UNIVERSIDAD POLITÉCNICA DE VALENCIA

ESCUELA POLITÉCNICA SUPERIOR DE GANDIA

I.T. Telecomunicación (Sist. de Telecomunicación)

"Diseño e implementación de una aplicación informática para la identificación de zonas de interés en imágenes médicas"

TRABAJO FINAL DE CARRERA

Autor/es:

Juan Carlos José Salom

Director/es:

Jordi Bataller Mascarell

Roberto Sanz Requena

GANDIA, 2012

Índice general

1. Introducción **5**

2. Preprocesamiento de imágenes **7**
2.1. Introducción 7
2.2. Definiciones 7
2.3. Operaciones a nivel de *pixel* 8
2.3.1. Reescalado de intensidad 8
2.3.2. Ecualización del histograma de la imagen 9
2.4. Operadores locales 10
2.4.1. Reducción de ruido mediante filtrado por valor medio 11
2.4.2. Realzamiento de bordes 11

3. Técnicas de segmentación en imágenes médicas **13**
3.1. Introducción 13
3.2. Segmentación basada en regiones 14
3.2.1. *Thresholding:* el nivel umbral 14
3.2.2. Crecimiento de regiones 16
3.3. Segmentación basada en contornos 16
3.3.1. Modelos deformables 19
3.3.1.1. El método *level set* 20

4. Descripción de la aplicación **23**
4.1. Tecnologías 23
4.1.1. Bibliotecas de procesamiento de imágenes médicas 23
4.1.2. *Insight Segmentation and Registration Toolkit* 24
4.1.3. Otras bibliotecas 25
4.1.3.1. *Boost* 25
4.1.3.2. *CMake* 26
4.2. Aplicación de segmentación 27
4.2.1. Introducción 27
4.2.2. Pruebas previas: selección de un método adecuado 27
4.2.3. Librerías *ITK* utilizadas 28
4.2.4. Análisis del código fuente 31
4.2.5. Resultados 39

5. Guía del usuario **43**
5.1. Uso del software 43
5.2. Compilación del software 45

6. Conclusiones y trabajo futuro **49**

Bibliografía **50**

Capítulo 1

Introducción

En la actualidad existen numerosas pruebas de diagnosis mediante imagen en el ámbito médico que proporcionan información muy valiosas acerca de las posibles patologías objeto de estudio. Más allá de la observación directa, estas imágenes médicas contienen información muy útil que puede extraerse mediante un procesamiento informático. En ambos casos –observación directa o procesamiento computacional– es generalmente imporante, si no imprescindible, identificar estructuras anatómicas como órganos completos (hígado, riñón, etc) partes o lesiones en ellos, e incluso inervaciones o sistemas sanguíneos. Este reconocimiento de zonas de interés en una imágen médica se conoce como segmentación. Existe una gran cantidad de técnicas matemáticas y algorítmicas aplicables en la segmentación, desarrolladas desde los años sesenta. El problema fundamental práctico es identificar cuál es más apropiada para el caso de interés particular, ya que no existe una solución utilizable de forma general.

En nuestro caso, el problema que se nos propone es identificar el hígado a partir de una serie de imágenes obtenidas mediante la utilización de la tomografía axial computarizada. No sólo se trata de un estudio genérico sino que se pretende disponer de una aplicación funcional que realice dicha tarea. Las imágenes médicas nos han sido proporcionadas por la clínica Quirón de Valencia, a través de nuestro cotutor Dr. Roberto Sanz Requena.

Por tanto en este proyecto, para abordar el problema de la segmentación del hígado en imágenes médicas, hemos realizado en primer lugar un estudio teórico de las técnicas aplicables evaluando su idoneidad en nuestro caso. A continuación hemos buscado qué herramientas y bibliotecas se encuentran disponibles para su utilización en el desarrollo de programas, analizando su conveniencia para nuestro problema, efectuando pruebas de concepto. Finalmente, una vez elegidas la biblioteca y la técnica de segmentación hemos desarrollado la aplicación informática.

La presente memoria está organizada de esta forma. En el capítulo 2 hablaremos de las técnicas genéricas de preprocesamiento de imágenes y cómo nos pueden ayudar a mejorar la segmentación e incluso hacerla posible. Una vez expuestos ciertos conceptos necesarios, en el capítulo 3 se introducirá al lector en cuestiones teóricas sobre segmentación de imágenes médicas, explicando con cierto detalle las cuestiones más importantes y haciendo mención de algunas técnicas de ámbitos más avanzados. El capítulo 4 consta de una descripción detallada del método que se ha seguido para obtener la segmentación automatizada del hígado, conteniendo además el código fuente de la aplicación explicado paso a paso. Finalmente, el capítulo 5 contiene las instrucciones necesarias para instalar y configurar el software necesario para compilar y ejecutar la aplicación de segmentación que hemos creado, centrándonos principal-

mente en sistemas *UNIX*.

Capítulo 2

Preprocesamiento de imágenes

2.1. Introducción

Las técnicas de preprocesamiento de imágenes se utilizan para mejorar las imágenes dadas y que sus características sean mejor percibidas tanto por el ojo humano como por los sistemas de análisis automatizado. Así pues, el observador será capaz de percibir detalles y sutilezas que no son inmediatamente visibles en la imagen original, debido a un contraste insuficiente, un nivel de ruido digital demasiado alto, o un rango dinámico que no encaja con nuestro *display,* entre otros fenómenos.

Un aspecto importante sobre los algoritmos de mejora es que nunca debemos esperar que proporcionen información que no se encuentre en la imagen original. El uso incorrecto de estas técnicas puede llevar a resultados no deseados, como la pérdida de información en la imagen o la obtención de una representación desmejorada respecto a la original si la característica que estamos intentando mejorar no se encuentra en el área objeto del análisis.

2.2. Definiciones

En lo que sigue, consideraremos una imagen digital como un *array* bidimensional de números que representan la distribución continua de intensidad de una determinada señal espacial, que es muestreada a intervalos regulares y cuantizada a un número finito de niveles. Cada elemento de este *array* se refiere a un *pixel* de la imagen. La función matemática que define la imagen digital es una señal de intensidad espacialmente distribuida, $f(m, n)$, donde m y n representan la posición de cada *pixel* respecto a dos ejes ortogonales generalmente definidos como horizontal y vertical. Asumiremos también que la imagen posee M filas y N columnas y que sus *pixels* tienen P niveles diferentes de intensidad, cuantizados desde 0 hasta $P - 1$.

El histograma de la imagen, una herramienta muy útil en imagen digital, se define como un vector que contiene la cantidad de *pixels* en la imagen para un determinado nivel de intensidad o nivel de gris[1]. El histograma de una imagen, $h(i)$, analíticamente

[1]Los niveles de intensidad son también referidos como *niveles de gris*, ya que el valor mínimo de intensidad representa el color negro y el máximo el blanco, teniendo diferentes tonalidades de gris

será:

$$h(i) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \delta(f(m,n) - i), \quad i = 0, 1, ..., P-1 \tag{2.2.1}$$

donde

$$\delta(x) = \begin{cases} 1, & si\, x = 0, \\ 0, & en\, cualquier\, otro\, caso \end{cases} \tag{2.2.2}$$

Existe otra técnica de vital importancia dentro del procesamiento de imágenes: la convolución. Explicada en palabras simples, trata de generar valores medios para cada *pixel* de la imagen tomando como argumentos de la operación tanto el valor del propio *pixel* como el de sus vecinos: se multiplican sus valores por los de una matriz que llamaremos *kernel* y el resultado de sumarlos todos será el nuevo valor para el *pixel*. Obtenemos la imagen de salida repitiendo esta operación para cada *pixel* de la imagen de entrada. Esto es especialmente útil, en numerosos casos, para la eliminación de ruido, ya que cada *pixel* de la nueva imagen tiene un valor medio generado a partir de un conjunto de *pixels* vecinos y cualquier irregularidad quedará fuertemente suavizada. Considerando un *kernel* $w(k,l)$ de dimensiones $(2K+1) \times (2L+1)$ existirá siempre un punto origen $(k,l) = (0,0)$. La convolución de una imagen con dicho *kernel* será:

$$g(m,n) = w(k,l) * f(m,n) = \sum_{k=-K}^{K} \sum_{l=-L}^{L} w(k,l) \cdot f(m-k, n-l), \tag{2.2.3}$$

siendo $g(m,n)$ la imagen de salida. A parte de su utilidad en la reducción de ruido, diferentes tipos de *kernels* se pueden aplicar a la imagen para resaltar propiedades interesantes o alterar sus características, siendo en este sentido crucial el valor de los coeficientes del *kernel*.

2.3. Operaciones a nivel de *pixel*

Se presentan a continuación métodos de mejora de imágenes que sólo dependen del nivel de intensidad del *pixel* y no toman en consideración las características de su vecindad ni las propiedades generales de la imagen.

2.3.1. Reescalado de intensidad

El reescalado de intensidad se usa cuando los valores de la imagen adquirida producen un rango dinámico[2] que sobrepasa el de nuestro sistema de *display,* o viceversa. También puede resultar interesante en el caso de que la mayor parte de los datos valiosos se encuentren concentrados en bandas de intensidad específicas, pues esta técnica permite, a costa de obviar el resto de información, expandirlas para que ocupen todo el rango dinámico del *display,* produciendo una imagen más adecuada para el observador. Si llamamos f_1 y f_2 los límites que definen la banda de intensidad que nos interesa, la función de escalado podría definirse como:

$$e = \begin{cases} f, & si\, f_1 \leq f \leq f_2 \\ 0, & en\, cualquier\, otro\, caso \end{cases} \tag{2.3.1}$$

en sus valores intermedios.

[2]El rango dinámico se define como la razón entre el menor y el mayor valor de una determinada cantidad.

Figura 2.1: Ejemplo de imagen sin un correcto reescalado de intensidad (izquierda) y la misma imagen después del proceso, ocupando el mismo rango dinámico que el *display (derecha).*

$$g = \left\{ \frac{e - f_1}{f_2 - f_1} \right\} \cdot (f_{max}) \tag{2.3.2}$$

donde e representa una imagen intermedia, g es la imagen de salida y f_{max} es el nivel máximo de intensidad del *display.* En la figura 2.1 puede apreciarse gráficamente la efectividad de este proceso.

2.3.2. Ecualización del histograma de la imagen

Pese a que el reescalado de intensidad es muy adecuado para resaltar información de la imagen en una banda de intensidad específica, a menudo no disponemos de la información necesaria para identificar las bandas de intensidad que nos van a ser útiles. En estos casos, es más conveniente distribuir la intensidad en la imagen tan uniformemente como sea posible para maximizar la información transmitida por la imagen al observador, haciendo que el histograma sea lo más plano posible. Este procedimiento está basado en la interpretación teórica de que el histograma normalizado de la imagen representa la función densidad de probabilidad de intensidad de la imagen.

En la sección 2.2 hemos visto que el histograma se define como $h(i)$, teniendo la imagen $P-1$ niveles de gris. Como el número total de *pixels* en la imagen es $M \cdot N$, para distribuir uniformemente el perfil de intensidad de la imagen cada valor de intensidad en el histograma deberá tener $\frac{M \cdot N}{P}$ *pixels,* lo que generaría un histograma totalmente plano. En general, es posible mover *pixels* hacia un valor de intensidad que provoque un incremento en el número de *pixels* en ese nuevo valor de intensidad, pero nunca es aceptable el proceso inverso, es decir, reducir el número de *pixels* que contiene un determinado valor de intensidad para alcanzar el valor ideal $\frac{M \cdot N}{P}$.

Un proceso verdaderamente simple de obtener una uniformidad aproximada es el

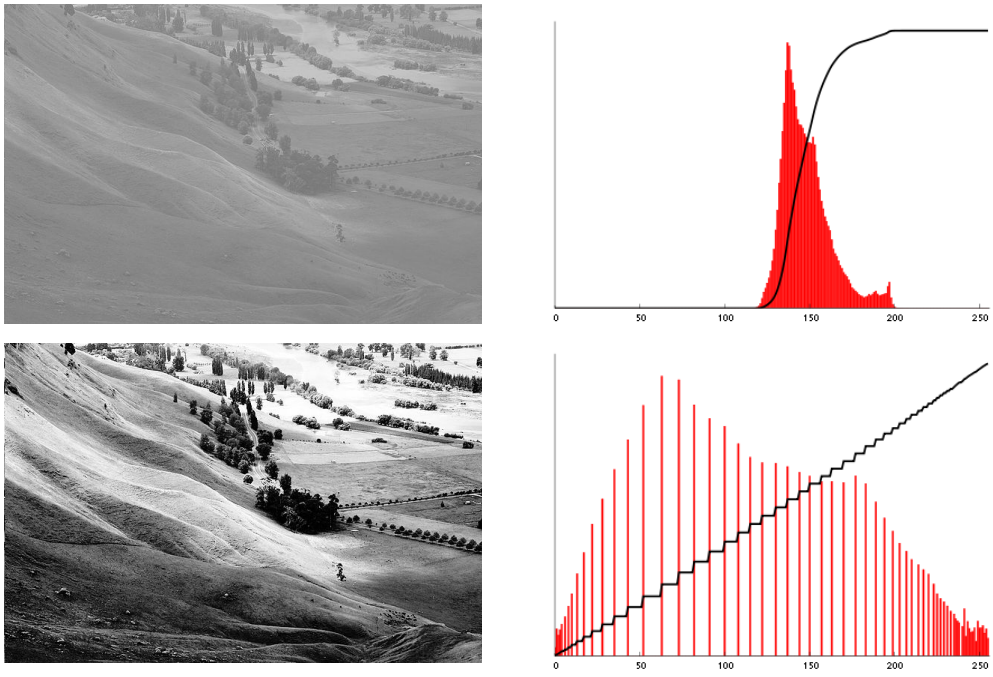

Figura 2.2: Imagen antes y después de la normalización de su histograma (filas superior e inferior respectivamente). En el histograma inferior podemos ver cómo los valores se encuentran repartidos de manera más uniforme en ambos ejes, dando una mejor visibilidad a la imagen.

de normalizar el histograma, esto es, acumulando los valores de intensidad progresivamente y dividiendo el resultado por el número total de *pixels*,

$$H(j) = \frac{1}{M \cdot N} \sum_{i=0}^{j} h(i), \quad j = 0, 1, ..., P-1 \tag{2.3.3}$$

Asimismo, la imagen mejorada $g(m, n)$ tendrá un histograma máximamente plano si la definimos como

$$g(m, n) = (P-1) \cdot H(f(m, n)) \tag{2.3.4}$$

La figura 2.2 muestra el efecto de la normalización.

