UNIVERSIDAD DE VALLADOLID

Escuela Técnica Superior
de Ingenieros de Telecomunicación

TRABAJO FIN DE GRADO

GRADO EN TECNOLOGÍAS DE TELECOMUNICACIÓN

Implementación eficiente de un algoritmo de reconstrucción de imagen cardiaca basado en compensación de movimiento y muestreo compresivo

Autor: Rubén Palacios Sánchez
Tutor: Carlos Alberola López

23 de junio de 2016

Título:	Implementación eficiente de un algoritmo de reconstrucción de imagen cardiaca basado en compensación de movimiento y muestreo compresivo
Autor:	Rubén Palacios Sánchez
Tutor:	Carlos Alberola López
Departamento:	TSCIT

Miembros del Tribunal

Presidente:	Carlos Alberola López
Secretario:	Federico Simmross Wattenberg
Vocal:	Marcos Martín Fernandez
Suplente1:	Santiago Aja Fernández
Suplente2:	Rodrigo de Luis García
Fecha:	23 de junio de 2016
Calificación:	

Resumen

La imagen de resonancia cardiaca es el estándar de facto para el cálculo de determinados parámetros geométricos y funcionales del corazón. Su obtención, sin embargo, conlleva largos tiempos de estancia en la máquina de resonancia y requiere de prolongadas apneas para evitar efectos de movimiento. Por ello, la tendencia actual consiste en submuestrear la señal que recibe la máquina y aplicar técnicas de postprocesado, algunas de ellas basadas en compensación de movimiento del corazón, para mantener la calidad en la adquisición de las imágenes pero con tiempos de máquina reducidos. Los algoritmos a emplear suelen tener elevado coste computacional, por lo que es deseable su implementación sobre procesadores dedicados.

El TFG que se propone consiste en la paralelización e implementación en GPU de un algoritmo de registrado de imagen ya contrastado empleando OpenCL.

Palabras clave

Optimización en GPU, OpenCL, Registrado grupal, Gradiente descendente, Funciones *B-Spline*

Agradecimientos

Agradezco la ayuda a varias personas por la realización de este trabajo de fin de grado. En primer lugar agradecerle a mi tutor, Carlos Alberola López la oferta del proyecto y guiarme en el mismo en todo momento y a Federico Simmross Wattenberg por la ayuda con OpenCL tanto en la resolución de dudas como en proporcionarme material que ha sido de gran ayuda. También quería agraceder enormemente a

Javier Royuela del Val por prestarme atención en todo momento, y resolverme los problemas cuando no sabía avanzar, así como explicarme el método de renstrucción de imagen cardíaca en su totalidad y los objetivos del proyecto.

En segundo lugar quería agradecer a Santiago Rodrigo Sanz Estébanez por proporcionarme su proyecto de fin de carrera y resolverme algunas dudas del método.

Por último quería mencional a mi compañero de clase y del laboratorio Christian Fortich por proporcionarme ayuda con Latex y orientarme a cómo emplearlo en mi proyecto.

Índice general

Capítulo 1

Introducción general

1.1 Introducción

Las enfermedades cardiovasculares (CVD, *cardiovascular diseases*) son un grupo de trastornos del corazón y de los vasos sanguíneos y es la principal causa de muerte prematura en Europa, a pesar de que las CVD hayan disminuido considerablemente en las últimas décadas. Se estima que más del 80 % de la mortalidad provocada por las CVD suceden en países desarrollados.

Las CVD están fuertemente relacionadas con el estilo de vida, principalmente con el consumo del tabaco, hábitos poco saludables, la inactividad física y el estrés. La Organización Mundial de la Salud (OMS) ha publicado que más de tres cuartas partes de las muertes por CVD se podrían prevenir manteniendo un estilo de vida saludable [1].

La importancia del peso corporal, la masa corporal y otras medidas de adiposidad en la predicción de las CVD han sido objeto de largos debates. Muchos estudios han demostrado que la incidencia de ciertos tipos de enfermedades cardiovasculares es mayor en personas más pesadas. La obesidad está asociada a una elevada presión arterial, lípidos y glucosa en sangre, y los cambios en el peso corporal son coincidentes con los cambios en esos factores de riesgo para la enfermedad [2].

Las CVD también contribuyen a la baja productividad y al aumento de los costes sanitarios, especialmente en la presencia de una población que envejece. La OMS y otras organizaciones ya han informado de la tendencia de la mortalidad cardiovascular a lo largo del tiempo. Esas publicaciones mostraron un aumento sustancial de las CVD en los países de Europa central y oriental en los que hubo cambios nutricionales, económicos y políticos recientemente [3].

Las enfermedades del corazón, pueden ser identificadas y localizadas a través de la técnica de imagen de resonancia magnética (MRI, *Magnetic Resonance Imaging*), en particular con la imagen de resonancia cardiaca (MRI cardiaca) para el caso de las CVD.

Las técnicas de MRI cardiaca constituyen hoy en día una de las herramientas más valiosas en la ayuda al diagnóstico, tratamiento y seguimiento de las patologías cardiacas, ya que pueden proporcionar información cuantitativa sobre el corazón que sería imposible de obtener de otra manera, además carece de efectos perjudiciales para el paciente por no requerir de exposición a radiación elevada ni la inyección de agentes químicos neurotóxicos que pueden ser necesarios en otras técnicas de adquisición, aunque también presentan ciertos inconvenientes, siendo el más destacable, la presencia de artefactos en las imágenes.

Un aspecto crucial en el análisis de MRI es el carácter dinámico de las mismas, ya que el propósito

suele consistir en medir un fenómeno en el tiempo, que en este caso será el movimiento cardiaco. Para la reconstrucción de las imágenes después de la adquisición de datos se va a hacer este análisis de carácter dinámico, ya que para la realización de tareas médicas no suele ser suficiente con una sola imagen. Se propone un método de registrado para la estimación del movimiento de datos de imágenes médicas dinámicas [4].

En el presente proyecto se pretende implementar el método propuesto en [5] para el caso de imágenes MR Cine (MR-C), donde el patrón de intensidad de puntos a lo largo del tiempo se considera no muy variable [6].

El método a utilizar se basa en un modelo de deformación libre (*Free Form Deformation*) 3D (2D + tiempo) basado en B-splines, una métrica de similitud que minimiza las variaciones en intensidad a lo largo del tiempo y una optimización empleando un método de gradiente descendente con una estimación adaptativa del tamaño de los pesos.

FFD es una técnica geométrica usada para modelar deformaciones simples. Se basa en la idea de encerrar un objeto dentro de un cubo u otro objeto y transformar éste último de modo que el objeto interior inicial se vea deformado en consecuencia [7].

En relación al tipo de formulación del problema, se pueden establecer dos grupos bien diferenciados: registrado de las imágenes de manera consecutiva por pares (transformaciones secuenciales) y otro que será escogiendo una imagen de referencia de una determinada fase cardiaca y realizar el registrado del resto de las imágenes a partir de ésta (transformaciones no secuenciales).

Las transformaciones secuenciales, tienen el inconveniente de que al estimar el movimiento entre pares de imágenes, podría haber un pequeño error, el cual se iría acumulando. Las transformaciones no secuenciales, tienen el inconveniente de que la intensidad de la secuencia de imágenes presentará variaciones que se acumularán en el tiempo, haciendo que el registrado sea cada vez más difícil a medida que pasa el tiempo [8].

Para solucionar los problemas de los tipos de transformaciones, se establece el registrado grupal, donde la métrica se utiliza para realizar un seguimiento de toda la trayectoria a lo largo del tiempo de cada punto [6].

Un problema importante en la MRI se produce debido al movimiento provocado por la respiración del paciente, provocando desalineamientos en las imágenes adquiridas, los cuales han de ser corregidos. Normalmente una correcta adquisición solo será posible durante un corto intervalo de tiempo de la adquisición total, antes de producirse un cambio brusco en la posición del corazón al respirar de nuevo [9]. En consecuencia, la apnea en la MRI cardiaca ha sido la forma más común de obtener las imágenes del miocardio, ahora bien, el tiempo de estancia en el escáner ha de ser rebajado con el propósito de hacer un correcto escaneo a los pacientes con dificultades de mantener apneas prolongadas. Así pues, será necesaria una optimización del registrado en cuanto a tiempos se refiere [10].

