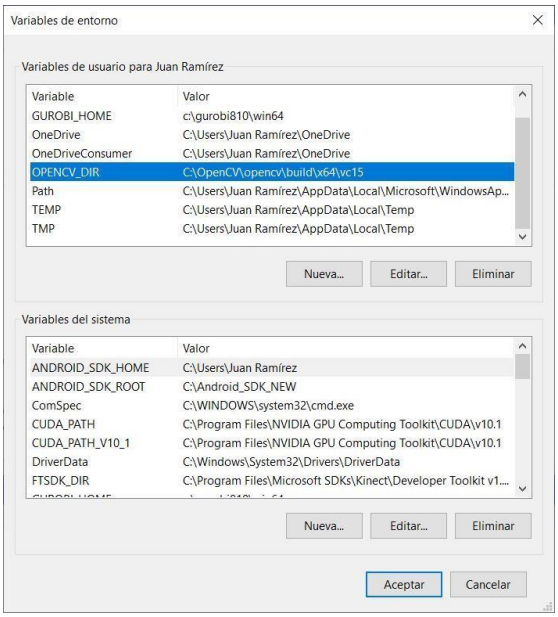
9 Instalación del proyecto en Visual Studio:

Para la implementación del método hemos hecho uso de la librería OpenCV en la versión 4.3.0 x64 vc15, y del software Microsoft Visual C++ 2019 en su versión Community. Para instalar el proyecto siga los siguientes pasos:

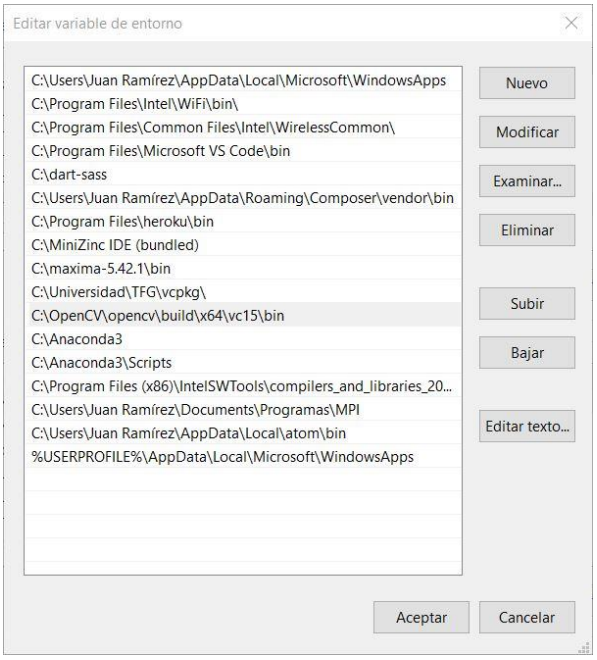
1. Descargue OpenCV de https://opencv.org/opencv-4-3-0/.