2.4. Operadores locales

L*os* operadores locales permiten mejorar la imagen asignando nuevos valores a los *pixels* de forma que este nuevo valor dependerá, además del *pixel* en cuestión, de los que están a su alrededor. Si ampliamos el radio de acción de estos operadores para incluir más *pixels* vecinos es posible refinar la imagen de salida obteniendo resultados diferentes. En esta sección se presentan los operadores locales más comunes en procesado de imágenes médicas.

Figura 2.3: Efecto del filtrado de ruido por valor medio usando una máscara de tamaño 3×3.

2.4.1. Reducción de ruido mediante filtrado por valor medio

El filtrado por valor medio se consigue mediante la convolución de la imagen con un *kernel* de tamaño $(2K+1) \times (2L+1)$ donde cada uno de sus coeficientes tiene un valor inverso al número total de coeficientes del *kernel*. Por ejemplo, si $K = L = 1$ obtenemos un *kernel* de 9 elementos como el siguiente,

$$w(k,l) = \begin{pmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{pmatrix} \tag{2.4.1}$$

al cual nos referimos como máscara de 3×3[3]. Normalmente, este tipo de filtro de suavizado reduce el ruido en la imagen a expensas de su nitidez, ya que provoca un efecto borroso y los contornos presentes en la imagen pierden cierto grado de percepción. El tamaño del *kernel* es un factor muy determinante en la correcta aplicación de esta técnica, ya que los detalles de la imagen con unas dimensiones aproximadamente iguales o menores a las del *kernel* serán prácticamente eliminadas mientras que los detalles o estructuras más grandes apenas se verán afectadas. El grado de reducción de ruido es directamente proporcional al tamaño del *kernel*, siendo mayor cuanto mayor sea éste. En la figura 2.3 podemos ver el resultado de aplicar una máscara de dimensiones 3×3.

2.4.2. Realzamiento de bordes

El realzamiento de bordes es de mucha importancia puesto que el sistema visual humano utiliza los contornos como factor clave en la comprensión de los contenidos de una imagen. Los bordes pueden ser identificados y realzados según su orientación, de forma selectiva. Además, las imágenes con bordes realzados pueden ser combinadas con las originales para preservar el contexto, en caso de que se degrade la información.

[3]Nótese que esta convolución básicamente realiza una operación de media aritmética de los valores de un grupo de *pixels*.

Es fácil ver que la convolución con alguno de los siguientes *kernels* producirá un realzado de las líneas y contornos horizontales:

$$w_{H1}(k,l) = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} \tag{2.4.2}$$

$$w_{H2}(k,l) = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \tag{2.4.3}$$

mientras que estos otros dos,

$$w_{V1}(k,l) = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} \tag{2.4.4}$$

$$w_{V2}(k,l) = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} \tag{2.4.5}$$

producirán un realzado de contornos verticales. Existe también la posibilidad de realzar ambas direcciones simultáneamente mediante una máscara omnidireccional como la siguiente:

$$K_{HP}(k,l) = \begin{pmatrix} -1/8 & -1/8 & -1/8 \\ -1/8 & 1 & -1/8 \\ -1/8 & -1/8 & -1/8 \end{pmatrix} \tag{2.4.6}$$

Suponiendo que estos *kernels* se aplican sobre una imagen de valores positivos, nótese que el resultado puede ser una imagen con valores tanto positivos como negativos. Esto se puede evitar añadiendo un valor de compensación o tomando sólo el valor absoluto para los *pixels* de la imagen de salida. Estas son buenas ideas si lo que queremos es obtener una imagen en la que sólo es relevante la información de sus contornos. Si, en cambio, buscamos una imagen de salida en su contexto completo pero con los bordes realzados debemos realizar una unión de la información de ambas imágenes.

Capítulo 3

Técnicas de segmentación en imágenes médicas

3.1. Introducción

La segmentación, o separación de estructuras de interés respecto al fondo o respecto a otras estructuras, es una función de análisis fundamental para la cual han sido desarrollados numerosos algoritmos en el campo del procesamiento de imágenes. En imágenes médicas, la delineación automatizada de diferentes componentes de la imagen es utilizada para analizar estructuras anatómicas y tipos de tejido, distribución espacial de su función o actividad y regiones patológicas. La segmentación también puede ser usada como paso inicial para la visualización y compresión de la imagen. Normalmente, la segmentación de un objeto se consigue bien identificando todos sus *pixels* o *voxels* que lo forman o bien localizando aquellos que pertenecen a su contorno. El criterio más fundamental se basa en la intensidad de los *pixels*, aunque otros atributos, como su textura, que puede ser asociada con cada *pixel*, también son susceptibles de ser usados en el proceso.

Como la segmentación requiere la clasificación de *pixels*, es a menudo entendida como un problema de reconocimiento de patrones y por tanto estudiada mediante técnicas asociadas a este respecto. Particularmente, en imágenes médicas, que pueden tener una variabilidad muy elevada en sus datos, las técnicas de reconocimiento de patrones son de especial interés, pues proporcionan flexibilidad y una automatización adecuada. Otro procedimiento utilizado se basa en las redes neuronales, donde la clasificación se realiza mediante cálculos de procesamiento paralelo distribuido y no lineal. Hoy en día tenemos disponibles numerosas estructuras y algoritmos de redes neuronales que pueden ser usados para la segmentación de imágenes médicas.

Existe además un procedimiento de segmentación relativamente nuevo basado en modelos deformables que proporciona un mecanismo considerablemente diferente de las técnicas fundamentales y los métodos de reconocimiento de patrones. En este procedimiento, un modelo de contorno, que es flexible, se sitúa en la vecindad de la zona que se desea segmentar y de forma iterativa se va ajustando hasta encajar con el contorno de la estructura de interés. Los modelos deformables son especialmente útiles en la segmentación de imágenes con anomalías, ruido o con contornos no muy definidos entre sus estructuras. Un aspecto importante de los patrones deformables es que es posible incorporar información *a priori* en el contorno base del objeto, aunque tam-

bién posee algunos puntos débiles, como la posibilidad de que el contorno no converja hacia el deseado por tener unas dimensiones inadecuadas o que directamente no converja hacia ninguna posición cóncava.

En algunos procedimientos de análisis de imágenes, la presencia de estructuras con diferentes propiedades sugiere el uso de varias técnicas de segmentación de forma secuencial, estando éstas especialmente diseñadas para el caso. Por ejemplo, se pueden dedicar los pasos previos a reducir el tamaño de los datos disponibles sin perder demasiada información, y usar los restantes pasos para refinar el resultado mediante técnicas más elaboradas y robustas pero que generan un mayor coste computacional. La mejor opción en la elección de las técnicas a usar y puede que también su orden dependerá de la naturaleza del problema así como de nuestra capacidad computacional.

A continuación pasamos a describir brevemente las técnicas de segmentación básicas más importantes, que podemos clasificar en dos grandes grupos: la segmentación basada en regiones y la segmentación basada en contornos.

3.2. Segmentación basada en regiones

El principio básico de la segmentación por regiones es la búsqueda de zonas en la imagen que cumplan ciertos requisitos de homogeneidad, de manera que puedan ser diferenciadas del resto de elementos.

3.2.1. *Thresholding:* el nivel umbral

El *thresholding* es uno de los métodos de segmentación de regiones más comunes e intuitivos, ya que se basa únicamente en el valor de intensidad de sus *pixels*. Su estudio se puede diferenciar según 2 procedimientos generales: el *global thresholding* y el *local* –o *adaptive*– *thresholding*.

El *global thresholding* parte de la premisa de que la imagen tiene un histograma[1] bimodal y en consecuencia el objeto puede ser aislado del resto por una simple operación de comparación de sus valores tonales con un cierto valor umbral T. Supongamos que tenemos una imagen $f(x, y)$ con un histograma como el de la figura 3.1. Puede apreciarse que la imagen posee niveles agrupados alrededor de dos modos dominantes, y por tanto la manera más obvia de aislar nuestro objeto del resto será escoger un valor umbral T para separar ambos modos. La imagen $g(x, y)$ resultante, después de aplicar nuestro criterio umbral, está compuesta únicamente de valores binarios donde representamos con 1 el subconjunto que pertenece al objeto y con 0 el resto. De una forma más analítica, podemos decir que:

$$g(x,y) = \begin{cases} 1, & si\, f(x,y) > T \\ 0, & si\, f(x,y) \leq T \end{cases} \tag{3.2.1}$$

Existen varias maneras de encontrar un nivel de umbral T adecuado. La más sencilla consiste en elegir el valor mínimo que se encuentre entre dos modos, como en el ejemplo de la figura 3.1. Otro método un poco más avanzado consiste en minimizar la

[1] Histograma: Dentro del marco de la imagen digital, los histogramas representan gráficamente el número de *pixels* frente a cada valor tonal. En escala de grises, se entiende el valor tonal como la intensidad del *pixel*.

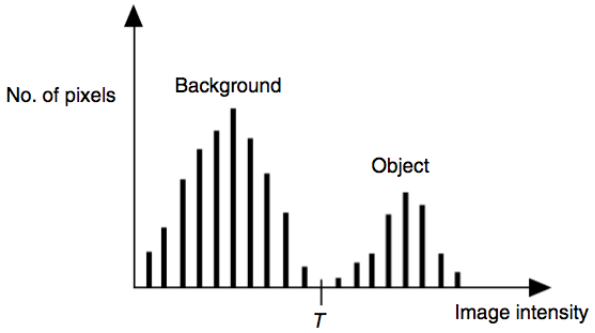

Figura 3.1: Ejemplo de histograma bimodal con un valor umbral T.

Figura 3.2: Imagen binaria, resultado del proceso de *thresholding*.

probabilidad de error mediante métodos de clasificación. Por ejemplo, podemos calcular el error cometido como la suma del número total de *pixels* que pertenecen al objeto y que hemos clasificado fuera por error, y el número de *pixels* que no pertenecen al objeto y hemos clasificado como parte del objeto por error. De esta manera tenemos un procedimiento sencillo para refinar el valor umbral, aunque tendremos que diseñar un algoritmo que pueda decidir con relativo acierto cuándo un *pixel* pertenece o no al objeto. En imágenes con histograma bimodal esta tarea no suele resultar demasiado complicada.

En determinados casos, en cambio, no puede encontrarse un umbral global a partir del histograma. Además, puede ocurrir también que un sólo nivel umbral sea insuficiente para obtener unos resultados de segmentación adecuados. Esto se da si por ejemplo el fondo de la imagen no es constante o el contraste de los objetos de la imagen es variable. La técnica del *local thresholding* se basa en la partición de la imagen en varias sub-imágenes, de manera que en cada una de éstas se incluya parte del objeto y parte del fondo. Mediante el análisis del histograma de cada una de las sub-imágenes, intentaremos encontrar un nivel umbral adecuado para cada una de ellas: en el caso de que su histograma sea bimodal, el umbral será el mínimo entre picos, al igual que en el caso anterior. Si por el contrario su histograma es unimodal, tendremos que interpolar los valores umbral de las sub-imágenes de su entorno cercano.

3.2.2. Crecimiento de regiones

A diferencia de los métodos de *thresholding,* donde la clave radica en la diferencia de intensidades de los *pixels*, el crecimiento de regiones se centra en encontrar grupos de *pixels* con intensidades similares. El algoritmo parte siempre de un *pixel* –o un grupo de ellos– que pertenece al objeto o estructura de interés, llamado semilla o *seed*, el cual puede proporcionarse manualmente por inspección visual o mediante un algoritmo encargado de encontrarlo automáticamente. Seguidamente, los *pixels* vecinos son examinados uno a uno y añadidos a la región creciente, eso sí, en el caso de que sean suficientemente similares basándonos en algún criterio de homogeneidad. Este procedimiento se repetirá iterativamente hasta que ya no puedan ser añadidos más *pixels* a la región.