1.2 Objetivos

El objetivo perseguido con la elaboración de este proyecto es, por lo expuesto en la sección anterior, la compensación del movimiento en imágenes de resonancia magnética cardíaca usando el registrado grupal para el análisis preciso de las curvas de intensidad temporales y una optimización de los tiempos de ejecución mediante el uso de la GPU.

Este objetivo global se desglosa en los siguientes subobjetivos:

- Implementación de un algoritmo para el procesado de imágenes en MR-C realizando una adaptación del método propuesto para un correcto seguimiento en base a los fundamentos del procesado de imagen médica previamente estudiados.
- Paralelización e implementación en GPU del algoritmo de registrado de imagen con el objeto de minimizar los tiempos de estancia en el escáner.
- Uso de técnicas de postprocesado, para que reduzca aun más el tiempo de estancia en la máquina de resonancia. En este postprocesado también es de gran importancia la paralelización, ya que la obtención de resultados será más rápida y además de poder atender a más pacientes, es conveniente saber si no se ha producido ningún error antes de que el paciente se marche, puesto que, en este caso, habría que repetir el proceso [11].
- Estudio de los resultados en función de diversos parámetros del problema mediante una comparación de los tiempos de ejecución del algoritmo.

1.3 Fases para la implementación del proyecto

En el presente proyecto se han seguido las siguientes fases:

1. Estudio general del algoritmo de reconstrucción de imagen cardiaca que se va a implementar, cómo se realiza el registrado no rígido de la MRI dinámicas basado en B-splines, en qué consiste el registrado grupal, el algoritmo de la FFD y el método del gradiente descendente.
2. Estudio de la paralelización e implementación en GPU del algoritmo de registrado de imagen así como la importancia de dicha paralelización.
3. Familiarización con la programación en GPU y realización de programas sencillos empleando OpenCL.
4. Implementación en C de las rutinas de MATLAB sobre el registrado.
5. Implementación en OpenCL de aquellas rutinas críticas en tiempo de ejecución.
6. Realización de pruebas para la comprobación del registrado y la observación de si realmente mejora el algoritmo en cuanto a tiempos de ejecución se refiere cambiando el valor de los diferentes parámetros y tamaños de datos.

1.4 Material empleado

Será necesario, para la realización de este proyecto, el acceso a las herramientas *software* y *hardware* que se detallan a continuación:

Software:

- MATLAB 2015a: lenguaje de programación técnico de alto nivel y entorno de desarrollo integrado para el desarrollo de algoritmos, visualización, análisis de datos, y computación numérica [12].
- Eclipse Mars.1: plataforma de *software* compuesto por un compuesto de programación de código abierto [13].
- Latex: sistema de composición de textos, orientado a la creación de documentos escrito.
- Sistema Operativo Scientific Linux 7.2.

Hardware:

- PC con las siguientes características:
  - Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz.
  - 16 GB de memoria RAM.
  - Disco duro de 500 GB de capacidad.

Finalmente, el trabajo se realizará en el Laboratorio 26 de la Escuela Técnica Superior de Ingenieros de Telecomunicación (ETSIT) de la Universidad de Valladolid (UVA).

1.5 Estructura del documento

En cuanto a la estructura del informe, el capítulo 2 se dedica al registrado, el cual se dividirá en 5 secciones. Empezaremos en la sección 2.1 con una introducción a la estructura cardiaca de la que vamos a realizar el registrado.

En la sección 2.2, se realiza un recorrido introductorio al registrado, exponiendo los principios básicos que se van a emplear, añadiendo un repaso al registrado grupal, que es en el que se centra el proyecto.

En la sección 2.3, se exponen las fases a seguir para la realización del registrado con objeto de facilitar la posterior implementación.

En cuanto a la sección 2.4, se describe en mayor detalle el método propuesto para el registrado grupal en MR-C desde un punto de vista teórico.

En la última sección del capítulo 2, se hablará de la optimización en GPU y cómo puede ayudar el uso de la GPU a los pacientes, y además se mencionará el lenguaje de programación utilizado para realizar dicha optimización.

El capítulo 3 se dedicará a la programación en GPU. En este capítulo se describen las ventajas e inconvenientes de la GPU frente a la CPU, de los diferentes estándares de programación para la GPU y mencionaremos la terminología del estándar que se va a usar, que será OpenCL. Por último, este capítulo se ha reservado para tener un primer contacto con OpenCL, donde veremos cómo se realiza un primer programa sencillo.

En el capítulo 4 se realiza la implementación de código. Este capítulo se divide en dos secciones. En la primera sección se describen las fases para la implementación en C de las rutinas de MATLAB sobre el registrado y la posterior implementación en en OpenCL de aquellas rutinas de mayor carga computacional. En la segunda sección se detallan una serie de consejos que pueden ser de gran ayuda en la implementación del código.

En el capítulo 5 se van a mostrar los resultados, realizando diferentes pruebas variando el valor de los parámetros que ha de introducir el usuario, así como diferentes tamaños de los datos iniciales. En este apartado el objetivo principal será comparar los diferentes tiempos que se tarda en hacer el registrado en diferentes ocasiones, y si vale la pena utilizar más precisión o una ROI mayor.

La memoria concluye con una sección de conclusiones y líneas futuras.

Capítulo 2

Herramientas básicas

2.1 Estructura cardíaca

Este proyecto se centra en las estructuras del corazón; el corazón es el órgano muscular principal del aparato circulatorio, es un músculo hueco y piramidal situado en la cavidad torácica y funciona como una bomba aspirante e impelente, impulsando la sangre a todo el cuerpo.

La segmentación de las estructuras de interés en las imágenes (estableciendo el marco de trabajo) se realiza manualmente en las regiones correspondientes al endocardio del ventrículo izquierdo (LVen), al epicardio del ventrículo izquierdo (LVep), al miocardio del ventrículo izquierdo (LVmi), teniendo en cuenta que se obtiene de la diferencia de las dos primeras estructuras (siempre y cuando no entren en contacto) y al endocardio del ventrículo derecho (RVen). Dichas regiones se marcarán finalmente sobre todos los instantes temporales de la secuencia de imágenes.

En la figura 2.1 se muestra la forma de cada una de las estructuras:

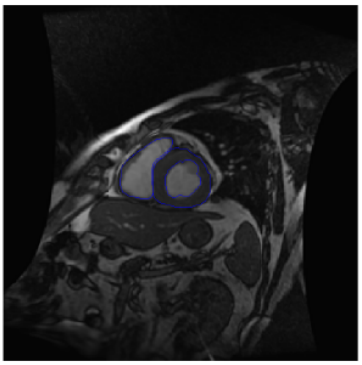


Figura 2.1: Estructura de interés.

El endocardio es una membrana que recubre localmente las cavidades del corazón. Forma el revestimiento interno de las aurículas y ventrículos. El endocardio es más grueso en las aurículas y presenta tres capas: la capa interna o endotelial, la capa media o subendotelial y la capa externa o subendocárdica.

Se trata de un delgado revestimiento interno del corazón que se encuentra constituido por células endoteliales y una delgada capa de tejido conectivo laxo. En el endocardio mural se agregan una túnica músculo-elástica rudimentaria y antes del miocardio, una capa gruesa subendocárdica de tejido conectivo laxo vascularizado. Sin embargo, debemos de recordar que el endocardio valvular es avascular.

Las válvulas semilunares (aórtica y pulmonar) presentan comisuras y las auriculoventriculares constituyen un aparato valvular (válvula, cuerdas y músculo papilar). Todas se hallan insertas en un anillo fibroso denso. Con la edad, se vuelven más gruesas y más opacas [14].

El epicardio es una membrana viscosa (la capa visceral del pericardio) que cubre la superficie externa del corazón. Esta membrana junto con la capa parietal, constituyen la bolsa pericárdica en que se encuentra el corazón.

Está formado por una única capa de células mesoteliales, cuyas células varían entre planas o cúbicas según el grado de distensión y tejido conectivo laxo que contiene los vasos sanguíneos y nervios; presenta además una importante cantidad de tejido adiposo.

El pericardio es una membrana fibroserosa de dos capas que envuelve al corazón y a los grandes vasos separándolos de las estructuras vecinas. Forma una especie de bolsa o saco que cubre completamente al corazón y se prolonga hasta las raíces de los grandes vasos [15].