2. Descomprímalo en una ruta que preferiblemente no contenga espacios en blanco o caracteres extraños.

3. Cree la variable de entorno OPENCV_DIR:

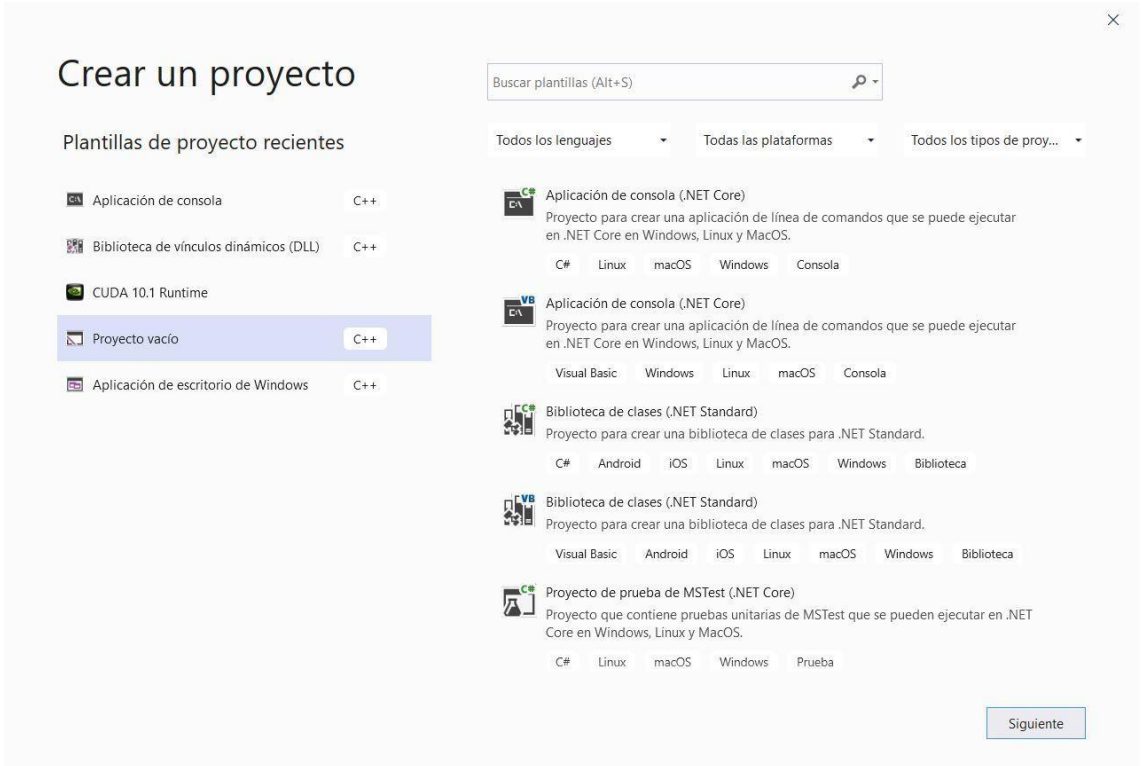


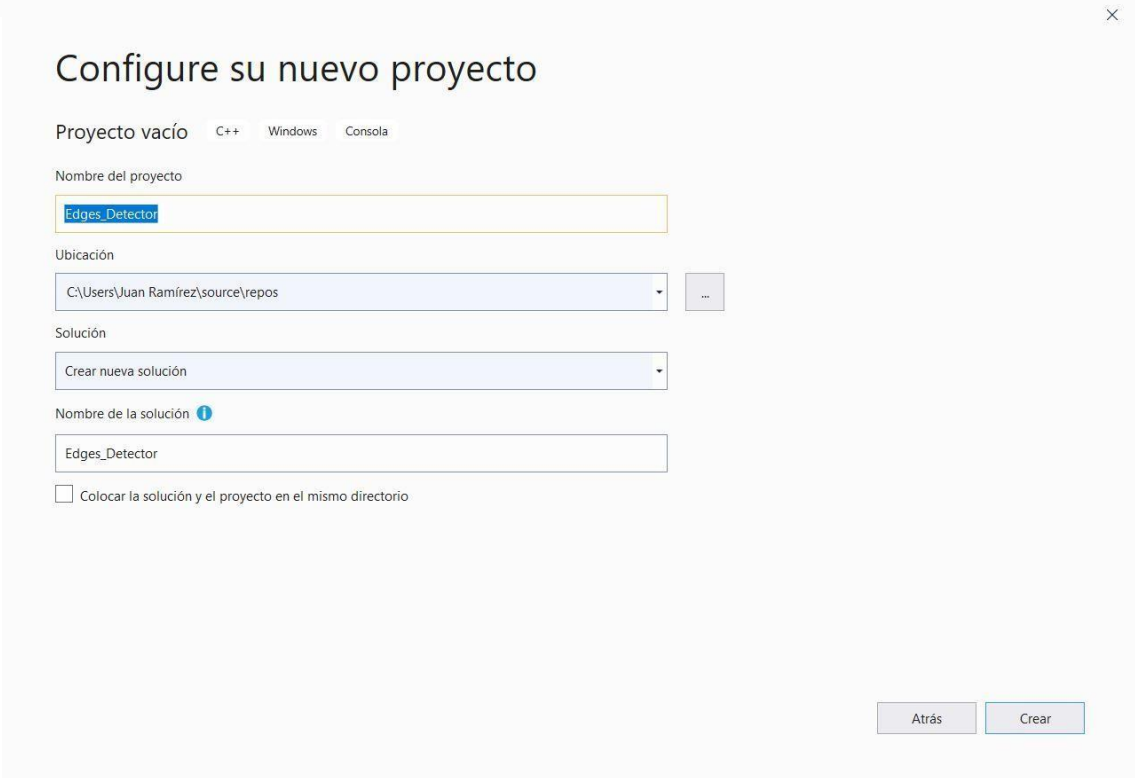
4. Añada la ruta de los binarios de OpenCV a la variable Path.