Los resultados del crecimiento de regiones dependen fuertemente de la elección del criterio de homogeneidad –también llamado test de uniformidad–, es decir, en decidir cuándo un *pixel* pertenece o no a la región. Un posible método podría consistir en comparar el valor de intensidad del *pixel* con el valor de intensidad medio en una determinada área: si la diferencia es menor a, por ejemplo, dos veces la desviación típica de la intensidad en la región, se clasifica el *pixel* como perteneciente a ésta. De lo contrario, el *pixel* será clasificado como parte del contorno. Otra cosa a tener presente es que diferentes *seeds* no tienen por qué converger hacia la misma región como resultado.

Los algoritmos de crecimiento de regiones tienen la ventaja de que son capaces de segmentar la gran mayoría de regiones que posean propiedades similares y que se encuentren espacialmente separadas del resto.

3.3. Segmentación basada en contornos

En una imagen, los límites o contornos de los objetos se definen mediante el gradiente de intensidad de sus *pixels,* que será una aproximación de la derivada de primer

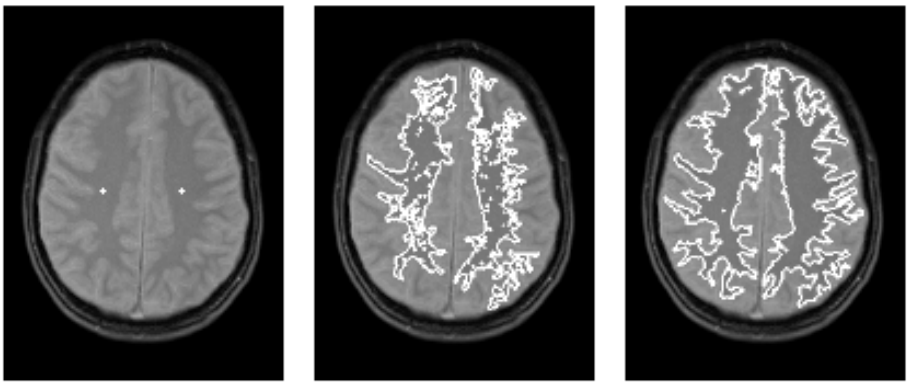

Figura 3.3: Evolución temporal típica de un algoritmo de crecimiento de regiones.

orden de la función que la representa. Dada una determinada imagen $f(x, y)$, podemos calcular la magnitud del gradiente como

$$|G| = \sqrt{G_x^2 + G_y^2} = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \tag{3.3.1}$$

y su dirección como

$$D = tan^{-1}\left(\frac{G_y}{G_x}\right) \tag{3.3.2}$$

donde G_x y G_y son los gradientes en las direcciones x e y, respectivamente. Como la función imagen es de naturaleza discreta no permite la aplicación de la diferenciación continua y los gradientes deben ser calculados mediante aproximaciones usando, entre otros, el operador de *Sobel*, el operador de *Prewitt* o el de *Frei-Chen*.

En imagen digital, la mayoría de los operadores gradiente implican el cálculo de convoluciones, como pueden ser las sumas ponderadas de intensidades de *pixel* en un entorno local. Estos valores ponderados pueden representarse en forma de *array* numérico –entiéndase como máscara o *kernel*–. Por ejemplo, en el caso del operador de *Sobel* 3×3 existen dos máscaras como las siguientes,

$$\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \qquad \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \tag{3.3.3}$$

donde la primera de ellas se utiliza para el cálculo de gradiente en la dirección horizontal G_x y la segunda para el del gradiente vertical G_y. La imagen que representa el gradiente de magnitud se genera combinando G_x y G_y mediante la ecuación 3.2. En la figura 3.4 se muestra el resultado de aplicar las máscaras de Sobel por separado sobre una imagen arbitraria.

Algunos algoritmos de detección de contornos usan el operador gradiente seguido de algún operador de valor umbral para decidir si se ha encontrado –o no– un contorno, dando como resultado una imagen binaria. Nótese que la elección del valor umbral en estos casos no es sencilla puesto que generalmente, en imágenes médicas,

Figura 3.4: Efecto del operador de Sobel 3x3 sobre la famosa «Lena».

suelen coexistir estructuras con contornos sensiblemente más sutiles que el resto.

Las técnicas de segmentación basadas en contornos son considerablemente rápidas desde el punto de vista computacional y no requieren información *a priori* sobre la topología de la imagen, pero es bastante común que el contorno de salida no encierre completamente el objeto a segmentar, por lo que se hará necesaria alguna técnica de post-procesado para enlazar los espacios vacíos. Como primera solución y más simple se puede examinar la periferia de los *pixels* pertenecientes a un contorno (por ejemplo en áreas de 3×3 o 5×5) e ir llenando los espacios mediante algún mecanismo que sea capaz de detectar similitud de dirección o intensidad en estos *pixels*. Sin embargo, esto resulta computacionalmente costoso y por tanto no demasiado factible en muchos casos. Como siguiente paso y para remediar este problema, es razonable el uso de técnicas híbridas semiautomáticas, donde el usuario se encargará de enlazar manualmente los bordes cuando éstos resulten ambiguos y el algoritmo de enlazado no sea capaz de continuar.

Además del operador gradiente existe otro conocido operador que también suele utilizarse para la detección de bordes; como ya sabemos los máximos de la derivada de primer orden se corresponden con los ceros en la derivada de segundo orden, por tanto, el operador de *Laplace* se convierte en una herramienta útil para este propósito. Se define el *Laplaciano* ∇^2 de una función $f(x, y)$ como:

$$\nabla^2 f(x,y) = \frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2} \tag{3.3.4}$$

Al igual que en el caso anterior y debido a la naturaleza discreta de $f(x, y)$ es necesario aproximar el operador mediante máscaras de convolución $N \times N$. Las siguientes matrices representan diferentes aproximaciones para $N = 3$:

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \quad \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} \quad \begin{pmatrix} -1 & -2 & -1 \\ -2 & 4 & -2 \\ -1 & -2 & -1 \end{pmatrix} \tag{3.3.5}$$

Dicho esto, para formar los contornos en la imagen simplemente deberemos localizar

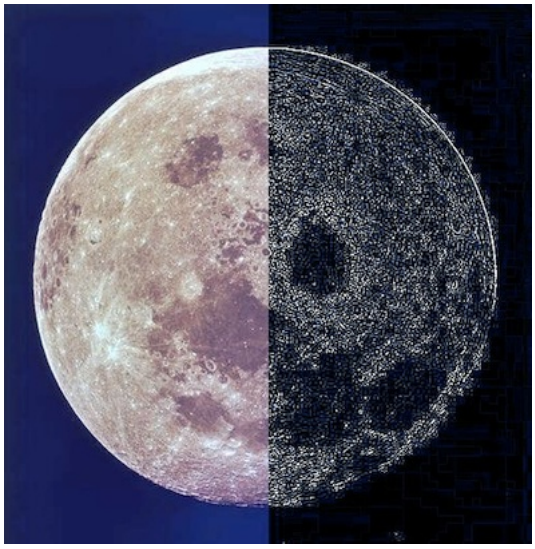

Figura 3.5: En esta imagen de la Luna la parte derecha ha sido procesada con un filtro de *Laplace* de tamaño 3×3, siendo la parte izquierda su aspecto original.

los *pixels* en los que el Laplaciano toma el valor cero.

En general todos los métodos de detección de contornos que están basados en el operador de *Laplace* o en el gradiente son muy sensibles al ruido presente en la imagen y es muy común la aplicación previa de métodos de suavizado que reduzcan su efecto.

3.3.1. Modelos deformables

Pese a la existencia de todas las técnicas anteriores y otras mucho más avanzadas que aquí no se han expuesto, la segmentación de imágenes continúa siendo una ciencia con muchas dificultades. Particularmente, la segmentación de imágenes médicas implica trabajar con una alta variabilidad en la forma en los objetos y de calidad de imagen, así como presencia de ruido y de artefactos de muestreo.

Para abordar estas dificultades, los *modelos deformables* –también llamados *contornos activos* o *snakes*– han sido extensamente estudiados y ampliamente usados en la segmentación de imágenes médicas con resultados prometedores. Los modelos deformables son curvas o superficies definidas dentro del dominio de una imagen que pueden moverse bajo la influencia de *fuerzas internas*, las cuales son definidas mediante la curva o la superficie en sí, y *fuerzas externas*, que son calculadas con los datos que contiene la imagen. Las fuerzas internas están diseñadas para mantener el modelo suave[2]. Las fuerzas externas, por otro lado, están definidas para mover el modelo por los dominios del objeto u otras zonas de interés. Forzando los contornos extraídos para que sean suaves y incorporando información previa sobre la forma del objeto, los modelos deformables se convierten en una herramienta robusta al ruido e irregularidades y permite describir los contornos integrándolos en una descripción matemática coherente y consistente. Más aún, como los modelos deformables están

[2]Esto es, sin deformaciones muy angulosas o exageradas.

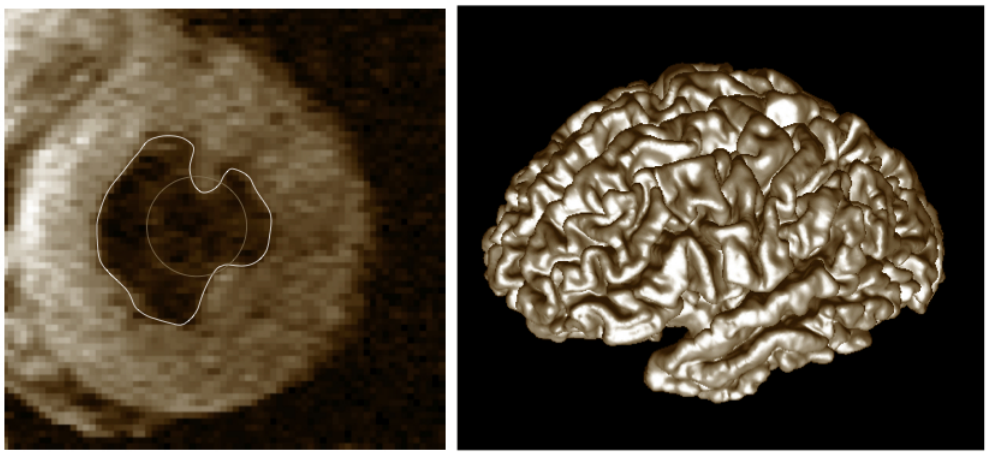

Figura 3.6: *Izquierda:* uso de un contorno deformable para extraer la pared interna del ventrículo de un corazón humano. El círculo representado en gris es la forma inicial del contorno. *Derecha:* ejemplo de uso de una superficie deformable para reconstruir la superficie del cerebro a partir de una imagen 3D.

definidos en el espacio continuo, la representación resultante del contorno puede abordar precisión a nivel de *subpixel,* lo cual es una característica altamente deseada en el ámbito de imágenes médicas.

Existen básicamente dos tipos de modelos deformables: los *paramétricos* y los *geométricos*. Los modelos deformables paramétricos representan curvas y superficies explícitamente en sus formas paramétricas durante la deformación, lo que permite una interacción directa con el modelo y una representación compacta y por tanto una implementación en tiempo real rápida. Sin embargo, la adaptación topológica del modelo, como la división o unión de partes durante la deformación, puede resultar difícil al usar modelos paramétricos. Por otro lado, los modelos deformables geométricos pueden abordar los cambios topológicos de forma natural. Estos modelos, basados en la teoría de *evolución de curvas*, representan curvas y superficies implícitamente como *conjuntos de nivel –level sets–* de una función escalar de mayor orden dimensional. Su parametrización se computa una vez completada la deformación, permitiendo así una adaptación topológica cómoda. A pesar de esta diferencia fundamental, ambos métodos recaen en principios fundamentales muy similares.

A continuación se expone brevemente el método *Level Set,* en el cual se basa la teoría de modelos deformables geométricos, ya que es uno de los principios utilizados por el algoritmo de segmentación que se presenta en el siguiente capítulo.

3.3.1.1. El método *level set*

El método *level set* se usa principalmente por su propiedad de adaptabilidad topológica, además de proporcionar la base matemática utilizada por los modelos deformables geométricos.

En el método *level set*, una curva está representada como un conjunto de nivel de una función escalar bidimensional –llamada función *level set*– que normalmente se

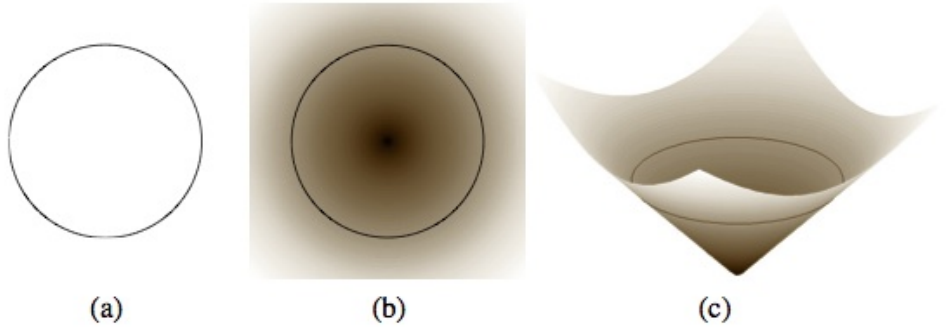

Figura 3.7: Ejemplo de cómo embeber una curva a modo de *level set*. (a) Contorno arbitrario en forma de círculo. (b) Representación de una función *level set* donde el círculo es el conjunto de nivel cero. (c) Mapa de alturas de dicha función.

define en el mismo dominio que la imagen. El conjunto de nivel se define como el subconjunto de puntos para los cuales la función toma el mismo valor. El único propósito de la función *level set* es el de proveer una representación implícita de la curva envolvente.