El miocardio es el tejido muscular del corazón, músculo encargado de bombear la sangre por el sistema circulatorio mediante contracción. Contiene una red abundante de capilares indispensables para cubrir sus necesidades energéticas. El músculo cardíaco funciona involuntariamente, sin tener estimulación nerviosa.

En los ventrículos las fibras musculares alcanzan su mayor espesor sobre todo en el ventrículo izquierdo, siendo éste el encargado de bombear sangre oxigenada a través de la arteria aorta.

2.2 Fundamentos del registrado

El registrado es una técnica de tratamiento de imágenes que consiste en hacer corresponder las mismas, que provienen de una o varias fuentes. Se puede utilizar para cambiar la apariencia de las mismas mediante rotaciones, translaciones o extensiones entre otras. En otras palabras, el registrado de imágenes se refiere a la búsqueda de una transformación óptima para la alineación entre los objetos en (al menos) dos imágenes.

Tiene muchas aplicaciones en procesado de imagen, como la fusión de la información de una imagen, el modelado de la población o la alineación de estudios dependientes del tiempo, que será la aplicación de este documento. Una aplicación importante del registrado de imágenes se da en medicina. En imágenes médicas, los estudios dependientes del tiempo incluyen la deformación de un cuerpo continuo, el desarrollo de una patología dada o el paso de un fluido a través de una determinada región. En estas situaciones, el objetivo suele ser alinear un conjunto de más de dos imágenes. La metodología propuesta (o una forma más simple de la misma) es generalmente aplicable a la mayoría de los problemas de registrado en los estudios dependientes del tiempo [6].

El registrado de imágenes se ha clasificado entre registrado rígido y registrado no rígido. En el registrado rígido, las imágenes se supone que son de objetos que solo tendrán que ser rotados o transladados con respecto a otro para lograr la correspondencia, y son aquellas transformaciones geométricas que conservan todas las distancias. Estas transformaciones solo permiten operaciones de rotación y translación. En el registrado no rígido, la correspondencia entre las imágenes no se puede alcanzar sin la extensión de las mismas, es decir, estas transformaciones no conservan todas las distancias. La mayor parte de las partes del cuerpo humano no se ajusta al registrado rígido y la mayoría del trabajo en torno al registrado hoy en día envuelve el desarrollo de las técnicas de registrado no rígido[16].

Como ya se ha mencionado, en el registrado de imagen de cine cardiaco (MR-C), el patrón de intensidad de puntos a lo largo del tiempo se considera no muy variable y para el problema específico de la alineación del miocardio en las MRI, hay que tener en cuenta que:

- El registrado rígido no sería suficiente en la práctica. Por lo tanto, los procedimientos de registrado no rígido serían los más adecuados.

- El objetivo es el alineamiento de toda la secuencia. Por lo tanto, el procedimiento podría beneficiarse de la adopción de una perspectiva global en su formulación, lo que a la vez influye en la estrategia de optimización que debe aplicarse [6].

En el presente proyecto se utiliza un método de registrado basado en puntos de control para estructuras de interés del corazón. Para una correcta estimación del movimiento cardiaco, se debe considerar la caracterización de la dinámica cardiaca local, la cual es descrita por una FFD basada en B-splines, que es una buena herramienta para el modelado de deformaciones 3-D. La idea básica de las FFD es deformar objetos manipulando una malla subyacente de puntos de control. La deformación resultante controla la forma de los objetos y produce un suavizado y una transformación. Las funciones B-spline, están controladas localmente, lo que hace que sean muy eficientes computacionalmente, incluso si tenemos un gran número de puntos de control. Se empleará un registrado no rígido ya que el registrado rígido no es capaz de realizar correctamente la compensación del movimiento en la práctica [17].

En la figura 2.2 se muestra de forma gráfica la manipulación de una malla de puntos de control que se le aplicará a la ROI de una imagen.

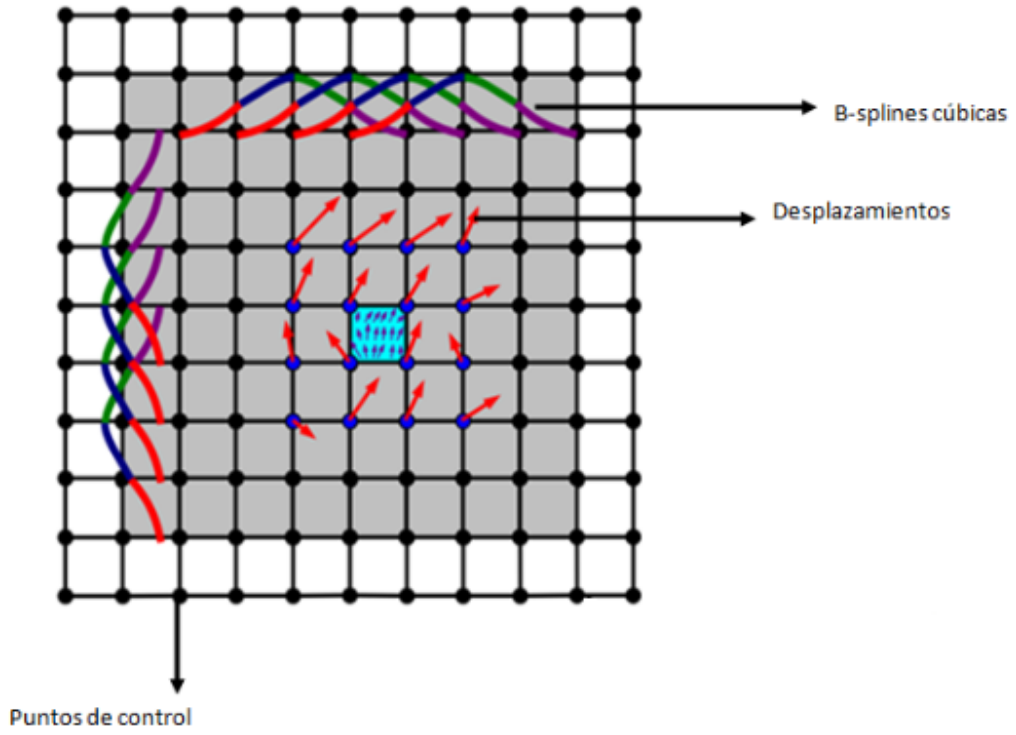


Figura 2.2: Manipulación de una malla B-spline.

El objetivo consiste en deformar una placa rectangular en 2D ó 3D, por la manipulación de una malla superpuesta sobre ella. Se propone el producto tensorial de B-splines cúbicos como la función de deformación de FFD, ya que una B-spline tiene un control local. Esta propiedad hace posible manipular localmente la malla cuando un punto en la placa es desplazado a la posición especificada, de modo que la nueva malla pueda ser calculada eficientemente a pesar de un gran número de puntos [18].

2.2.1 Registrado grupal

En el presente proyecto, se propone un registrado grupal y no rígido basado en el gradiente descendente.

Suponemos que nos dan un conjunto de imágenes $\beth = I_1, ..., I_N \quad con \quad 1 \leq n \leq N$ indexadas cada una de ellas $(I_n(x_n))$. En la figura 2.3 se indica que el problema del registrado consiste en encontrar un conjunto de parámetros desde un marco común de referencia al espacio de las imágenes.

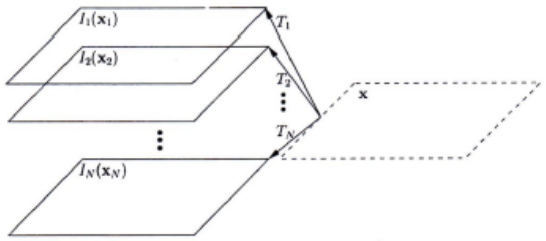


Figura 2.3: Representación gráfica del problema del registrado grupal.