5. Instale Visual Studio:

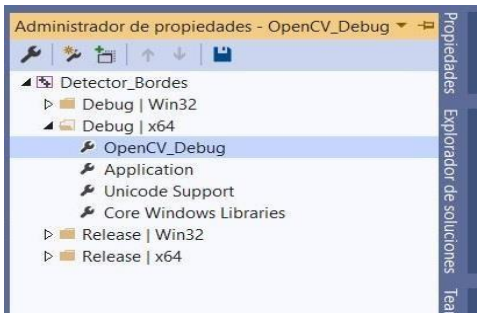
6. Cree un proyecto nuevo vacío, preferiblemente no coloque la solución en el mismo directorio del proyecto.

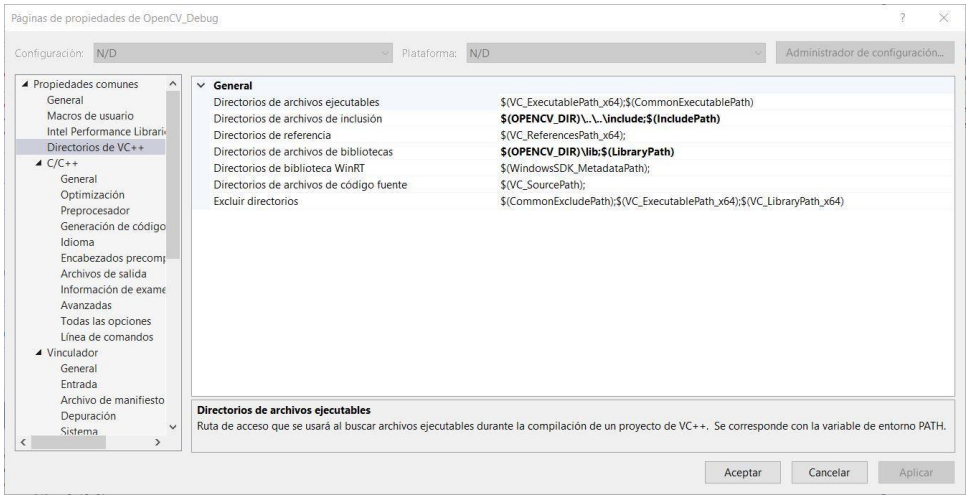


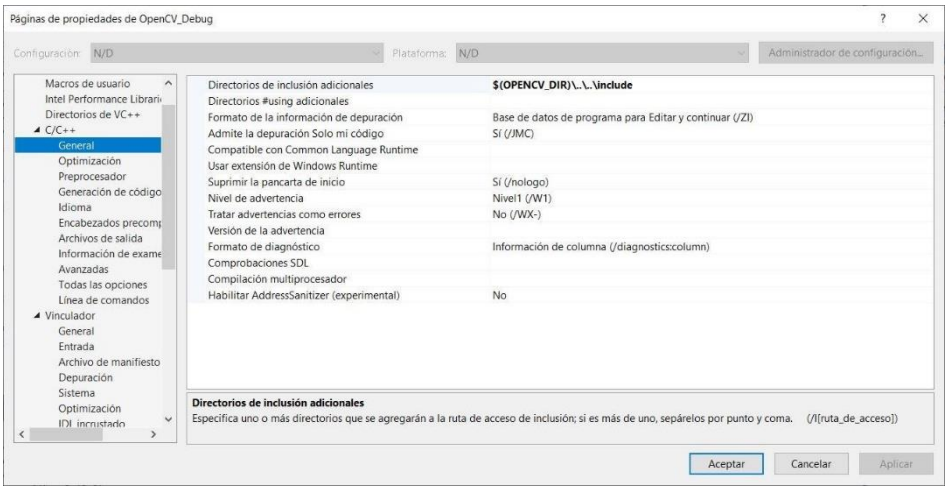


7. Copie los ficheros fuentes en la carpeta de la solución.

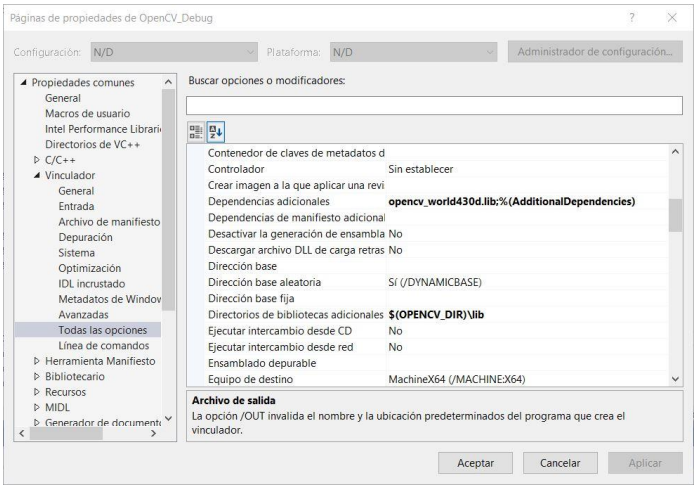
8. Cree dos propiedades personalizadas desde el administrador de propiedades una para Debug | x64 y otra para Release | x64. Edite los siguientes campos de las propiedades:

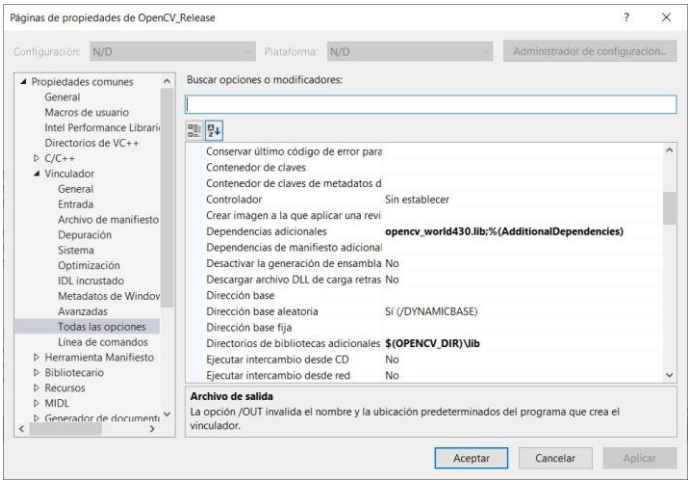




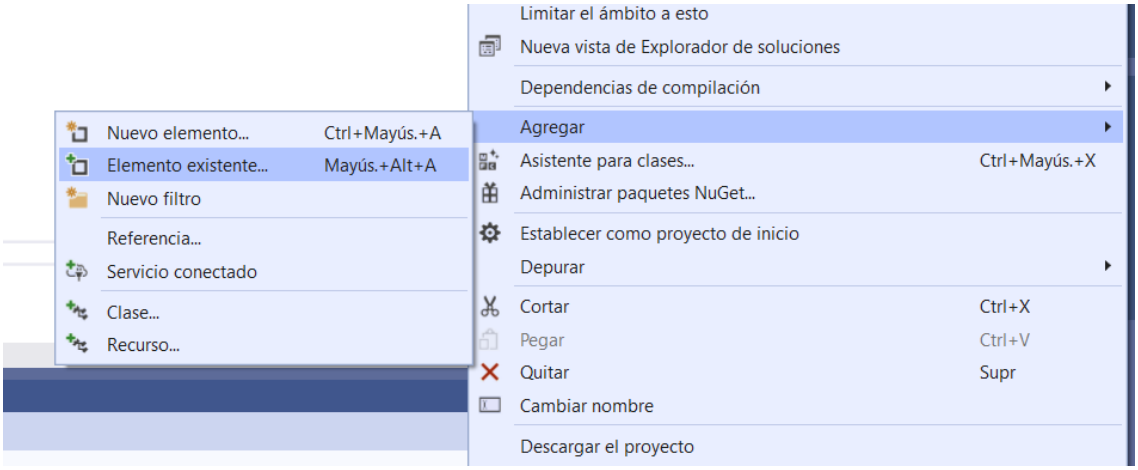


9. Añada las rutas de las librerías OpenCV a las propiedades creadas:

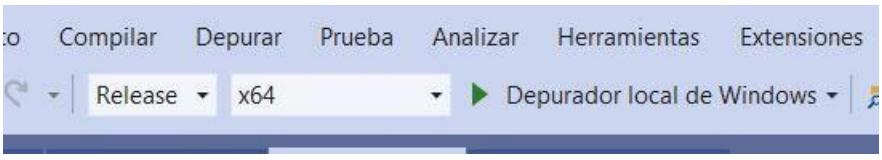




10. Añada los ficheros fuente al proyecto:



11. Seleccione la opción de ejecución como Release x64 o Debug x64:



Si todo ha ido bien, debería poder compilar el proyecto sin problemas.

10 Conclusiones y trabajos futuros:

Nuestro objetivo inicial era crear una herramienta útil para la comunidad de desarrolladores implicados en visión por computador. También nos habíamos propuesto dar visibilidad al método de detección de bordes.

Creemos que estos dos objetivos se han completado. Hemos creado una aplicación útil y sencilla mediante la cual los interesados pueden probar el método, tanto con imágenes reales como con las imágenes de test del modo de prueba. Además, con la publicación de la aplicación y de los ficheros fuentes en nuestro repositorio de GitHub, tanto los hispanohablantes como los anglosajones pueden acceder a la explicación teórica del método y a los detalles de su implementación.

Personalmente creo que he aprendido nuevos y desconocidos conceptos de procesamiento de imágenes con OpenCV, y gracias a la lectura de bibliografía relacionada con la visión por computador he adquirido un fondo de conocimiento que sin el desarrollo de este proyecto quizás no hubiera adquirido.

Actualmente, nuestra implementación funciona con un único suavizado previo a la detección. Como futuras mejoras al proyecto podemos completar el método con el empleo de suavizado iterativo de la imagen. También se podría investigar el uso de la librería de paralelización en GPU nativa de OpenCV, o el uso de OpenMP para paralelizar algunas subrutinas en CPU.