En lugar de monitorizar la curva a través del tiempo, el método *level set* actualiza los valores de la función en coordenadas fijas a través del tiempo. La función *level set* permanece como función matemática válida aún si la curva embebida cambia su topología, como por ejemplo dividiéndose en varias curvas cerradas. Llamando $\phi(x, y)$ a la función *level set,* su conjunto de nivel cero será

$$Z = \{(x, y) \,|\, \phi(x, y) = 0\} \tag{3.3.6}$$

donde para $\phi(x, y)$ se asume que tomará valores positivos en puntos del interior de la curva y negativos en puntos del exterior. Si consideramos Z como una curva dinámica que se deforma según su dirección normal a una velocidad v decimos que $\phi(x, y)$ satisface la ecuación *level set*

$$\frac{\partial \phi}{\partial t} = v|\nabla \phi| \tag{3.3.7}$$

la cual se trata de una ecuación en derivadas parciales –de tipo *Hamilton-Jacobi*– que puede resolverse mediante métodos numéricos, como las diferencias finitas, aunque en general su solución requerirá de técnicas sofisticadas.

Capítulo 4

Descripción de la aplicación

La finalidad de la aplicación es la segmentación automática de estructuras en imágenes médicas en formato *DICOM*[1]. Particularmente, tratamos el caso de la segmentación del hígado en imágenes generadas mediante tomografía axial computarizada.

Se espera como entrada un directorio conteniendo un conjunto de imágenes de iguales dimensiones y de una serie de parámetros numéricos que repercuten en operadores de diferente índole. Como salida se generará un nuevo directorio que contenga las máscaras binarias –tanto en formato imagen como en formato texto– por separado correspondientes a cada una de las imágenes de entrada. Adicionalmente, se podrá generar una máscara «maestra» mediante la combinación de varias de estas máscaras individuales.

4.1. Tecnologías

Nuestro programa está desarrollado en lenguaje *C++*, usando las librerías *Insight Registration and Segmentation Toolkit (ITK)* como recurso algorítmico de segmentación de imágenes, las librerías *Boost* para el manejo del sistema de ficheros y el sistema de compilación multiplataforma *CMake*. A continuación explicaremos las distintas soluciones existentes y los motivos por los que hemos decidido hacer uso de estas bibliotecas.

4.1.1. Bibliotecas de procesamiento de imágenes médicas

A día de hoy, algunas de las bibliotecas de procesamiento de imágenes más importantes son las siguientes:

- ***MathWorks Matlab: Image Processing.*** *Matlab* es uno de los programas más utilizados en el ámbito científico y de investigación por sus numerosas bibliotecas, sus herramientas de simulación y sobre todo por su facilidad de uso. Sin embargo, su rendimiento computacional es bajo, llegando a resultar demasiado lento para determinadas operaciones complejas.
- ***OpenCV Library.*** *OpenCV* es un conjunto de bibliotecas de código abierto de procesamiento de imágenes de ámbito general, desarrolladas en *C++*. Pese

[1] *DICOM (**D**igital **I**maging and **CO**mmunication in **M**edicine),* es un estándar en el campo de la imagen médica. Incluye una definición de formato de imagen y un protocolo de comunicaciones de red (`http://medical.nema.org/standard.html`)

a ser un proyecto muy completo, no está diseñado específicamente en el marco de imagen médica y su documentación no es muy rigurosa y organizada.

- ***MIPAV.*** Se trata de una aplicación con interfaz gráfica de procesamiento de imágenes médicas de uso libre creada en el *Center for Information Technology* de Maryland. Está desarrollada en lenguaje *Java*, y permite el análisis y la aplicación de algoritmos de procesamiento y segmentación a nivel de usuario. Existe una *API* para desarrolladores, aunque su uso es bastante complicado y el rendimiento respecto a bibliotecas desarrolladas en *C++* es inferior.

- ***Matrox Imaging Library.*** *MIL* es otro importante toolkit de carácter comercial para el procesado de imágenes, pero funciona sólo bajo sistemas *Windows.*

En el siguiente apartado se discute *ITK,* que será nuestra elección para el presente proyecto.

4.1.2. *Insight Segmentation and Registration Toolkit*

El *Insight Toolkit (ITK)* es un conjunto de herramientas de software de código abierto para el registro y segmentación de imágenes o conjuntos de datos digitales muestreados en general. *ITK* está implementado en *C++* y utiliza el entorno de compilación *CMake,* lo que permite un desarrollo cómodo que es independiente del sistema operativo. Además, proporciona un proceso automático de encapsulado (llamado *Cable)* que genera interfaces entre entre *C++* y algunos de los lenguajes interpretados más populares como *Java, Python* y *Tcl,* posibilitando el desarrollo de aplicaciones mediante múltiples lenguajes de programación. El paradigma de implementación de *ITK* es el de programación genérica, el cual hace uso de plantillas (o *templates*) por lo que el mismo código puede ser aplicado genéricamente a cualquier clase o tipo de dato que soporte las operaciones usadas por la plantilla. El uso de plantillas en *C++* hace el código muy eficiente, permitiéndonos descubrir los problemas en tiempo de compilación en lugar de en tiempo de ejecución.

Como *ITK* es un proyecto de código abierto, desarrolladores de todo el mundo pueden usar, depurar, mantener y ampliar el software.

El *Insight Tollkit* está compuesto de varios subsistemas. Seguidamente se exponen brevemente los más importantes:

- **Conceptos elementales del sistema.** Como cualquier otro sistema de software, *ITK* está construido a partir de algunos conceptos elementales de diseño. Algunos de ellos son el de programación genérica, los punteros inteligentes para gestión de memoria, fábricas de objetos que permiten una instanciación adaptable de objetos, gestión de eventos y soporte multihilo (*multithreading*).

- **Métodos numéricos.** *ITK* utiliza las librerías numéricas *VNL,* las cuales son encapsuladores *C++* sobre las rutinas de análisis numérico *Netlib Fortran* (`http://www.netlib.org`).

- **Acceso y representación de datos.** Existen dos clases principales para la representación de datos: las clases $itk :: Image$ e $itk :: Mesh$. Adicionalmente, varios tipos de iteradores y contenedores están disponibles para la persistencia y la manipulación de los datos.

- ***Pipeline* de procesamiento de datos.** Las clases para la representación de datos –conocidas como *objetos de datos*– son manipuladas por filtros, los cuales a su vez pueden ser organizados en *pipelines* de flujo de datos. Estos *pipelines* mantienen el estado y por tanto se ejecutan únicamente cuando son necesarios, soportando además el uso de multi-hilo y *streaming*.

- ***Framework* de entrada/salida.** Asociados junto con el *pipeline* de procesamiento de datos están los *sources*, que son filtros encargados de iniciar el *pipeline*, y los *mappers*, que son filtros que finalizan el *pipeline*. Los ejemplos más claros de *sources* y *mappers* son los *readers* y *writers* respectivamente. Los *readers* leen datos –generalmente desde un archivo– y los *writers* se encargan de entregar los datos a la salida.

- **Objetos Espaciales.** En *ITK*, las formas geométricas se representan mediante la jerarquía de objetos espaciales. Usando una interfaz básica común, los objetos espaciales son capaces de representar regiones del espacio de diferentes maneras, como por ejemplo mediante las estructuras en *malla*, las *máscaras* de imagen o las ecuaciones implícitas.

- ***Level Set Framework*.** El framework para *level set* consiste en un conjunto de clases para crear filtros que resuelvan ecuaciones en derivadas parciales en imágenes usando métodos iterativos de diferencias finitas. Específicamente, éste contiene un filtro genérico de segmentación mediante *level set* y diferentes subclases útiles como filtros de valor umbral y operadores (*Laplace, Canny, etc.)*.

4.1.3. Otras bibliotecas

4.1.3.1. *Boost*

Boost es un paquete de librerías modernas basadas en el estándar de *C++*. Su código fuente descansa sobre la *Boost Software License*, la cual permite usar, modificar y distribuir gratuitamente las librerías. Éstas son multiplataforma, y soportan tanto los compiladores más conocidos como algunos menos populares.

Usar las librerías *Boost* es una de las mejores opciones para incrementar la productividad en proyectos C++ una vez explotados los recursos del estándar. Al estar basadas en la última revisión del estándar de C++, pueden ser usadas muchas de las utilidades basadas en el progreso actual que aún no han sido «oficialmente» aprobadas.

Boost contiene más de 90 librerías, por lo que aquí comentaremos sólo un extracto de las más relevantes:

- **Boost.Any.** Proporciona el tipo de dato *any*, que permite el almacenamiento de tipos de dato arbitrarios.

- **Boost.Array.** Permite el tratamiento de *arrays* como *containers*.

- **Boost.Asio.** Permite el desarrollo de aplicaciones que procesen los datos de forma asíncrona (como las aplicaciones de red).

- **Boost.Bimap.** Similar a *map* de la *STD*, pero permitiendo buscar tanto por clave como por valor.

- **Boost.Conversion.** Proporciona tres operadores de *casting* para realizar *downcasts, cross casts* y para la conversión entre valores de diferentes tipos numéricos.
- **Boost.DateTime.** Procesa, lee y escribe fechas y horas usando un formato flexible.
- **Boost.Exception.** Permite que la excepción lance datos adicionales para proveer más información al *catch handler.*
- **Boost.Filesystem.** Proporciona clases para el procesamiento de la información en las rutas además de contener varias funciones para el acceso a archivos y directorios.
- **Boost.Lambda.** Permite la definición de funciones anónimas.
- **Boost.Multiindex.** Permite la definición de nuevos contenedores que soporten múltiples interfaces como las propias de *vector* y *map* en la *STD.*
- **Boost.Regex.** Funciones para la búsqueda de texto mediante expresiones regulares.
- **Boost.Serialization.** Gracias a esta librería, objetos pueden ser serializados y, por ejemplo, ser guardados en archivos para su uso posterior.
- **Boost.Thread.** Permite el desarrollo de aplicaciones *multithread.*

4.1.3.2. *CMake*

CMake es un sistema extensible de código abierto que gestiona el proceso de compilado en el sistema operativo de manera independiente de la plataforma. A diferencia de otros sistemas de compilación multiplataforma, *CMake* está diseñado para ser usado junto con el entorno de compilación nativo. En los directorios contenedores de código fuente se encuentran los archivos de configuración de *CMake* –llamados *CMakeLists.txt*– que se usan para generar los archivos de compilación estándar –como lo son los *Makefile* en Unix, por ejemplo–, los cuales son usados como lo haríamos normalmente.

Cuando *CMake* se ejecuta, localiza archivos especificados en los *include*, librerías y ejecutables y puede decidir directivas opcionales de compilación. Toda esta información se guarda en un archivo a forma de *cache*, y puede ser editada por el usuario antes de la generación de los archivos nativos de compilación.

CMake está diseñado para soportar jerarquías de directorios complejas y aplicaciones que dependen de varias librerías. Como ejemplo, soporta proyectos dependientes de varias librerías –o *toolkits*–, donde cada una de ellas puede contener a su vez varios directorios y la aplicación puede depender además de otros archivos adicionales externos. También es capaz de manejar situaciones en las que sea necesario crear ejecutables para generar el código que finalmente se compilará y enlazará como parte de la aplicación.

Usarlo es sencillo: la compilación se controla creando uno o más archivos *CMakeLists.txt* en cada directorio que constituya un proyecto. Cada archivo *CMakeLists.txt* consiste en varios comandos con la forma

```
COMANDO (<lista de argumentos separados por espacios>)
```

Se proporcionan varios comandos predefinidos, pero es posible añadir nuestros propios comandos personalizados en caso de ser necesario. En el capítulo sobre instalación y configuración del software veremos esto con más detalle.

CMake fue creado como respuesta a la necesidad de tener un sistema de compilación multiplataforma potente para el *Insight Toolkit.*

4.2. Aplicación de segmentación

4.2.1. Introducción

La aplicación de segmentación, teniendo una estructura muy simple, consta principalmente de un *pipeline* de filtros de preprocesado y del algoritmo de segmentación basado en el trabajo «*Geodesic Active Contours*» de *Caselles et al.,* el cual está referido en la bibliografía por exceder el alcance de este documento.

Una segunda aplicación implementada por separado se encarga de combinar los *pixels* de interés de las máscaras resultado en caso de que la segmentación se aplique sobre un grupo de imágenes en bloque, lo cual mejora considerablemente el resultado en términos generales.

Se discute a continuación de forma programática el código de las aplicaciones intentando definir con más detalle las librerías y métodos más relevantes. Hay que tener presente que el estilo de programación es el basado en plantillas, por lo que los lectores con experiencia previa en este paradigma encontrarán muchos conceptos familiares.