El problema del registrado será encontrar el conjunto de transformaciones $\Gamma = T_n : \mathtt{x}_n = T_n(\mathtt{x})$, desde una referencia común 'x' de tal manera que se minimice la función de coste:

$$H(\Gamma) = \int_{\chi} V(I_1(T_1(\mathtt{x})), ..., I_N(T_N(\mathtt{x})))d\mathtt{x} \tag{2.1}$$

Esta formulación no hace ninguna suposición sobre la selección de la referencia. Para restringirlo sin dar prioridad a una imagen, una posibilidad podría ser restringir la deformación media para ser la identidad de la transformación:

$$\frac{1}{N}\sum_{n=1}^{N} T_n(\mathtt{x}) = \mathtt{x} \tag{2.2}$$

Las cuestiones clave de esta formulación es la definición de la métrica y el supuesto conjunto de transformaciones Γ.

La métrica propuesta en este proyecto es la métrica grupal, la cual hace una comparación con los valores de intensidad promedio de toda la secuencia:

$$V_G(\mathtt{x}) = \frac{1}{N}\sum_{n=1}^{N}\left(I_n(T_n(\mathtt{x})) - \frac{1}{N}\sum_{n'=1}^{N} I_{n'}(T_{n'}(\mathtt{x}))\right)^2 \tag{2.3}$$

La alineación conjunta de toda la secuencia simultáneamente no ha sido un concepto con gran desarrollo, a pesar de que estos métodos sean más robustos que sus homólogos basados en pares [6].

2.3 Estructura del registrado

Como ya se ha dicho anteriormente, el algoritmo del registrado va a estar basado en B-splines, y se puede descomponer en tres tareas principales:

- El modelo de deformación B-splines.
- El cálculo de la métrica, que es la varianza de la intensidad de la imagen a lo largo del tiempo.
- El proceso de optimización, que varía los parámetros del modelo de transformación para maximizar el criterio de coincidencia.

Modelo de deformación B-splines

El modelo de deformación define cómo se puede deformar una imagen para que coincida con otra. El modelo de deformación sirve para dos propósitos: lo primero es controlar cómo mover las características de una imagen respecto a una métrica y lo segundo es interpolar entre esas características donde no hay información que se use.

El registrado basado en B-splines es una de las transformaciones más importantes a lo largo de estos últimos años. Los algoritmos de registrado basados en splines utilizan los llamados puntos de control, tanto en la imagen de origen como en la de destino, y una función spline para definir las correspondencias fuera de esos puntos. Cada punto de control que pertenece a una malla spline tiene una influencia sobre todos los demás, ya que si perturba su posición, todos los demás puntos de la imagen transformada cambiarían. Esto sería una desventaja importante, ya que el coste computacional de mover un solo punto aumenta de forma abrupta.

Por el contrario, las B-splines solo se definen en la proximidad de cada punto de control, por lo que la perturbación de un punto de control solo afectaría a la transformación de los alrededores de ese punto.

Las técnicas de registrado no rígido basado en B-splines son muy populares debido a su aplicabilidad general, a la transparencia y a su eficiencia computacional. La correspondencia de los puntos de control es lo que se utilizará para el registrado, y será lo que determine el desplazamiento de cada imagen en cada punto. Una vez establecida la correspondencia entre los pares de puntos de control, se hará una interpolación.

Cálculo de la métrica

Como ya se ha dicho anteriormente, se hará uso del método del registrado grupal, mediante el cálculo de una métrica, que representa las variaciones de intensidad. En el registrado grupal, la métrica usada para realizar el seguimiento incorpora simultáneamente la información de la imagen de toda la trayectoria temporal que sigue un determinado punto.

Para la métrica propuesta se combina información de la orientación del gradiente y la intensidad de las imágenes, lo que favorecerá soluciones suaves que alineen la secuencia entera de forma simultánea, lo que implica una mayor robustez en comparación con los métodos basados en registrado por pares [19].

El objetivo en sí es conseguir una reducción del valor de la métrica final. Esa minimización se consigue mediante el gradiente descendente, esto es, hace uso del negativo del gradiente para aproximarse al óptimo.

Optimización

La optimización se refiere a la manera en que se ajusta la transformación para mejorar la similitud de la imagen. Un buen optimizador es uno que encuentra la mejor transformación posible de forma rápida y fiable.

En cuanto a la estrategia de optimización se plantea un procedimiento iterativo, que partiendo de la transformación nula realice el registrado grupal basado en una métrica que representa la variación de la intensidad a lo largo de la secuencia.

Para elegir un buen optimizador hace falta entender bien el problema del registrado, las restricciones que se pueden aplicar y conocimientos del análisis numérico. En las aplicaciones de registrado no rígido, escoger o diseñar un optimizador no es tarea fácil, y cuanto más flexible sea el modelo de transformación, más parámetros a describir harían falta [16].

2.4 Método

Como ya se ha comentado, se va a realizar una FFD basada en B-splines, y se van a utilizar los llamados puntos de control. También se va definir una región de interés (ROI, *Region Of Interest*). Esa ROI se definirá estableciendo un círculo en el centro de las imágenes, con un radio a elegir por el usuario según la zona de mayor actividad, y un pequeño margen, también elegido por el usuario para evitar discontinuidades importantes.

En la figura 2.4 se observa un ejemplo de la ROI aplicada a una imagen perteneciente a uno de los cortes de las imágenes cardiacas:

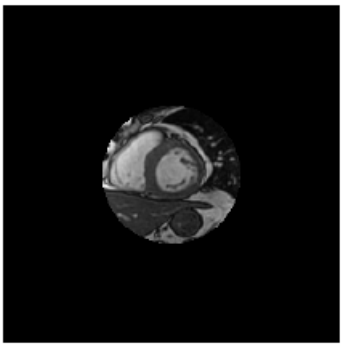


Figura 2.4: Ejemplo de una ROI más un margen.

La optimización se relizará solamente en la ROI, donde estarán las zonas de más movimiento, para así evitar zonas donde el movimiento es escaso, y esto hará reducir los costes computacionales considerablemente. Sobre la ROI se definirá una *bounding box*, donde se aplicará la optimización.

En cuanto a los puntos de control, será necesario definir una malla sobre la imagen, dada por los puntos $P = \{p_u\} = \{p_{u1\ldots ul}\}$ con $C_{1l} \leq u_l \leq C_{2l}$, y al trabajar en 2-D, $L = 2$.

Las coordenadas de los puntos de control en la imagen vendrán dadas por:

$$p_u = c + \Delta^p \circ u \tag{2.4}$$

donde $\Delta^p = (\Delta_1^p, \Delta_2^p)$ será la resolución en píxeles de la malla de puntos de control y $\circ$ el producto de Hadamard[1].

Los límites de la malla dados por la matriz 'C' se establecen de modo que se cubra por completo la ROI con un cierto margen para la aproximación vía B-splines para así evitar que valores grandes de desplazamientos produzcan inconsistencias en los bordes de la ROI dando lugar a errores en la interpolación.

Se define la transformación del sistema de coordenadas de los píxeles de la imagen 'x' al sistema de coordenadas de los puntos de control 'v' como:

$$\mathrm{v}(x) = x \circ \widehat{\Delta^p} \tag{2.5}$$

donde $\widehat{\Delta^p}$ denota la inversa de Hadamard de Δ^p para cada uno de sus elementos.

En la figura 2.5 se muestra una disposición de los parámetros característicos del método:

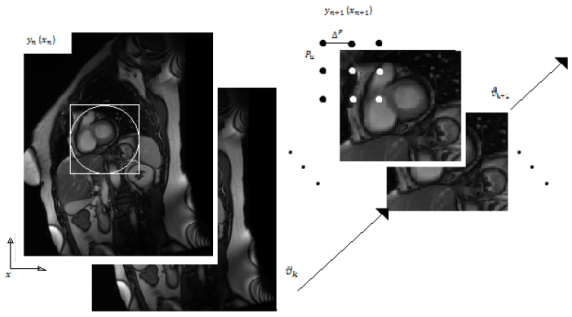


Figura 2.5: Disposición de los parámetros característicos del método.

El conjunto de parámetros de la transformación anterior viene dado por una serie de desplazamientos sobre los puntos de control para cada una de las imágenes, denotando θ_{nu} a cada uno de estos desplazamientos. La transformación queda entonces:

$$x_n = x + \sum_{u_1=C_{11}}^{C_{21}} \sum_{u_2=C_{12}}^{C_{22}} \left(\prod_{l=1}^{L} B_E(v_l(x_l - p_u l)) \right) \theta_{nu} \tag{2.6}$$

donde B_E representa la función B-spline uniforme de grado E.