Como posible mejora final, podríamos investigar el uso del método en detección de bordes en volúmenes 3D.

11 Glosario de términos:

11.1 Filtro Canny:

El filtro Canny se caracteriza por detectar los bordes filtrando en 4 direcciones, ejes x e y, y las diagonales. Este filtro aplica primero un suavizado gaussiano y tiene la propiedad de no detectar dos veces el mismo píxel borde.

11.2 Filtro en el dominio de la frecuencia:

El filtrado en el dominio de frecuencia usa la transformada de Fourier sobre las frecuencias de la imagen, para luego aplicarle un filtro y finalmente generar la imagen transformada aplicando la transformada inversa de la representación frecuencial. Para usar este método hay que definir el concepto de frecuencia de una imagen. Se consideran frecuencias bajas a las regiones de la imagen con poca variación en la tonalidad. De modo contrario, se consideran frecuencias altas a aquellas zonas de la imagen con una alta variación en su tonalidad.

Los filtros pueden ser de tres tipos: paso bajo, paso alto, y paso banda. Este método funciona gracias al Teorema de Convolución de la transformada de Fourier.

11.3 Filtro Laplaciano:

El filtro laplaciano se utiliza para detectar los bordes de la imagen. A diferencia de los filtros Sobel que usa las derivadas parciales primeras, el laplaciano usa las derivadas parciales segundas de los tonos de los píxeles de la imagen de modo que la imagen resultado se calcula como:

$$L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

Para el cálculo de las derivadas parciales segundas se usan kernels de convolución. Dos ejemplos de estos kernels pueden ser:

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}, \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

La convención de signos se puede invertir sin que ello produzca alteración en el resultado. Más información en [L6].