4.2.2. Pruebas previas: selección de un método adecuado

Como partimos de un conjunto de imágenes muy heterogéneas desde el punto de vista del contraste e intensidad de las regiones –ver figura 4.1–, era de esperar que los algoritmos de segmentación basados en métodos sencillos como el crecimiento de regiones no funcionaran correctamente. Pese a esto, empezamos realizando pruebas con algoritmos sencillos y fuimos aumentando progresivamente la complejidad.

Las pruebas se realizaron tanto de forma gráfica con la aplicación *MIPAV* como con la biblioteca *ITK,* creando pequeñas aplicaciones de test. A continuación se comentan los métodos utilizados y un pequeño resumen de las conclusiones:

- *Thresholding.* La segmentación mediante nivel umbral no es posible debido a que la zona del hígado tiene unos valores de intensidad que coinciden con muchas otras zonas de la imagen. Las pruebas se realizaron con *MIPAV.*

- *Crecimiento de regiones.* Los métodos basados en regiones arrojaron buenos resultados en algunas de las imágenes, pero en general se descartó su uso debido a la falta de solidez para segmentar el conjunto completo de imágenes. Se probaron los métodos *connected threshold* –que tiene en cuenta el valor de intensidad de los *pixels* vecinos respecto a un *pixel* central– y *neighborhood connected* –que compara los valores no sólo con un *pixel* sino de un conjunto de ellos– de la biblioteca *ITK.*

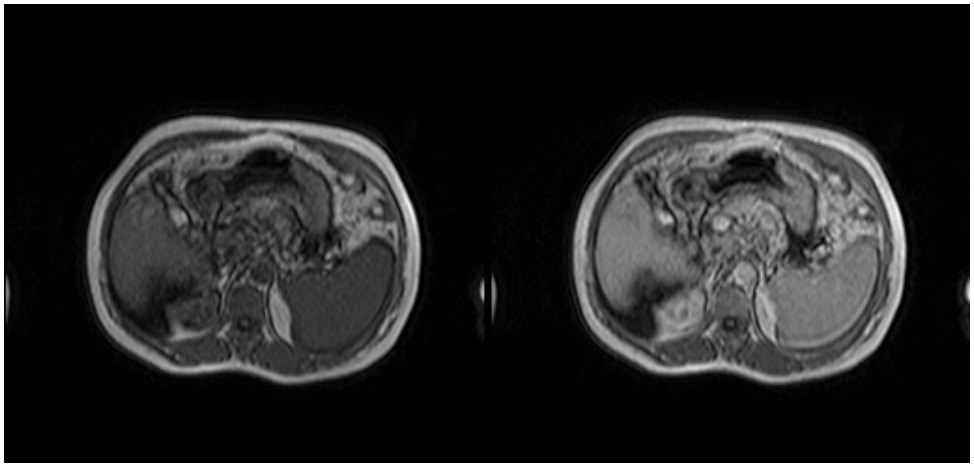

Figura 4.1: Las imágenes adquiridas mediante la tomografía axial presentan una importante variabilidad en el contraste y la intensidad.

- *Level Set (genérico).* En este caso obtuvimos la misma conclusión que con los métodos de regiones, aunque con unos resultados un tanto más precisos. Sin embargo, la poca fiabilidad para segmentar la totalidad de las imágenes llevaron también a su descarte. Se hicieron diferentes tests con los algoritmos disponibles en *MIPAV.*

- *Geodesic Active Contours.* Usando algoritmos que combinan *level sets* con contornos activos los resultados pasaron a ser sorprendentemente estables, y en especial el algoritmo basado en el trabajo de *Casselles et al.* bajo la implementación sugerida en la documentación de *ITK.* Más adelante veremos ejemplos gráficos de los resultados.

4.2.3. Librerías *ITK* utilizadas

Las siguientes librerías del *Insight Toolkit* se incluyen en la cabecera del programa:

- `itk::Image<TPixel,VImageDimension>`

  Clase que representa un objeto imagen n-dimensional. La plantilla se construye sobre un tipo de *pixel* y un valor de dimensión. Una imagen se modela como un *array,* definido por un índice de comienzo y un tamaño. Es posible acceder y/o modificar el valor de los *pixels* mediante iteradores, usando las funciones *SetPixel() y GetPixel().*

- `itk::ImageFileReader<TOutputImage,ConvertPixelTraits>`

  Objeto de tipo origen de datos, que lee datos de una imagen desde un único archivo. En realidad este objeto es un filtro general capaz de extraer datos desde varios formatos de imagen. *TOutputImage* es el tipo de imagen que esperamos leer, y aunque este tipo sea diferente al del archivo que hemos abierto, este automáticamente es convertido al especificado por *TOutputImage*, en caso de ser posible. Normalmente, sólo será necesario especificar un nombre de archivo con la extensión correcta en el momento de instanciar el objeto.

- `itk::ImageFileWriter<TInputImage>`

  Escribe los datos de una imagen en un archivo. Al igual que en el caso anterior, sólo será necesario especificar un nombre de archivo con extensión para que el *writer* escriba el formato correctamente.

- `itk::ImageRegionIteratorWithIndex<TImage>`

  Iterador multidimensional cuya plantilla se construye sobre un tipo de imagen. Este iterador recorre la imagen por filas mediante el operador $++$, guardando la información de posición del índice. Tiene permiso tanto de lectura como de escritura, mediante los métodos *Get()* y *Set()* respectivamente.

- `itk::BinaryThresholdImageFilter<TInputImage, TOutputImage>`

  Convierte la imagen de entrada en una imagen binaria, mediante *thresholding*. La plantilla se construye sobre un tipo de imagen de entrada y un tipo de imagen de salida, y se espera que estas imágenes tengan el mismo número de dimensiones. Los pixels de la imagen de salida podrán tener sólo dos valores, especificados por *InsideValue* y *OutsideValue*. Los niveles umbral los definen *UpperThreshold* y *LowerThreshold*, teniendo que definir obligatoriamente al menos uno de ellos.

- `itk::RescaleIntensityImageFilter<TInputImage, TOutputImage>`

  Realiza un reescalado de intensidad, aplicando la siguiente transformación lineal a los valores de intensidad de los *pixels* de la imagen de entrada:

$$Pixel_{out} = (Pixel_{in} - Min_{in}) \cdot \frac{(Max_{out} - Min_{out})}{(Max_{in} - Min_{in})} + Min_{out} \qquad (4.2.1)$$

  donde Max y Min representan los valores máximos y mínimos de intensidad presentes en las imagenes de entrada o salida.

- `itk::CurvatureAnisotropicDiffusionImageFilter <TInputImage, TOutputImage>`

  Los métodos de difusión anisotrópica se formulan para reducir el ruido –o los detalles no deseados– en imágenes pero preservando algunas de sus características específicas. En general, estos métodos asumen que las transiciones entre luz y oscuridad en la imagen son interesantes, y a diferencia de los métodos isotrópicos, consiguen la reducción sin afectar a la visibilidad de los contornos. Específicamente, este filtro realiza la difusión anisotrópica en una imagen escalar mediante el uso de la ecuación modificada de difusión de curvatura (*MCDE*):

$$f_t = |\nabla f| \nabla \cdot c(|\nabla f|) \frac{\nabla f}{|\nabla f|} \qquad (4.2.2)$$

- `itk::GradientMagnitudeRecursiveGaussianImageFilter <TInputImage,TOutputImage>`

Realiza el cálculo de la magnitud del gradiente en una imagen o en una región de una imagen, mediante la convolución *IIR* con la primera derivada de un *kernel* de *Gauss (aproximación):*

$$\frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{x^2}{2\sigma^2}\right) \tag{4.2.3}$$

- `itk::SigmoidImageFilter<TInputImage, TOutputImage>`

La *función sigmoide,*

$$f(x) = \frac{1}{1+e^{-x}} \tag{4.2.4}$$

que aparece en gran cantidad de procesos en la ciencia, se usa en este filtro para suavizar la visibilidad de los contornos mediante una transformación del rango de intensidades presentes en la imagen. En particular, este filtro aplica sobre los contornos calculados la siguiente modificación de la anterior fórmula:

$$f(x) = (Max - Min) \cdot \frac{1}{\left(1+e^{-\frac{x-\beta}{\alpha}}\right)} + Min \tag{4.2.5}$$

donde x es un determinado *pixel* de entrada, $f(x)$ representa un *pixel* de salida y α y β son constantes proporcionadas por el usuario.

- `itk::GeodesicActiveContourLevelSetImageFilter <TInputImage, TFeatureImage,TOutputPixelType>`

Los contornos activos geodésicos pueden considerarse una subclase del método genérico *level set.* Se usan para la segmentación de estructuras en una imagen basándose en una máscara de contorno, digamos $g(I)$, el cual tiene valores cercanos al 0 cerca de los bordes y valores cercanos al 1 en zonas de intensidad aproximadamente uniforme. Normalmente, la máscara de contorno es una función del gradiente, como por ejemplo:

$$g(I) = exp^{-|(\nabla * G)\cdot I|} \tag{4.2.6}$$

Donde I representa el valor de intensidad y G el operador de *Gauss.* La otra entrada que requiere el método es un *level set* como punto de partida, el cual se trata de una imagen real que contiene el contorno inicial como el conjunto de nivel cero, o *zero level set.* En este método, el término de advección adicional,

$$A(x) = -\nabla g(x) \tag{4.2.7}$$

se encarga de atraer este contorno inicial hacia los bordes reales, siendo $g(x)$ la imagen ya procesada que contiene la informacion de contornos. Este procedimiento está basado en el trabajo «Geodesic Active Contours» de *Caselles et al.*

- `itk::FastMarchingImageFilter<TLevelSet, TSpeedImage>`

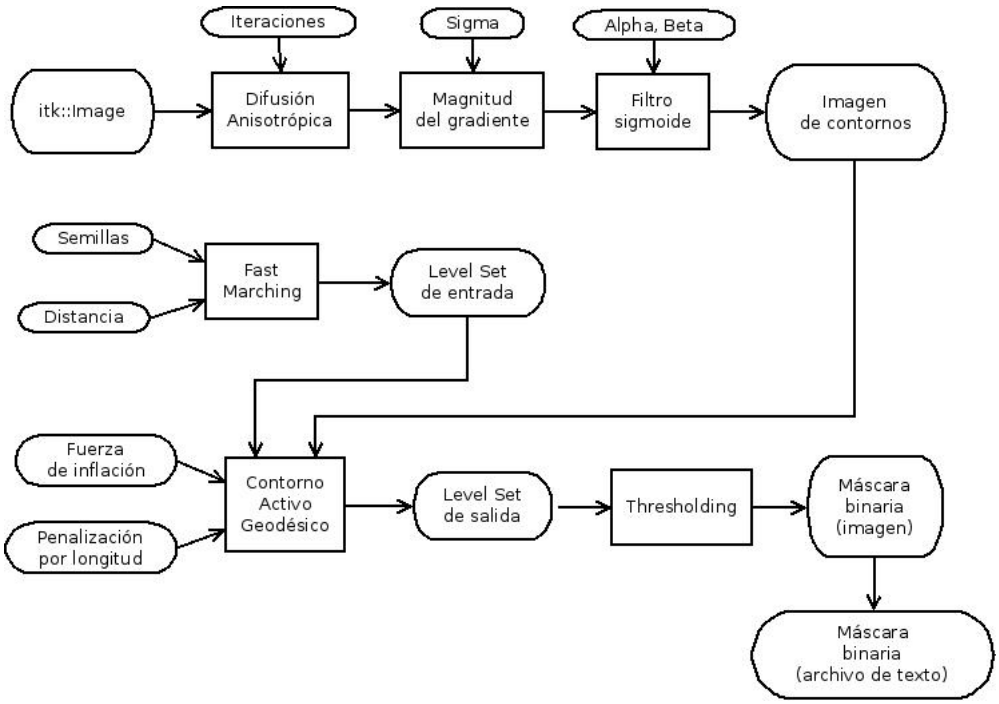

Figura 4.2: Diagrama de flujo de datos de la aplicación de segmentación.

Los métodos *fast marching* se utilizan para resolver el problema de propagación de frentes. Se desarrollaron gracias al trabajo de *J.A. Sethian «Level Set Methods and Fast Marching Methods»*, e involucran el cálculo iterativo de nuevas posiciones para una curva dada cuando sobre esta se aplica una fuerza F. En este caso particular, el filtro *fast marching* resuelve una ecuación *eikonal* donde la velocidad se presupone siempre positiva y dependiente únicamente de la posición.