Cabe notar que para un uso compacto de las funciones B-spline es importante usar órdenes 1, 2 ó 3, puesto que sus derivadas primeras son continuas (en caso de recurrir a derivadas de orden superior, habrá que recurrir también a órdenes superiores). A continuación se indica como reseña la primera derivada:

$$\frac{\partial x_{nl}}{\partial \theta_{nk}} = \frac{\partial x_{n1}}{\partial \theta_{nul'}} = \left(\prod_{l''=1}^{L} B_E(v_{l''}(x_{l''} - p_{ul''})) \right) \delta(l, l') \tag{2.7}$$

[1]El producto de Hadamard es una operación binaria que toma matrices de mismas dimensiones y produce otra matriz donde cada elemento de la matriz resultado es el producto de los elementos de esa misma posición de las matrices originales

En la figura 2.6 se observa la disposición de los desplazamientos de todos los puntos de la imagen debido a la deformación provocada por la malla de puntos de control sobre la ROI:

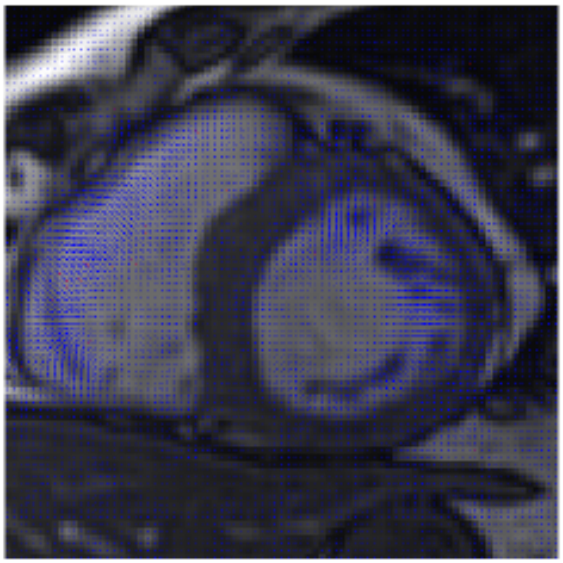


Figura 2.6: Disposición de los desplazamientos de los píxeles sobre la imagen.

Para su implementación óptima, se debe indicar que para un determinado parámetro habrá algunos píxeles para los que las derivadas sean nulas. Además, el valor de estas derivadas ha de ser precomputado, puesto que no existe dependencia con los valores que toma la matriz de parámetros θ

De cara a determinar las dimensiones de la malla de puntos de control, nos interesa, en primer lugar, definir el radio de influencia de la malla de puntos de control como:

$$r_l^P = \frac{(E+1)\Delta_l^P}{2} \tag{2.8}$$

De ahí se extrae que un determinado punto de control afectará a los puntos contenidos en la caja que recubre a la ROI si y sólo si:

$$(p_{ul} + r_l^P > x_l^{C_1}) \bigwedge (p_{ul} - r_l^P < x_l^{C_2}) \quad \forall l \tag{2.9}$$

siendo $\bigwedge$ la conjunción lógica[2]. Desarrollando las anteriores condiciones, tenemos que los mínimos en la matriz que garantizan que no existe ningún punto de control extra que se pueda añadir que tenga influencia sobre la caja que recubre la ROI son:

$$\begin{aligned} C_{1l} &= \lfloor \frac{c_l - x_l^{C_1} + r_l^P}{\Delta_l^P} \rfloor \\ C_{2l} &= \lfloor \frac{c_2 - c_l + r_l^P}{\Delta_l^P} \rfloor \end{aligned} \tag{2.10}$$

[2]La conjunción lógica de $A \bigwedge B$ significa que la proposición A es verdadera si A y B son ambas verdaderas; de otra manera es falsa

A tenor de lo anterior, para la implementación se debe tener en cuenta la propiedad de soporte compacto de los B-splines.

En primer lugar, se han de calcular los valores no nulos de $\prod_{l=1}^{L} B_E(v_l(x_l - p_{ul}))$ para cada uno de los píxeles de la imagen para almacenarlos en una matriz.

Además se deben indexar los índices de los parámetros a los que corresponden cada uno de los elementos de la matriz, codificando los límites inferior y superior de los índices 'u' de los parámetros sobre los cuales el producto de las bases B-splines es no nulo.

Se extrae que el punto 'x' se sitúa en el índice:

$$\mathtt{u}^{\mathtt{x}} = (\mathtt{x} - c) \circ \widehat{\Delta^p} \tag{2.11}$$

y los límites para dicho punto vienen dados por:

$$U_l^x = \begin{cases} [round(u_l^x) - \frac{E}{2}, round(u_l^x) + \frac{E}{2}] & \text{si } E \text{ es par} \\ [\lfloor u_l \rfloor - \frac{E-1}{2}, \lfloor u_l \rfloor + \frac{E-1}{2}] & \text{si } E \text{ es impar} \end{cases} \tag{2.12}$$

De forma similar se calculará el gradiente del coste (almacenando sólo los valores en los que la derivada es no nula), usando una matriz junto con la correspondencia entre las entradas de dicha matriz y los píxeles de la imagen. Los límites inferior y superior que denotan el área de influencia se codifican sobre los valores no nulos de la derivada respecto al parámetro correspondiente sobre cada una de las dimensiones 'l' de la imagen.

Estos límites vendrán dados por:

$$\chi_l^{\mathtt{x}} = [\lceil p_{ul} - \lceil r_l^P \rceil \rceil, \lceil p_{ul} - \lceil r_l^P \rceil \rceil + 2\lceil r_l^P \rceil] \tag{2.13}$$

Finalmente, la transformación en (2.6) se puede calcular con todos estos elementos mediante la multiplicación de la matriz de valores del producto de las bases de B-splines por los parámetros θ correspondientes a los índices dados en la matriz de índices y U_l^x la posterior suma de los elementos de la matriz resultado.

El objetivo del registrado consiste en alinear toda la secuencia. Para ello, sería beneficioso adoptar un procedimiento con una perspectiva global en su formulación, que influirá en la estrategia de optimización que se aplicará posteriormente.

El objetivo en sí es la minimización de la función de coste del gradiente dado por la siguiente ecuación:

$$H(\tau) = \int_{\chi} V_\tau(\mathtt{x}) d\mathtt{x} \tag{2.14}$$

donde 'V' representa la métrica a utilizar, de modo que, además de conseguir una reducción óptima en el valor de la métrica final, se obtenga un resultado realista de forma visual.

La minimización mediante el gradiente descendente relaciona la función de coste a minimizar con el gradiente integrando sobre toda la ROI.

Entonces, el problema del registrado se transforma en la minimización del valor de la métrica sobre los parámetros $\theta_{k,n}$. El gradiente total vendrá dado por:

$$\frac{\partial V}{\partial \theta_{k,n}}(x) = \frac{\partial V}{\partial y_n} \sum_{l=1}^{L} \frac{\partial y_n}{\partial x_n^l} \frac{\partial x_n^l}{\partial \theta_{k,n}}(x) \tag{2.15}$$

donde cada término de la derecha se corresponde respectivamente al gradiente de la métrica $\frac{\partial V}{\partial y}$, el gradiente de la imagen (intensidades) $\frac{\partial y}{\partial x}$ y el gradiente de la transformación $\frac{\partial x}{\partial \theta}$ y finalmente, se combinan para dar lugar al gradiente total $\frac{\partial V}{\partial \theta}$.

Evidentemente, para facilitar la modularidad y reducir el coste computacional, todos los gradientes irán encapsulados en diferentes funciones.

Por otro lado, debido a que la transformación basada en B-splines usa únicamente información local, se puede dar el caso de que en determinadas regiones de la imagen el comportamiento de la transformación fuera muy irregular, dando lugar a la aparición de artefactos en la imagen transformada, y lo ideal sería añadir los términos de suavidad, los cuales no han sido implementados en el presente proyecto.