11.4 Filtro Prewitt:

Este filtro usa dos kernels de convolución para calcular las derivadas parciales primeras. El uso de este filtro permite un cálculo aproximado del gradiente de un modo muy eficiente. Las derivadas parciales se obtienen a partir de la convolución de dos kernels, uno horizontal y el otro vertical. El cálculo sería de la siguiente manera:

$$\frac{\partial I}{\partial x} = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} * I \;\; ; \;\; \frac{\partial I}{\partial y} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} * I \;\; (*) \; operador \; de \; convolución$$

El módulo del gradiente se calcula como:

$$|\nabla I| = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}$$

Y el ángulo:

$$\theta = \; arctg\left(\frac{\frac{\partial I}{\partial y}}{\frac{\partial I}{\partial x}}\right)$$

11.5 Filtro Sobel:

El filtro Sobel funciona de modo análogo al filtro Prewitt, pero introduce un factor de incremento de la componente central de los kernels. Los kernels en este caso tienen la forma:

$$\begin{pmatrix} -1 & 0 & 1 \\ -a & 0 & a \\ -1 & 0 & 1 \end{pmatrix}; \begin{pmatrix} -1 & -a & -1 \\ 0 & 0 & 0 \\ 1 & a & 1 \end{pmatrix}$$

Donde a > 1.

11.6 Momento ortogonal:

Para obtener una descripción matemática y funcional de esta tecnología puede consultar los artículos [A11] y [A12].

11.7 Operadores convolucionales:

Un operador convolucional es una matriz cuadrada de tamaño nxn, donde n es impar. Estos operadores se usan para alterar la imagen, resaltando o suavizando características de esta. La convolución es una operación diferente al producto matricial, su forma de operar cosiste en sumar los productos uno a uno de la matriz del kernel y la submatriz de la imagen. El resultado de la operación es un único número que ocupará la posición central de la submatriz de la imagen, reemplazando al valor anterior.

La siguiente imagen muestra cómo actúa este operador.

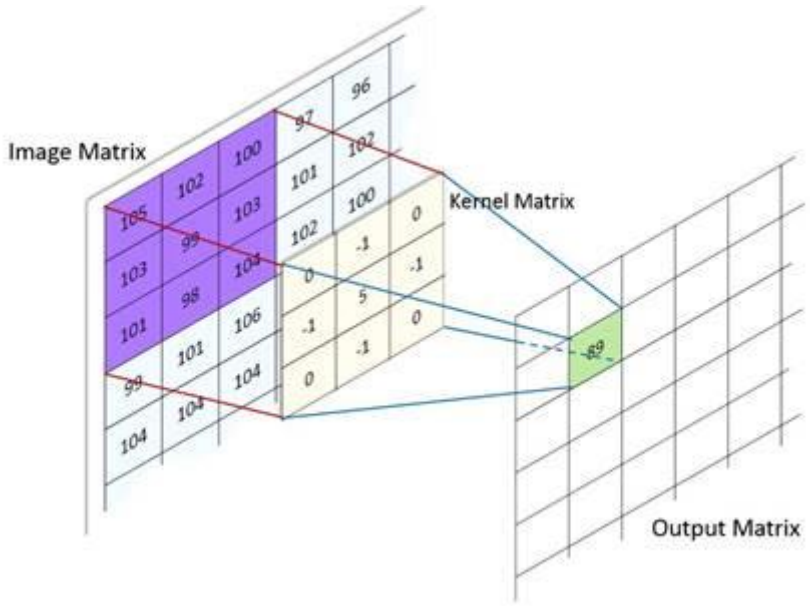


Fig12 Convolución

11.8 Operador Hueckel:

Ver punto 3.18 de [B8].

11.9 Ruido gaussiano:

El ruido gaussiano en imágenes digitales consiste en la alteración de los valores de los píxeles siguiendo una distribución gaussiana. En una distribución gaussiana más de 95% de los valores se encuentran entre $(\mu - 2\sigma, \mu + 2\sigma)$ y aproximadamente el 70% se encuentra entre $(\mu - \sigma, \mu + \sigma)$. Siendo μ la media de la distribución y σ la desviación estándar.