4.2.4. Análisis del código fuente

La manera más sencilla de introducirse en el algoritmo es inspeccionando el diagrama de la figura 4.2, el cual pasamos a explicar a continuación mostrando el código implicado. Empezaremos con las declaraciones de tipos de imagen, tanto de entrada como de salida:

```
const unsigned int Dimension = 2;
typedef float InputPixelType;
typedef unsigned char OutputPixelType;

typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

El formato de imagen de entrada principal que la aplicación debe soportar es *DICOM*, por lo que el tipo de *pixel* de entrada será *float,* aunque *DICOM* soporta otros tipos de datos como *char* o *short*. Como tipo de *pixel* de salida hemos escogido *unsigned char* puesto que nos permitirá guardar las imágenes de salida tanto en *DICOM* como en otros formatos más fáciles de visualizar sin software específico, como *JPEG* o *PNG*.

Como ya hemos comentado en la sección anterior, en *ITK* las entradas y salidas de datos en formato fichero pasan por un operador que realiza un *casting* implícito según la extensión que decidamos asignarles. Veamos ahora las instanciaciones del lector y el escritor de ficheros:

```
typedef itk::ImageFileReader< InputImageType > ReaderType;
typedef itk::ImageFileWriter< OutputImageType > WriterType;

typedef itk::GDCMImageIO ImageIOType;
ImageIOType::Pointer gdcmImageIO = ImageIOType::New();

ReaderType::Pointer reader = ReaderType::New();
WriterType::Pointer writer = WriterType::New();
```

La declaración del tipo *GDCMImageIO* es un requerimiento ya que los ficheros en formato *DICOM* contienen una cabecera especial. Más adelante aplicaremos este tipo de entrada/salida tanto al *reader* como al *writer* mediante el método *SetImageIO()*. A continuación, procedemos a la declaración e instanciación del filtro de suavizado de difusión anisotrópica, que mejorará la imagen en términos de ruido digital:

```
typedef itk::CurvatureAnisotropicDiffusionImageFilter< InputImageType,
InputImageType > SmoothingFilterType;

SmoothingFilterType::Pointer smoothing = SmoothingFilterType::New();
```

Este filtro necesita la definición de una serie de parámetros de entrada, los cuales serán introducidos mediante la línea de comandos para una mayor personalización por parte del usuario, aunque con los valores típicos tradicionales se podrán resolver la mayor parte de los casos:

```
const double TimeStep = atof( argv[4] );
const int NumberOfIterations = atoi( argv[5] );
const double ConductanceParameter = atof( argv[6] );

smoothing->SetTimeStep( TimeStep );
smoothing->SetNumberOfIterations( NumberOfIterations );
smoothing->SetConductanceParameter( ConductanceParameter );
```

El parámetro *TimeStep* se corresponde exactamente con $\triangle t$ en la ecuación usada por *ITK* para el cálculo por aproximación mediante diferencias finitas. Un valor típico estable para este parámetro es 0,125, aunque podremos probar otros valores siempre que sean inferiores a 0,250. En este enlace hacia la documentación se encuentra información detallada.

Por otro lado, el *ConductanceParameter* controla la sensibilidad del término de conductancia en la ecuación básica de difusión anisotrópica. Sus valores típicos oscilan entre 0,5 y 2, siendo mayor el suavizado y el efecto borroso cuanto mayor sea este parámetro. Para una mayor preservación de la información en la imagen deberemos usar valores pequeños. Podemos encontrar más detalles en este enlace.

El siguiente paso es la instanciación del operador que realiza el cálculo por convolución de la magnitud del gradiente de la imagen, lo cual arrojará una imagen binaria cuya única información es la de los contornos aproximados de las estructuras presentes en ella:

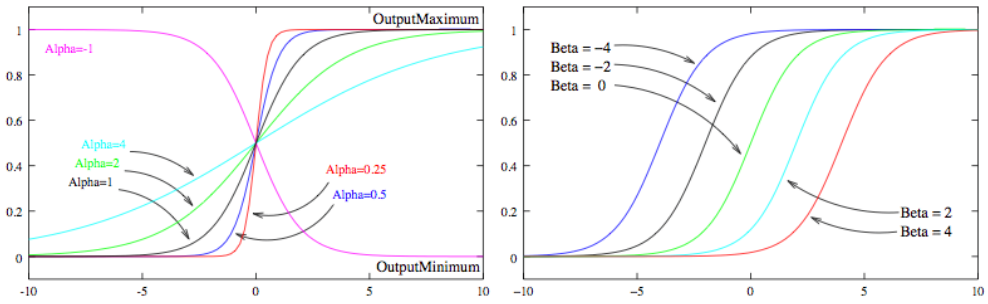

Figura 4.3: Izquierda y derecha respecivamente: parámetros α y β del filtro *sigmoide*.

```
typedef itk::GradientMagnitudeRecursiveGaussianImageFilter
         < InputImageType, InputImageType > GradientFilterType;

GradientFilterType::Pointer gradientMagnitude =
                                    GradientFilterType::New();
```

El parámetro σ de la expresión *gaussiana* –que tiene su influencia en la sensibilidad al ruido presente en la imagen– será tomado de la línia de comandos y aplicado al operador mediante el método *SetSigma()*, de forma análoga al caso anterior:

```
const double sigma = atof( argv[7] );

gradientMagnitude->SetSigma( sigma );
```

Tradicionalmente una imagen cruda de contornos se procesa mediante un filtro de tipo *sigmoide* con la finalidad de obtener un mayor contraste y resaltar la información útil. Seguidamente en la aplicación se encuentra la instanciación de este filtro:

```
typedef itk::SigmoidImageFilter< InputImageType,
                                 InputImageType > SigmoidFilterType;

SigmoidFilterType::Pointer sigmoid = SigmoidFilterType::New();
```

A lo que siguen las definiciones de sus parámetros α y β:

```
const double alpha = atof( argv[8] );
const double beta = atof( argv[9] );

sigmoid->SetAlpha( alpha );
sigmoid->SetBeta( beta );
```

En el filtro *sigmoide*, el parámetro α especifica la amplitud de la ventana de intensidad mientras que el parámetro β define su centro. La figura 4.3 representa gráficamente estos conceptos.

Hasta este punto, tenemos instanciados todos los operadores que nos permitirán obtener una de las dos entradas necesarias para el cálculo de la segmentación: la imagen de contornos. El segundo parámetro necesario es un *level set* inicial, obtenido a partir de un mapa de distancias generado por el operador de *fast marching*. Las siguientes línias corresponden a su instanciación:

```
typedef itk::FastMarchingImageFilter< InputImageType,
                                      InputImageType > FastMarchingFilterType;

FastMarchingFilterType::Pointer fastMarching =
                                      FastMarchingFilterType::New();
```

El filtro de *fast marching* necesita dos parámetros de entrada: la posición de partida o semilla –que deberá encontrarse dentro del hígado– y un valor de distancia inicial, que deberá ser aproximadamente la distancia que existe entre la semilla y el contorno del hígado. Estas dos variables son pasadas al filtro de *fast marching* mediante una variable de pila especial, llamada *NodeContainer:*

```
typedef FastMarchingFilterType::NodeContainer NodeContainer;

NodeContainer::Pointer seeds = NodeContainer::New();
```

Una vez obtenido el puntero *seeds* de tipo *NodeContainer*, crearemos un objeto de tipo *Index* para almacenar las coordenadas x e y de la semilla:

```
InputImageType::IndexType seedPosition;

seedPosition[0] = atoi( argv[1] );
seedPosition[1] = atoi( argv[2] );
```

Ahora podemos almacenar la semilla junto con la distancia inicial –tomada de la línea de comandos– en una variable de tipo *Node:*

```
typedef FastMarchingFilterType::NodeType NodeType;
NodeType node;

const double initialDistance = atof( argv[3] );
const double seedRadius = - initialDistance;

node.SetValue( seedRadius );
node.SetIndex( seedPosition );
```

Al inicializar *seedRadius* como el negativo de la distancia inicial estamos haciendo que el método *fast marching* genere un *level set* de tamaño adecuado, ya que irá incrementando este valor hasta llegar a cero donde coincidirá con el contorno deseado. Finalmente, inicializamos el objeto *seeds* e insertamos dentro el *node,* pasándolo al filtro *fast marching* mediante el m*étodo SetTrialPoints():*

```
seeds->Initialize();
seeds->InsertElement( 0, node );

fastMarching->SetTrialPoints( seeds );
```

En este punto ya tenemos todo lo necesario para ejecutar el algoritmo principal de segmentación basado en contornos activos geodésicos. Como siempre, el primer paso es instanciar el operador:

```
typedef itk::GeodesicActiveContourLevelSetImageFilter<
    InputImageType, InputImageType > GeodesicActiveContourFilterType;

GeodesicActiveContourFilterType::Pointer geodesicActiveContour =
    GeodesicActiveContourFilterType::New();
```

Este operador usa tres parámetros diferentes para controlar la evolución del contorno, que serán tomados como argumentos desde la línea de comandos:

```
const double propagationScaling = atof( argv[10] );
const double curvatureScaling = atof( argv[11] );
const double advectionScaling = atof( argv[12] );

geodesicActiveContour->SetPropagationScaling( propagationScaling );
geodesicActiveContour->SetCurvatureScaling( curvatureScaling );
geodesicActiveContour->SetAdvectionScaling( advectionScaling );
```

El parámetro *PropagationScaling* deberá ser ajustado en los casos en los que el contorno sea especialmente borroso o de topología complicada, ya que controla el nivel de fuerza de expansión (la máscara será más amplia cuanto más grande sea este valor). Un buen valor por defecto para este parámetro es 2,71. Según el trabajo de *Casselles et al.* tanto el parámetro *CurvatureScaling* como *AdvectionScaling* pueden ser inicializados a 1,0 en la mayoría de los casos. Cabe resaltar que cuanto mayor sea el valor de *CurvatureScaling* más «suave» será la máscara, aunque puede tener un efecto de contracción de su área final.

Existen otros dos parámetros a inicializar sobre el contorno activo geodésico, que tienen como función controlar sus criterios de finalización:

```
const double MaximumRMSError = atof( argv[13] );
const int MaximumIterations = atoi( argv[14] );

geodesicActiveContour->SetMaximumRMSError( MaximumRMSError );
geodesicActiveContour->SetNumberOfIterations( MaximumIterations );
```

Existen por tanto dos criterios de finalización: la ejecución se detendrá bien cuando el error sea inferior a un determinado umbral o bien cuando se llegue al número máximo de iteraciones. Ambos parámetros son pasados como argumentos desde la línea de comandos, pero habría que considerar como buenos los valores por defecto de 0,015 para *MaximumRMSError* y 1000 para *MaximumIterations*. Un valor inferior para el error o un mayor número máximo de iteraciones arrojarán un resultado más exacto en la mayoría de los casos pero hay que tener en cuenta que el coste computacional será mucho más alto.

Una vez conseguida la máscara que representa la segmentación aproximada del hígado el siguiente paso será pasarla a través de un filtro que realice el proceso de *thresholding* para que podamos obtener en la salida una imagen puramente binaria –esto es, con *pixels* de valor 0 o 255 únicamente–. En el caso del operador de *thresholding* los parámetros son estáticos y se aplican directamente sobre el código fuente:

```
typedef itk::BinaryThresholdImageFilter< InputImageType,
                      OutputImageType > ThresholdingFilterType;
ThresholdingFilterType::Pointer thresholder =
                      ThresholdingFilterType::New();

thresholder->SetLowerThreshold( -1000.0 );
thresholder->SetUpperThreshold( 0.0 );
thresholder->SetOutsideValue( 0 );
thresholder->SetInsideValue( 255 );
```

Como opción adicional se ha añadido un filtro que da como salida únicamente el contorno de la máscara, siendo por tanto el resultado una curva cerrada en lugar de

una figura sólida. Este filtro se introduce muy fácilmente en el flujo del programa (ver indicaciones en el código fuente):

```
typedef itk::BinaryContourImageFilter<OutputImageType,
                        OutputImageType> BinaryContourFilterType;
BinaryContourFilterType::Pointer binaryContour =
                        BinaryContourFilterType::New();

binaryContour->SetBackgroundValue( 0 );
binaryContour->SetForegroundValue( 255 );
binaryContour->SetFullyConnected( 0 );
```

Lo siguiente que encontramos en el código fuente es una de sus partes más importantes, que es la formación del *pipeline*. Éste es un reflejo exacto del diagrama de la figura 4.2, y en él se define el camino que siguen los datos de la imagen a través de los filtros y operadores:

```
smoothing->SetInput( reader->GetOutput() );

gradientMagnitude->SetInput( smoothing->GetOutput() );

sigmoid->SetInput( gradientMagnitude->GetOutput() );

geodesicActiveContour->SetInput( fastMarching->GetOutput() );

geodesicActiveContour->SetFeatureImage( sigmoid->GetOutput() );

thresholder->SetInput( geodesicActiveContour->GetOutput() );

writer->SetInput( thresholder->GetOutput() );
```

Sin embargo, en este punto aún no se ha llamado al método que dispara su ejecución, que en *ITK* se trata del método *Update()*. Al llamar a *Update()* desde la cola del *pipeline*, que en nuestro caso es el *writer*, provocará una consecución de llamadas en cadena siguiendo el flujo del diagrama 4.2. El motivo de no llamarlo todavía es porque tenemos que recorrer todos los archivos presentes en un determinado directorio de origen y aplicar la segmentación en cada uno de ellos. Entonces el *pipeline* tendrá que ser ejecutado dentro del bucle encargado de recorrer este directorio, en cada una de sus iteraciones.

En las consideraciones sobre sistemas de ficheros es cuando toma relevancia el uso de las librerías *Boost*. Empezaremos tomando las rutas deseadas de entrada y salida de ficheros desde la línea de comandos y guardándolas en variables de tipo *path* que la librería *Boost.Filesystem* nos proporciona:

```
path iparg( argv[15] );
path oparg( argv[16] );
```

Por otra parte, para generar el directorio de salida de forma sólida y que no se puedan dar ambigüedades o conflictos añadiremos en el nombre una marca de tiempo, de manera que se pueda ejecutar la aplicación con exactamente los mismos argumentos sin sobreescribir los resultados anteriores. Para esto también usamos una de las librerías *Boost*, en este caso *Boost.DateTime:*

```
ptime now = second_clock::local_time();
string timestamp = to_iso_string( now );
```

También nos queda por inicializar la extensión de las imágenes de salida que será el último de los argumentos que tomará la aplicación:

```
string oext = ( argv[17] );
```

Pese a que en principio sólo está concebido el trabajo con archivos *DICOM* este argumento puede resultar útil para una visualización más rápida y práctica de los resultados. Sin embargo, por mecanismos internos en el *casting* implícito que realizan los objetos *data source* de *ITK*, sólo podremos especificar las siguientes extensiones de salida: *dcm, jpg* y *png.*

A continuación se encuentra el bucle *for* responsable de recorrer cada uno de los ficheros del directorio de entrada, para lo cual también se ha utilizado un iterador especial de directorios de *Boost.* Se encuentra encapsulado dentro de una estructura *try/catch* que nos permite contemplar errores típicos en el manejo del sistema de ficheros, como por ejemplo rutas inexistentes debido a errores tipográficos, directorios no válidos, etc., además de la comprobación de que la ruta que hemos introducido apunta hacia un directorio y no un fichero. Seguidamente podemos ver su aspecto:

```
try {
    if ( exists( iparg ) )  {
        if ( is_regular_file( iparg ) ) {
            cout << iparg << "␣is␣not␣a␣directory." << endl;
        }
        else if ( is_directory( iparg ) )  {
            directory_iterator end_dirit;

for ( directory_iterator dirit( iparg ); dirit!=end_dirit; ++dirit ) {
. . .
. . .
```

El código que sigue a la anterior muestra representa una serie de consideraciones aburridas sobre la forma de manipular los nombres de los ficheros y las rutas para dotarlas de carácter multiplataforma, por lo que no es digno de mención. Existen unas líneas importantes que si comentaremos, en las que inicializamos los nombres de los archivos de entrada y salida en el *reader* y el *writer* mediante el método *SetFileName():*

```
reader->SetFileName( ipath_str );
writer->SetFileName( ofname1_str );
```

Los últimos pasos antes de empezar la ejecución del *pipeline* son los de comprobar que el archivo es un archivo bien formado, mediante el método *is_regular_file()* de *Boost* y aplicar el tipo de entrada/salida sobre el *reader* y el *writer* en caso de que el archivo sea *DICOM,* como habíamos anticipado anteriormente. Esto último se encontrará ya dentro de la otra estructura *try/catch* que contendrá las operaciones sobre los archivos de entrada:

```
if ( ipath.extension() == ".dcm" ||
     ipath.extension() == ".bmp" ||
     ipath.extension() == "" &&
     is_regular_file( dirit->path() ) )
{
 try
 {
 if ( ipath.extension() == ".dcm" ) reader->SetImageIO( gdcmImageIO );
```

```
 if ( oext == ".dcm" ) writer->SetImageIO( gdcmImageIO );
. . .
. . .
```

Podemos ver que también se añaden condiciones para las extensiones de entrada, lo cual es necesario para evitar un error puntual en el caso de que los ficheros de entrada carezcan de extensión en su nombre. Se añade como opción de entrada los ficheros *bitmap* por flexibilidad, pero la segmentación puede dar como resultado máscaras inusuales e inesperadas en formatos diferente a *DICOM.*

En este punto ya podemos iniciar la ejecución del *pipeline.* El filtro de *fast marching* requiere –por cuestiones del mecanismo interno– la especificación de las dimensiones del fichero de salida en cada nueva llamada que se realize. Esto a su vez requiere también una rellamada al filtro de *smoothing:*

```
smoothing->Update();

fastMarching->SetOutputSize( reader->GetOutput()->
                             GetBufferedRegion().GetSize() );

writer->Update();
```

A partir de aquí tenemos ya un archivo de físicamente guardado en el directorio de salida, por tanto podemos proceder a la generación del archivo de texto que contiene esta misma información de *pixels* en la forma de carácteres «0» y «1». Cada máscara en formato imagen tendrá su equivalente en formato texto con el mismo nombre de archivo salvo por la extensión –que será *txt*–.

Necesitamos crear primero un puntero que contenga esta máscara recientemente creada, pero en lugar de abrir el archivo del disco podremos inicializarlo desde la salida del último operador del *pipeline,* la cual se encontrará aún en memoria:

```
OutputImageType::Pointer regionImage = thresholder->GetOutput();

regionImage->Update();
```

Es un requerimiento de *ITK* que el puntero sea inicializado mediante el método *Update().*

Declararemos e instanciaremos también el iterador que recorrerá la máscara, para poder extraer la información del valor de intensidad de sus *pixels:*

```
typedef itk::ImageRegionIteratorWithIndex<
                                    OutputImageType > IteratorType;

IteratorType inputIt( regionImage,
                      regionImage->GetRequestedRegion() );
```

Declaramos y abrimos un fichero de texto vacío a continuación:

```
ofstream txtfile;
txtfile.open( ofname2_str.c_str() );
```

El bucle iterador realiza una operación sencilla: para cada *pixel* de la máscara que se encuentra en memoria, toma su valor de intensidad y escribe un «1» en el fichero de texto si su valor de es diferente de cero, en caso contrario, escribe un «0». El valor de intensidad es posible obtenerlo gracias a al método *Get()* disponible en *ITK* en los iteradores de imágenes:

```
for ( inputIt.GoToBegin(); !inputIt.IsAtEnd(); ++inputIt ) {
   float pixelValue = (float) inputIt.Get();

   if ( pixelValue != 0 ) txtfile << "1␣";
   else txtfile << "0␣";

   int imgwidth = reader->GetOutput()->
                                  GetBufferedRegion().GetSize()[0];

   if ( itPosition % imgwidth == 0 ) txtfile << "\n";
}

txtfile.close();
```

Las dos últimas líneas son debidas a que se introduce un salto de carro cuando se llega al final de las líneas de la imagen, puesto que los carácteres del archivo de texto son fieles a la posición de los *pixels* además de a sus valores de intensidad, evidentemente.

Aquí concluye la descripción de la aplicación de segmentación, donde sólo cabe comentar para finalizar las líneas relativas a la presentación de información útil de *debug* para el algoritmo de contornos activos geodésicos, que se muestran en la salida estándar al finalizar la segmentación de cada imagen:

```
cout << "File:␣" << ofname1_str
<< endl;

cout << "Size:␣" << reader->GetOutput()->GetBufferedRegion().GetSize()
<< endl;

cout << "Total␣iterations:␣"
<< geodesicActiveContour->GetElapsedIterations()
<< endl;

cout << "RMS␣deviation:␣"
<< geodesicActiveContour->GetRMSChange()
<< endl;
```

4.2.5. Resultados

En las figuras 4.4 y 4.5 podemos ver las máscaras binarias que hemos obtenido como resultado junto con sus correspondientes imágenes originales. Los valores de los parámetros usados para generar estos resultados son los siguientes:

Fast Marching	**Smoothing**	**Gradient mag.**	**Sigmoid**	**Geodesic Act. Con.**
SeedX = 145	TimeStep = 0,125	Sigma = 1,0	α= −0,5	Prop =2,71
SeedY = 250	Iterations = 5		β= 25,0	Curv = 1,0
Distance = 20	Conductance = 9,0			Adv = 1,0
				RMS = 0,015
				MaxIter = 1000

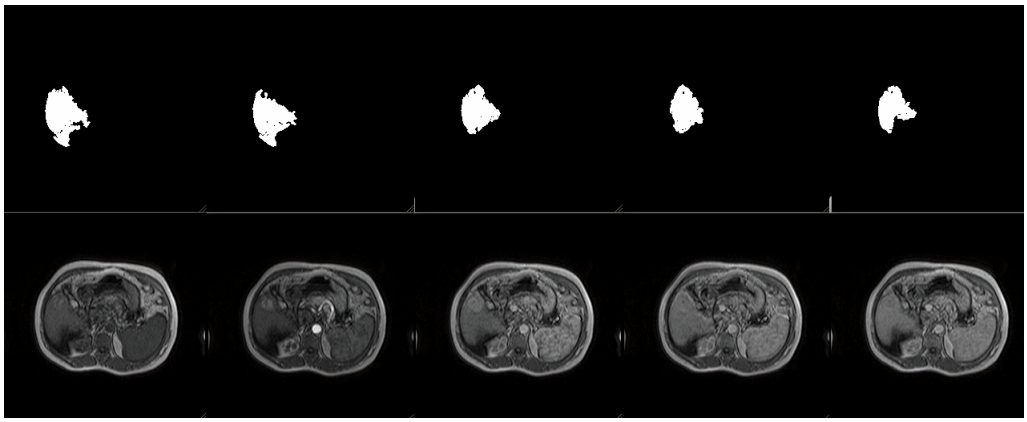

Figura 4.4: Resultados (1)

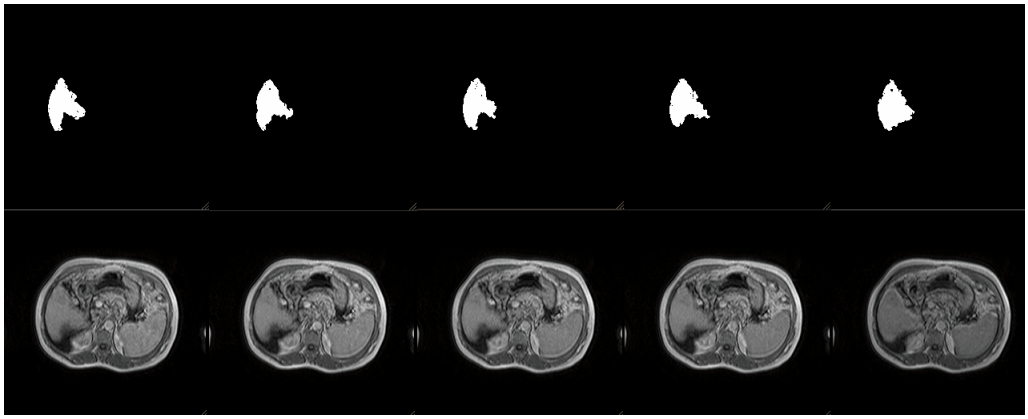

Figura 4.5: Resultados (2)

Figura 4.6: El resultado final de la segmentación es la suma de la información de todas las demás máscaras combinadas.

En la mayoría de los casos, deberemos realizar pruebas sucesivas mediante ensayo/error para obtener resultados satisfactorios, si bien es cierto que los parámetros por defecto indicados en la sección *4.2.4* y en la tabla anterior serán un buen comienzo y en general no habrá que modificarlos demasiado.

Analizando las imágenes puede apreciarse que individualmente las máscaras son de topología variable y no podremos utilizarlas como un resultado válido final de segmentación. Es por esto que deberemos combinar la información existente en el conjunto –sumando sus *pixels*–, como vemos en la figura 4.6. Esta máscara combinada representa con una aproximación muy alta el área del hígado, por lo que podemos considerar que la segmentación se ha realizado con éxito.

En la siguiente sección se explicará como usar esta segunda aplicación de combinado de *pixels* sobre el directorio de salida que hemos generado, además de todo el proceso de instalación y configuración del software.

Capítulo 5

Guía del usuario

El propósito de esta guía es el de orientar al usuario final en el uso y compilación del software sin requerirle conocimientos avanzados de programación o informática. Las instrucciones aquí presentes son genéricas y pueden aplicarse a cualquier sistema con tan sólo unas pequeñas diferencias.

5.1. Uso del software

Para ejecutar la aplicación sólo nos hará falta copiar en nuestro ordenador los binarios correspondientes a nuestro sistema operativo, que podemos encontrar en la ruta */software/bin* del *CD-ROM*, donde hay ejecutables para *Windows, Linux* y *Mac OS/X*.

Como la aplicación toma un gran número de parámetros, que ya hemos comentado en secciones anteriores, se proviene además de un pequeño *script* –llamado *launcher* o *launcher.bat*– que nos permitirá configurarlos fácilmente al mismo tiempo que nos servirá para lanzar la aplicación. Echémosle un vistazo:

```
#!/bin/sh
#
# A simple script to launch the segmentation app with custom params
#

## Fast marching node
SeedX='145'
SeedY='250'
InitialDistance='20'
## Smoothing
TimeStep='0.125'
NumberOfIterations='5'
ConductanceParameter='9.0'
## Gradient magnitude
Sigma='1.0'
## Sigmoid
SigmoidAlpha='-0.5'
SigmoidBeta='25.0'
## Geodesic active contour
PropagationScaling='2.71'
```

```
CurvatureScaling='1.0'
AdvectionScaling='1.0'
MaxRMSError='0.015'
MaxIterations='1000'
## I/O
InputPath='images/'
OutputPath='images/'
OutputFileExtension='.dcm'