En cuanto a la estrategia de optimización, como ya se ha mencionado antes, se plantea a continuación un procedimiento iterativo, que partiendo de la transformación nula, realice el registrado grupal basado en una métrica de cuadrados medios usando interpolación lineal. Por lo tanto se busca:

$$\tau* = \underset{\tau}{\operatorname{argmin}}\, H(\tau) = \underset{\theta}{\operatorname{argmin}}\, H(\theta) \tag{2.16}$$

Para ello se plantea un método de gradiente descendente con paso fijo (en todo caso, siempre positivo) ajustado empíricamente, con igual valor para todos los parámetros, que será aplicado sobre el gradiente, una vez se haya proyectado el gradiente sobre el subespacio dado por (2) en [6] antes de actualizar los parámetros, realizándose del siguiente modo:

$$\theta_{n+1} = \theta_n - W \circ \frac{\widehat{\nabla H(\theta)}}{|\chi|} \tag{2.17}$$

donde θ denota la matriz de transformación y 'H' la función de coste definida.

El método de gradiente descendente debe considerar la condición dada por (2) en [6]: $\frac{1}{N}\sum_{n=1}^{N} T_n(\mathtt{x}) = \mathtt{x}$ (esto es, se debe proyectar el gradiente sobre el subespacio dado por (2) en [6] antes de actualizar los parámetros). La proyección del gradiente se realiza de la siguiente manera:

$$\frac{\widehat{\partial H}}{\partial \theta_{k,n}} = \frac{\partial H}{\partial \theta_{k,n}} - \frac{1}{N}\sum_{n=1}^{N} \frac{\partial H}{\partial \theta_{k,n}} \tag{2.18}$$

En cuanto a la formulación práctica para la optimización del problema, en este caso habrá que tomar especial atención a las relaciones dimensionales a establecer, dado que el sistema de coordenadas definido en MATLAB y en C difiere del seguido en esta notación, por lo que se ha de tener en cuenta siempre el sistema en el que se está trabajando, para así evitar inconsistencias con respecto a los ejes de trabajo.

Debido a que la definición de la ROI implica la simetría de ésta y que el espaciado de puntos de control va a ser el mismo tanto vertical como horizontalmente, se tiene que la caja definida por la matriz 'C' será un cuadrado que circunscriba a la ROI.

Cabe indicar que debido al hecho de que se está realizando el registrado solamente en la bounding box de la ROI, es de esperar que existan discontinuidades en los límites de ésta, de forma que la transformación no se propague suavemente, dando lugar a ciertos artefactos, y eso se podría arreglar con un suavizado, el cual no se realiza en el presente proyecto.

Una vez definida la estrategia de optimización se ha de fijar las condiciones de salida del optimizador. Para ello, como condición de parada se debe cumplir simultáneamente que:

$\frac{1}{KN}\|\theta_{n-1} - \theta_n\| < \epsilon_\tau \quad y \quad \frac{1}{|\chi|}(H_{n-1} - H_n) < \epsilon_H$, con los umbrales mayores que cero y fijados empíricamente.

Esto consiste en fijar un umbral tanto para la variación de la métrica, ecuación (2.14) (un 0.5 % de la inicial) y en la norma de la matriz de parámetros de la transformación, ecuación (2.17) (un valor fijo que representará el desplazamiento de 0.1 píxeles para tener una convergencia plena).

Además se fijará un número máximo de iteraciones (aproximadamente entre 50 y 60 iteraciones), y por tanto ésto dará lugar a otra condición de parada que consistirá simplemente en $n \geq n_{max}$.

2.5 Optimización en GPU

El algoritmo de la FFD requiere demasiado tiempo de computación, por eso vamos a hacer uso de algoritmos adecuados para las ejecuciones en paralelo con la GPU. El cuello de botella es la computación de la B-spline cúbica, y se ha trabajado mucho en acelerar esa parte mediante distintas arquitecturas:

- Jiang et al. empleó una implementación basada en FPGA (*Field Programmable Gate Array*) con lo que pudo acelerar el tiempo de ejecución 3.2 veces con respecto al del de la CPU de 2.666 GHz.
- Rohlfing and Maurer redujeron más de 50 veces el tiempo de computación con 64 CPUs de una supercomputadora de memoria compartida.
- Más recientemente, Rohrer et al presentó un implementación basada en computación de B-spline en una plataforma Cell/B.E. (*Cell Broadband Engine*). Su arquitectura era 40 % más rápida que la ejecución en serie de un ordenador normal.

Esas técnicas proporcionan una mejora considerable en cuanto a tiempos de computación se refiere, pero requieren de un gran conocimiento técnico y son de coste alto. El uso de las GPUs es una buena solución de rendimiento alto y buen coste, además de solo ser necesario tener conocimientos del lenguaje de programación en C y apenas conciencia del hardware.

El principal requisito para que un algoritmo se beneficie de la ejecución de la GPU es el paralelismo de los datos. El algoritmo de FFD, consta de:

- La transformación de las imágenes utilizando las B-splines y una función de interpolación.
- La evaluación de una función de coste.
- Una optimización de esa función.

Individualmente, dichas componentes pueden ser formuladas de una manera en paralelo. Como ya se ha dicho, el algoritmo de la FFD consiste en deformar un volumen de una imagen mediante B-splines cúbicas. Esta técnica tiene la gran característica de garantizar la deformación continua en C^2. En la figura 2.7 se observa el algoritmo de la FFD basado en B-splines, donde se muestra el proceso de deformación de la malla de control:

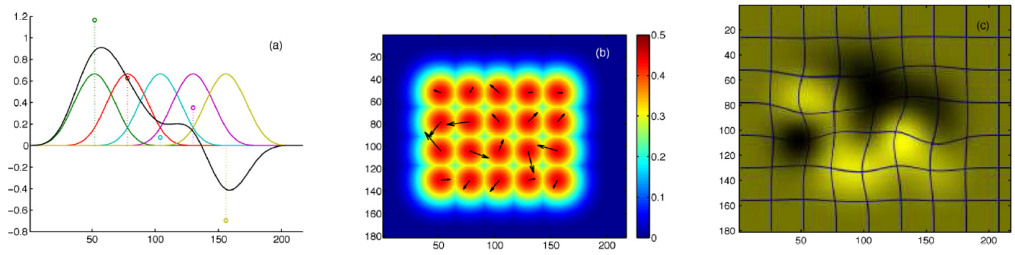


Figura 2.7: Proceso de deformacion basado en B-splines

Los métodos basados en B-spline cúbicas son muy caros computacionalmente. Por esta razón, el enfoque clásico solo optimiza un punto de control a la vez. La computación de la posición de cada píxel y sus nuevas intensidades son totalmente independientes, por lo que su cálculo sería muy adecuado para una implementación en paralelo. Así pues, en este trabajo vamos a optimizar todos los puntos de control de manera simultánea y vamos a interpolar la imagen completa en cada paso; nos beneficiaremos, en consecuencia, de que la computación basada en GPU es tanto más eficiente cuanto mayor sea el número de puntos procesados en paralelo.

La computación basada en GPU es más eficiente cuantos más datos se procesan al mismo tiempo y lo que se va a hacer es optimizar todos los puntos de control e interpolar la imagen completa en cada paso.

Capítulo 3

Programación en GPU

3.1 Introducción a la programación en GPU

La GPU es un término acuñado por nVidia para denotar que sus aceleradoras 3D no solo eran capaces de calcular geometría y mapear texturas de una escena 3D. Son la evolución de las tarjetas aceleradoras 3D, que eran meros rasterizadores 2D.

- La CPU está pensada para procesar flujos de control complejos que manejan datos secuencialmente. Sus características principales son:
  - *Branch prediction*: predicción de saltos condicionales, que introducen un retardo en los procesadores.
  - Ejecución *out-of-order*: es un paradigma utilizado en la mayoría de los microprocesadores de alto rendimiento como forma de aprovechar los ciclos de instrucción que de otro modo serían desperdiciados produciéndose cierta demora de trabajo [20]
  - *Pipeline* (tuberías virtuales) no demasiado profundo, esto es, no habrá demasiada segmentación de las instrucciones, en consecuencia, permite una velocidad más baja.
  - Cachés grandes.
  - Muy poco paralelismo, implementado mediante:
    - Extensiones de la arquitectura principal.
    - Recopilación de todo el núcleo de la CPU (*multicore*).

Figura 3.1: CPU: microprocesador (chip).