11.10 Ruido speckle:

Este tipo de ruido se produce en las imágenes obtenidas por láser o ultrasonido. La alteración de la imagen se produce cuando la dispersión de las ondas reflejadas se acopla constructiva o destructivamente. El acoplamiento constructivo produce tonos más brillantes y el destructivo atenúa el tono real.

12 Fuentes de información:

12.1 Bibliografía:

[B1] Fco. Javier Ceballos "Programación orientada a objetos con C++ (5ª Edición)" – Ed. Ra-Ma

[B2] Jesús Tomás, Vicente Carbonell, Antonio Albiol, Gonzalo Puga "Visión Artificial. Google Play Games. Android Wear. TV y Auto (3ª Edición)" – Ed. Marcombo

[B3] Adrian Kaehler, Gary Bradski "Learning OpenCV3 - Computer Vision in C++ with the OpenCV Library" – Ed. O'Reilly

[B4] Gloria Bueno García, José Luis Espinosa Aranda, Ismael Serrano Gracia, Oscar Déniz Suárez, Jesús Salido Tercero, Noelia Vállez Enano "Learning Image Processing with OpenCV" – Packt Publishing

[B5] P. J. Deitel, H. M. Deitel "Cómo programar C++ (6ª Edición)" – Ed. Pearson Prentice Hall

[B6] Scott Meyers "Effective STL – 50 Specific Ways to Improve Your Use of the Standard Template Library" Ed. Addison-Wesley Professional Computing Series

[B7] Nicolai M. Josuttis "The C++ Standard Library – A tutotial and reference (2ª Edición)" – Addison-Wesley

[B8] Gerhard X. Ritter, Joseph N. Wilson "Handbook of Computer Vision Algorithms in Image Algebra (2ª Edición)" – Ed. CRC Press

12.2 Artículos relacionados:

[A1] A. Trujillo et al. 2012 “Accurate subpixel edge location based on partial area effect”

[A2] Tabatabai, A.J.; Mitchell, O.R. 1984 “Edge Location to Subpixel Values in Digital Imagery”

[A3] Bin et al. 2008 “Subpixel edge location based on orthogonal Fourier-Mellin moments”

[A4] Hagara, Kulla 2011 “Edge Detection with Sub-pixel Accuracy Based on Approximation of Edge with Erf Function”

[A5] A. Fabijanska 2014 “Gaussian-Based Approach to Subpixel Detection of Blurred and Unsharp Edges”

[A6] MacVicar-Whelan, Binford 1991 “Stereo Vision for Facet Type Cameras”

[A7] Zhang, Meng, Li 2018 “Rock-Ring Accuracy Improvement in Infrared Satellite Image with Subpixel Edge Detection”

[A8] Qiucheng Sun et al. 2016 “A subpixel edge detection method based on an arctangent edge model”

[A9] Qiucheng Sun et al. 2014 “A robust edge detection method with sub-pixel accuracy”

[A10] James A. Moorer “The HueckelEdge Operator”

[A11] G.A. Papakostas, E.G. Karakasis, D.E. Koulouriotis 2013 “Orthogonal Image Moment Invariants: Highly Discriminative Features for Pattern Recognition Applications”

[A12] G.A. Papakostas, D.E. Koulouriotis, E.G. Karakasis 2010 “Computation strategies of orthogonal image moments: A comparative study”

[A13] Ziou, Tabbone 1998 “Edge Detection Techniques - An Overview”

[A14] Torre y Poggio 1986 “On Edge Detection”

12.3 Links de interés:

[L1] Documentación de OpenCV - https://docs.opencv.org/4.3.0/

[L2] Documentación Online de C++ - http://www.cplusplus.com/doc/

[L3] Documentación de Visual Studio - https://docs.microsoft.com/es-es/cpp/?view=vs-2019

[L4] Foro de incidencias - https://stackoverflow.com/

[L5] Consultas teóricas - https://es.wikipedia.org/wiki/Wikipedia:Portada

[L6] Laplaciano - https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm

[L7] Edge Detection in OpenCV 4.0, A 15 Minutes Tutorial - https://medium.com/sicara/opencv-edge-detection-tutorial-7c3303f10788