./geodesicSeg $SeedX $SeedY $InitialDistance $TimeStep
$NumberOfIterations $ConductanceParameter $Sigma $SigmoidAlpha
$SigmoidBeta $PropagationScaling $CurvatureScaling
$AdvectionScaling $MaxRMSError $MaxIterations $InputPath
$OutputPath $OutputFileExtension
```

Como puede observarse los parámetros están en forma de variables que podemos editar como si de un archivo de texto normal se tratara. Normalmente, el parámetro que vamos a editar más a menudo será *InputPath,* es decir, el directorio de entrada donde se encuentran las imágenes *DICOM* que queremos segmentar. El parámetro *OutputPath* que representa el directorio de salida lo podremos dejar sin tocar una vez hayamos escogido uno adecuado, ya que los resultados no se sobreescribirán.

Dicho esto, y suponiendo que hemos editado el *script* con los parámetros deseados, sólo tendremos que abrir un terminal, acceder dentro del directorio donde hayamos copiado los binarios y ejecutar el *launcher*:

```
$ ./launcher
```

si nos encontramos en *Linux* o *Mac, o*

```
$ launcher.bat
```

en *Windows.*

Si todo ha ido correctamente, dentro del directorio de salida tendremos un subdirectorio con este aspecto,

```
binary-masks-20120623T111603/
```

que contiene las máscaras en formato imagen y en formato texto para cada fichero de entrada dado, respetando los nombres de archivo originales y conteniendo el sufijo *-mask.*

Ahora sólo nos falta ejecutar la utilidad de combinado de máscaras, *imageMerge*, que encontraremos junto con los demás archivos binarios. Esta aplicación toma los siguientes parámetros:

```
$ imageMerge <directorio_de_entrada> <anchura_imagen> <altura_imagen>
```

Entonces, suponiendo que las máscaras que hemos generado en el paso anterior tienen unas dimensiones de 512x512 y que el directorio donde se encuentran es el de arriba, escribiríamos:

```
./imageMerge binary-masks-20120623T111603/ 512 512
```

La máscara combinada se guardará automáticamente en ese directorio en formato *DICOM,* acompañada de su correspondiente máscara en formato texto, con los nombres *merged_mask.dcm* y *merged_mask.txt.*

5.2. Compilación del software

El único requisito para poder compilar las aplicaciones será tener instalado en nuestro sistema el paquete *GNU Compiler Collection (GCC)*, en su versión 4.6 o superior. También es posible hacerlo mediante otros compiladores, como el que contiene el *Visual C++* de *Microsoft*, pero no serían aplicables las instrucciones multiplataforma que aquí se encuentran. Por otro lado, para los comandos de terminal supondremos por comodidad que nos encontramos en un sistema *Unix*.

Los pasos a seguir son los siguientes:

1. Instalar *CMake*. En distribuciones *linux* se puede encontrar fácilmente mediante cualquier gestor de paquetes. Por ejemplo, en sistemas basados en *Debian* podemos escribir:

   ```
   $ sudo apt-get install cmake cmake-curses-gui
   ```

   En el caso de sistemas *Windows* o *Mac*, usaremos los instaladores que se encuentran en la ruta /software/resources/ del *CD-ROM*.

2. Extraer en nuestro *PC* el archivo *InsightToolkit-4.1.0.tar.gz* que se encuentra en la ruta */software/resources/* del *CD-ROM*. En lo que sigue supondremos que este directorio es */home/user/InsightToolkit-4.1.0*

3. Crear un directorio para alojar los archivos binarios de la biblioteca *ITK*. En lo que sigue supondremos que este directorio es */home/user/itk_build*

4. Mediante el terminal accedemos dentro del directorio anterior:

   ```
   $ cd /home/user/itk_build
   ```

5. Desde ahí teclearemos:

   ```
   $ ccmake /home/user/InsightToolkit-4.1.0/
   ```

6. En la pantalla de configuración de *CMake*, pulsaremos la tecla '*c*'

7. Veremos que aparecen varios parámetros; tendremos que dejar todo en *OFF* salvo las opciones *ITK_V3_COMP* y *ITK_BUILD_ALL*. Para modificar una opción bastará con mover el cursor hacia ella y pulsar *intro*, y seguidamente volver a pulsar *intro* para confirmarla.

8. Pulsar la tecla *'c'* de nuevo.

9. Pulsar *'g'*.

10. Ejecutar la orden *make* y esperar a que termine. Puede resutltar en un proceso de varios minutos.

    ```
    $ make
    ```

11. En este punto tendremos los archivos binarios necesarios de la bibilioteca *ITK*, pero necesitamos agregar en el *PATH* la ruta donde se encuentran. Para ello, nos desplazamos al directorio raíz del usuario del sistema,

    ```
    $ cd
    ```

    y editamos el archivo *.bashrc* (en *OS/X* este archivo se llama *.profile*), agregando la siguiente línea:

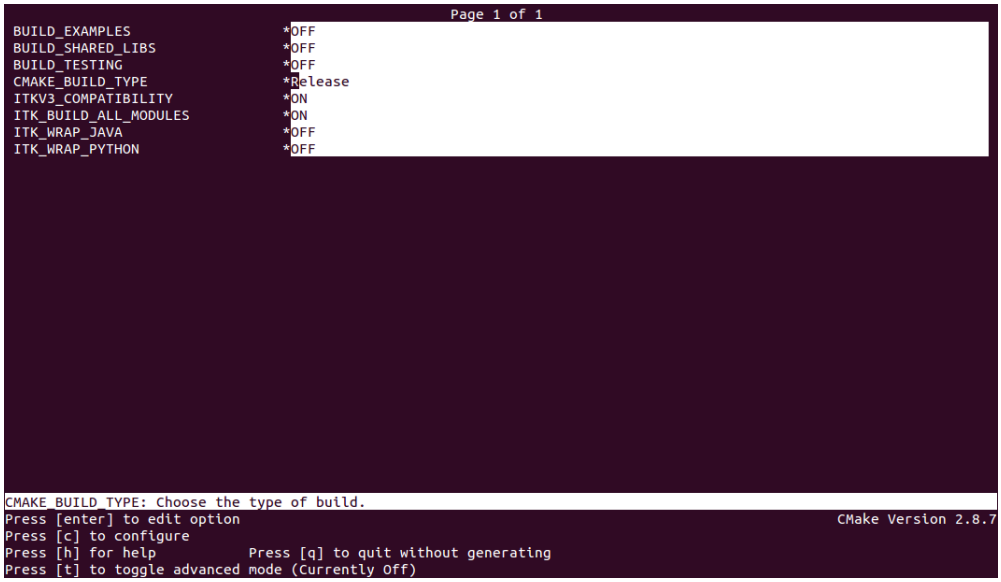

Figura 5.1: Pantalla de configuración de CMake.

```
export ITK_BIN=/home/user/itk_build
```

12. Reiniciar el terminal.

13. Extraer en nuestro *PC* el archivo *boost_1_47_0.tar.gz* que se encuentra en la ruta */software/resources/* del *CD-ROM*. En lo que sigue supondremos que este directorio es */home/user/boost_1_47_0*

14. Mediante el terminal accedemos a este directorio:

```
$ cd /home/user/boost_1_47_0
```

15. Ejecutamos la siguiente orden:

```
$ ./bootstrap
```

y seguidamente,

```
$ ./b2
```

para compilar la biblioteca *boost*. Este proceso puede tardar varios minutos.

16. Ahora copiaremos el contenido de la carpeta /src/ del *CD-ROM* en nuestro *PC* y abriremos el archivo de texto *CMakeLists.txt* que se encuentra en el directorio *geodesicSeg/*. Editaremos las líneas siguientes,

```
SET(BOOST_INCLUDEDIR <path_to_boost>)
SET(BOOST_LIBRARYDIR <path_to_boost/stage/lib>)
```

para que contengan la rutas correctas hacia el directorio donde se encuentran los binarios de *boost*,

```
SET(BOOST_INCLUDEDIR /home/user/boost_1_47_0)
SET(BOOST_LIBRARYDIR /home/user/boost_1_47_0/stage/lib)
```

17. Ahora ya tenemos todo listo para compilar nuestra aplicación de segmentación. Accederemos al directorio *geodesicSeg/* que contiene el archivo *CMakeLists.txt* que acabamos de editar y ejecutamos:

```
$ ccmake .
```

18. Al igual que antes, pulsaremos dos veces *'c'* y luego *'g'*, pero esta vez sin tocar nada.

19. El paso final es ejecutar *make,*

```
$ make
```

20. Repetimos los puntos del 16 al 20 para la aplicación *imageMerge*, que se encuentra en su correspondiente carpeta junto con *geodesicSeg* dentro de */src/.*

Capítulo 6

Conclusiones y trabajo futuro

Hemos dedicado este proyecto a la segmentación de zonas de interés en imágenes médicas, en concreto a detectar el hígado en una serie temporal de imágenes provenientes de tomografía axial computarizada. Fundamentalmente, el trabajo en el plano teórico ha consistido en encontrar y valorar qué algoritmos son de aplicabilidad a nuestro problema. Así mismo, en la parte práctica, hemos tenido, en primer lugar, que buscar y evaluar bibliotecas que resultaran útiles a nuestro propósito y, una vez elegido un software, probar y comparar los algoritmos ofrecidos por él para, finalmente, encontrar aquél que mejor detecte el hígado en las imágenes de prueba. Todo este trabajo ha supuesto un gran esfuerzo de formación tanto en el área de algoritmos de procesamiento de imágenes médicas como en las habilidades de programación.

El resultado ha sido plenamente satisfactorio. Los resultados obtenidos son correctos: la zona de interés es claramente identificada, rápidos: el tiempo de cómputo es bajo, y robustos: los típicos cambios de intensidad y ruido existentes en las imágenes no afectan sustancialmente a los resultados.

Estos rendimientos sugieren la idoneidad de emplear estas técnicas para segmentar estructuras biológicas en estudios funcionales (estudios de perfusión, y estudios de difusión en resonancia magnética, medicina nuclear, tomografía y ecografía) de carácter dinámico, en los que las características de la imagen cambian temporalmente.

Como trabajos futuros podemos sugerir los siguientes:

- En el ámbito circunscrito a este mismo proyecto, se podría realizar un procesamiento más intenso de la primera imagen de la serie temporal de forma que su segmentación fuera aprovechada para el procesamiento de las restantes imágenes. Respecto a la obtención de la máscara final de identifica la región, que ahora mismo es una suma de las máscaras obtenidas para cada imagen de la serie, cabría probar otras alternativas como la votación (un pixel aparece en la máscara final si aparece un número mínimo de veces) o la interpolación a una curva modelo (en la un pixel que en algún instante pertenezca a una máscara, figurará en la máscara final si su historia temporal se ajusta a la curva modelo de variación de intensidad en el tiempo). Así mismo, cabría realizar un conjunto de pruebas exhaustivas del software desarrollado en este proyecto sobre un abanico de casos mucho más amplio, para tratar de identificar más claramente el rango de valores de los parámetros de ajuste de los algoritmos utilizados.

- En un plano mucho más general, sugerimos nuevos trabajos que darían lugar a nuevos proyectos final de carrera. En primer lugar, sería muy conveniente disponer de una herramienta genérica que mostrara al usuario las imágenes objeto

de estudio y le permitiera seleccionar parámetros, y escoger zonas o puntos concretos, para el procesamiento posterior. En este último sentido, algunos de los algoritmos necesitan una «semilla» (un punto interno a la zona que identificar). Sería de gran utilidad una aplicación que encontrara, con alta probabilidad, un punto perteneciente a la zona. Por su puesto, esta aplicación es dependiente de la estructura a segmentar. Finalmente, en las series temporales de imágenes hay un movimiento de las estructuras debido a la respiración o al cambio de postura del paciente. Es muy conveniente disponer de una aplicación que compense dichos movimientos. Esta aplicación obtendría un nuevo juego de imágenes, donde habría movido datos de una altura a otra para un instante determinado.

Finalmente sólo nos queda destacar que el esfuerzo que ha supuesto la realización de este proyecto ha revertido, a nivel personal, en una inmensa mejora de conocimientos y habilidades, por lo que sentimos una gran satisfacción al concluirlo.

Bibliografía

[1] Isaac Bankman. *Handbook of Medical Image Processing and Analysis.* ACADEMIC PRESS, Johns Hopkins University, Baltimore, MD, USA, December, 2008.

[2] Tenn Francis Chen. Medical image segmentation using level sets. *Technical Report CS-2008-12*, 2008.

[3] Vicent Caselles *et al.* Geodesic active contours. *International Journal of Computer Vision*, 22(1):61–79, 1997.

[4] Will Schroeder *et al. The ITK Software Guide, Second Edition.* November 21, 2005.

[5] Matthew McAullife. *MIPAV User's Guide Volume 2: Algorithms.* National Institutes of Health, Center for Information Technology, Rockville, Maryland, USA, December 5, 2008.

[6] Tim McInerney and Demetri Terzopoulos. Deformable models in medical image analysis: A survey. *Medical Image Analysis*, 1(2):91–108, 1996.

[7] Amar Mitiche and JK Aggarwal. Image segmentation by conventional and information-integrating techniques: a synopsis. *Image and Vision Computing*, 3-2:50–62, 1985.

[8] Bjarne Stroustrup. *The C++ Programming Language (Third Edition).* Addison-Wesley, ISBN 0-201-88954-4, September 7, 2004.

[9] Hui Zhang and Jason E. Fritts. Image segmentation evaluation: A survey of unsupervised methods. *Computer Vision and Image Understanding*, 110:260–280, 2008.