- La GPU está pensada para procesar flujos de control simples que manejan datos en paralelo. Está orientada a maximizar el *throughput* y sus características principales son:
  - El *pipeline* puede ser muy profundo.
  - Ramificaciones: vaciar el *pipeline* sería muy costoso.
    - Se procesan las dos ramas hasta que se sabe cuál era correcta.
    - Se multiplica el número de recursos ocupados para tomar la decisión.
  - El paralelismo es intrínseco a la arquitectura.

- Cientos, miles de ALU disponibles para cada instrucción.

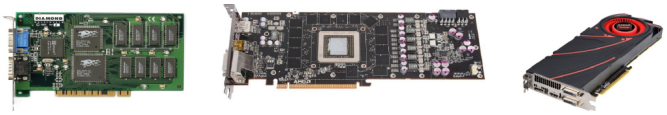


Figura 3.2: GPU: microprocesador + memoria E/S.

La GPU funciona como un subsistema autónomo, con su propio procesador, memoria y periféricos. La CPU envía peticiones y recoge resultados, y tiene un funcionamiento asíncrono, como el resto de subsistemas. La CPU coordina todo el trabajo.

En cuando a la programación en GPU, existen multitud de arquitecturas. Hay tres marcas dominantes: Intel (60 %), nVidia (20 %) y AMD (20 %), y cada marca tiene su propia arquitectura.

Para unificar todas las plataformas de procesamiento, podemos hablar de CUDA y OpenCL, las cuales presentan al usuario una arquitectura abstracta, que ocultan la complejidad de la arquitectura real, simulan lo que la arquitectura real no puede hacer (a costa de tardar más) y facilitan la comunicación entre la GPU y la CPU.

CUDA es propiedad de nVidia y solo funciona en plataformas nVidia. OpenCL es un estándar de *Khronos Group* y funciona en cualquier plataforma (mientras haya driver). Con OpenCL pueden coexistir *drivers* de varias marcas, donde cada uno ve los dispositivos de su propia marca y la CPU (la implementación de nVidia no deja ver la CPU). OpenCL abstrae también la CPU de la máquina (un programa para GPU correrá igualmente en CPU más despacio o más rápido, dependiendo del programa).

Para programar una GPU, hay que seguir una serie de pasos. El procedimiento general será el siguiente:

1. Averiguar qué plataformas están presentes (solo en OpenCL, en CUDA hay una plataforma).
2. Averiguar qué dispositivos están presentes.
3. Enviar a la RAM de la GPU los datos a procesar.
4. Enviar a la GPU los programas a ejecutar en paralelo (normalmente muy cortos y simples).
5. Ordenar a la GPU que ejecute los programas.
6. Esperar a que la GPU acabe de procesar.
7. Traer los datos procesados a la RAM de la máquina [21].

3.2 Terminología de OpenCL

Como vamos a trabajar con OpenCL, vamos a ver una terminología informal de ella:

– **Global Memory:** unión de todas las memorias RAM de los dispositivos (GPU o CPU) seleccionados para trabajar.

– **Constant Memory:** zona de la *global memory* declarada como de solo lectura durante una ejecución concreta.

- **Local Memory:** banco de memoria pequeña (decenas de KB) y muy rápida, típicamente un orden de magnitud más que la *global memory*.

- **Private Memory:** conjunto de registros conectados a una ALU concreta. Típicamente un orden de magnitud más rápida que la *local memory*.

- **Processing Element:** (abstracción de) una ALU. Cada *processing element* tiene su propia *private memory*.

- **Compute Unit:** conjunto de *processing elements* que comparten un banco de *local memory*.

- **Device:** una GPU o la CPU del sistema. Formado por una o varias *compute units*.

- **Platform:** unión de todos los *devices* que puede manejar un determinado *driver*.

- **Kernel:** programas que corren en los *processing elements*, que son funciones escritas en lenguaje OpenCL, que es parecido a C pero con algunas extensiones. Cada *processing element* ejecuta una copia (*instance*) de un kernel, y cada copia opera con datos distintos. OpenCL replica el kernel por cuantos *processing elements* se necesite.

- **Context:** entorno en el que se definen los *kernels* y la *global memory*. Formado por uno o varios *devices*, sus memorias RAM y sus *command queues*.

- **Comand Queue:** objeto que lanza *kernels* en un *device* dado de un *context* concreto.

- **Work Item:** realización (*instance*) de un kernel ejecutándose en un *processing element* concreto.

- **Work Group:** conjunto de *work items* que se ejecutan en una *compute unit* dada (y que son realizaciones del mismo *kernel*).

- **Global ID:** identificador (0...n-1) de un *work item* dado en un *kernel* concreto.

- **Local ID:** identificador (0...n-1) de un *work item* dado en un *work group* concreto de un *kernel* dado.

- **Program:** conjunto de *kernels* de un mismo fichero fuente [21].

La visión de OpenCL se muestra en la figura 3.3:

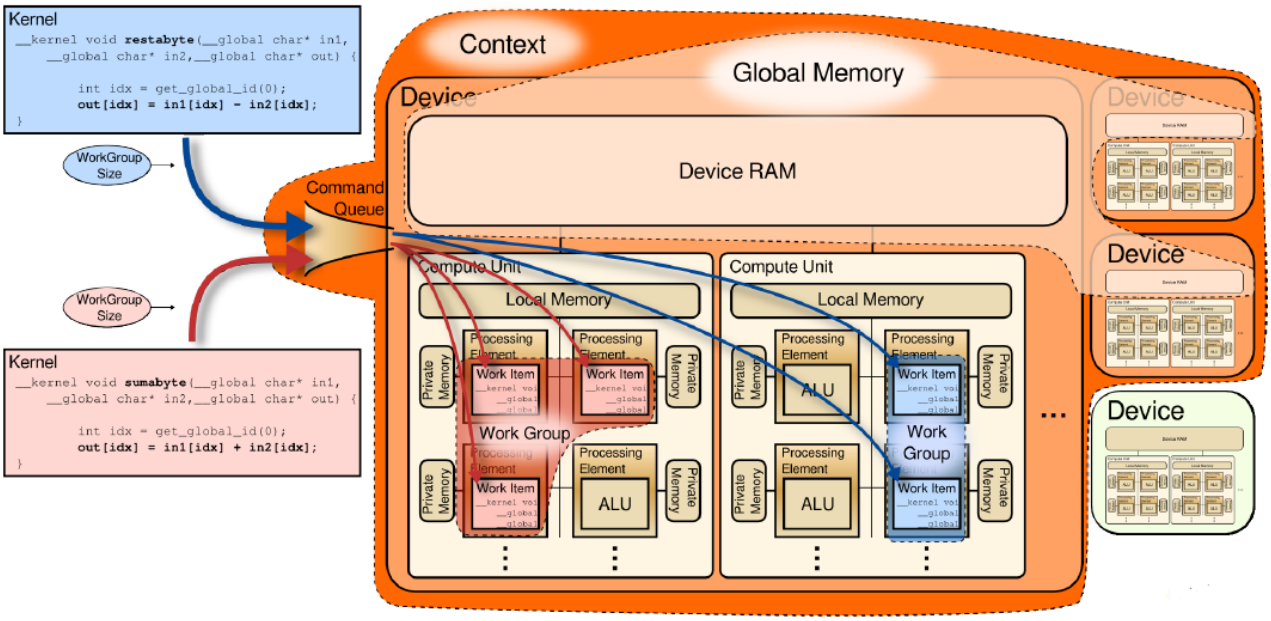
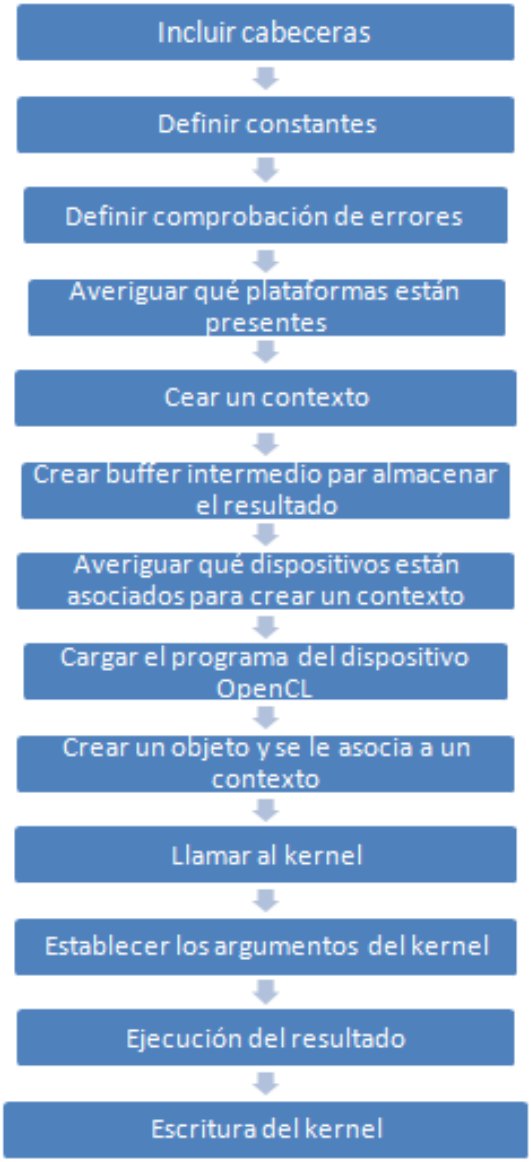


Figura 3.3: Visión de OpenCL.

3.3 Tutorial de OpenCL: "Hello World".

Para escribir un programa complejo en OpenCL, primero hay que tener unos conocimientos básicos. Vamos a explicar una estrategia general para la ejecución de un programa sencillo en C, simplemente que muestre por pantalla *Hello World*.

Para facilitar este primer contacto, se muestra el esquema general que se va a seguir mediante un diagrama:



Comencemos con los ficheros de cabecera, simplemente como en cualquier otra API utilizada en C++, hay que incluir ficheros de cabecera cuando se use la API de OpenCL. Por lo general suele estar en el directorio CL dentro del directorio *include* principal. Para las cabeceras C++ tenemos:

```
#include <utility>
#define __NO_STD_VECTOR // Use cl::vector instead of STL version
#include <CL/cl.hpp>
```

Para nuestro programa de inicialización, vamos a utilizar un pequeño número de cabeceras C++ adicionales que están relacionadas con OpenCL:

```
#include <cstdio>
#include <cstdlib>
#include <fstream>
```

```
#include <iostream>
#include <string>
#include <iterator>
```

Como vamos a solicitar dinámicamente un *device* de OpenCL para devolver la cadena *Hello World*, la definimos como una constante para usar en los cálculos:

```
const std::string hw("Hello World\n");
```

Una propiedad muy común de la mayoría en las llamadas a la API OpenCL es que pueden devolver un código de error (tipo *cl_int*) como resultado de la función, o almacenar el código de error en una ubicación que le pasa el usuario como un parámetro de la llamada. Por simplicidad, definimos una función, *checkErr*, para ver que una cierta llamada se ha completado satisfactoriamente, y en ese caso, OpenCL devuelve el valor *CL_SUCCESS*, en caso contrario, devuelve el código del error y finaliza el programa:

```
inline void checkErr(cl_int err, const char * name)
{
    if (err != CL_SUCCESS) {
    std::cerr << "ERROR: " << name  << " (" << err << ")" << std::endl;
    exit(EXIT_FAILURE);
    }
}
```

El primer paso para inicializar y usar OpenCL es crear un *context*. El resto del trabajo de OpenCL (lo que es la creación de los *devices*, las memorias, compilar y ejecutar los programas) se lleva a cabo dentro de este *context*. Un *context* puede tener varios *devices* asociados (por ejemplo *devices* GPU o CPU). En este ejemplo, vamos a utilizar un único *device*, *CL_DEVICE_TYPE_CPU*, para el *device* CPU. Pero antes de que podamos crear un *context*, hay que definir qué plataformas hay presentes. La clase *cl::Platform* proporciona el método estático *cl::Platform::get* para esto y devuelve una lista de plataformas. Por ahora vamos a seleccionar la primera plataforma y usarla para crear un *context*. El constructor *cl::Context* debe ser satisfactorio, y en este caso, el valor de *err* sería *CL_SUCCESS*:

```
int main(void)
{
    cl_int err;
    cl::vector< cl::Platform > platformList;
    cl::Platform::get(&platformList);
    checkErr(platformList.size()!=0 ? CL_SUCCESS : -1, "cl::Platform::get");
    std::cerr << "Platform number is: " << platformList.size() << std::endl;std::string
        platformVendor;
    platformList[0].getInfo((cl_platform_info)CL_PLATFORM_VENDOR, &platformVendor);
    std::cerr << "Platform is by: " << platformVendor << "\n";
    cl_context_properties cprops[3] = {CL_CONTEXT_PLATFORM,(cl_context_properties)(
        platformList[0])(), 0};
    cl::Context context(CL_DEVICE_TYPE_CPU,
                        cprops,
                        NULL,
                        NULL,
                        &err);
                        checkErr(err, "Context::Context()");
```

Antes de meternos con los *devices*, donde se realiza realmente todo el trabajo, lo primero que vamos a hacer es asignar un *buffer* OpenCL intermedio, para guardar el resultado del *kernel* que se va a ejecutar en el *device*, que será la cadena *Hello World*. Por ahora simplemente se va a asignar algo de memoria en el *host* y vamos a solicitar que OpenCL use ese *buffer* directamente, mediante el *flag CL_MEM_USE_HOST_PTR*:

```
    char * outH = new char[hw.length()+1];
    cl::Buffer outCL(context,
                     CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
                     hw.length()+1,
                     outH,
```

```
                    &err);
                    checkErr(err, "Buffer::Buffer()");
```

En OpenCL muchas operaciones se realizan con respecto a un *context* dado. Por ejemplo, las asignaciones de *buffers* (operaciones 1D) o las imágenes (regiones de memoria 2D y 3D) son todas operaciones del *context*. Pero también existen operaciones específicas de los *devices*, como pueden ser la compilación del programa y la ejecución de los *kernel*. Para realizar dichas operaciones, vamos a hacer uso de la API de C++, mediante *object.getInfo<CL_OBJECT_QUERY>()*:

```
cl::vector<cl::Device> devices;
devices = context.getInfo<CL_CONTEXT_DEVICES>();
checkErr(devices.size() > 0 ? CL_SUCCESS : -1, "devices.size()>0");
```

Ahora ya que tenemos la lista de *devices* asociados para un *context*, en este caso un único *device* de la CPU, se necesita cargar y compilar el programa (el programa que queremos ejecutar en los *devices*). Las primeras líneas del siguiente código simplemente cargan el programa del *device* OpenCL del disco, lo convierten en una cadena, y crean un objeto *cl::Program::Sources* utilizando el constructor de ayuda. Dado un objeto del tipo *cl::Program::Sources* un *cl::Program*, se crea un objeto y se le asocia a un *context*. A continuación, se compila para un determinado conjunto de *devices*:

```
std::ifstream file("lesson1_kernels.cl");
checkErr(file.is_open() ? CL_SUCCESS:-1, "lesson1_kernel.cl");
std::string prog(
std::istreambuf_iterator<char>(file),
(std::istreambuf_iterator<char>()));
cl::Program::Sources source(1,
                            std::make_pair(prog.c_str(), prog.length()+1));
                            cl::Program program(context, source);
                            err = program.build(devices,"");
                            checkErr(err, "Program::build()");
```

Un programa dado puede tener varios hilos, llamados *kernels*, y para llamar a un *kernel* debemos compilar un *kernel object*. Se supone que no existe un mapeo directo entre los nombres de los *kernel*, representados como una cadena, a una función con el atributo *__kernel* en el programa que se va a ejecutar. En este caso, podemos compilar un *kernel cl::kernel object*. Los argumentos del *kernel* se establecen usando la API de C++ *kernel.setArg()*, que toma el ínice y el valor para un argumento en particular:

```
cl::Kernel kernel(program, "hello", &err);
checkErr(err, "Kernel::Kernel()");err = kernel.setArg(0, outCL);
checkErr(err, "Kernel::setArg()");
```

Ahora que el código base está hecho, es hora de ejecutar el resultado (el *buffer* de salida con la cadena *Hello World*). Todos los cálculos de los *devices* se hacen mediante un *command queue*, que es una interfaz virtual para el *device* en cuestión. Cada *command queue* tiene un mapeo uno a uno con un *device* determinado, que se crea con el *context* asociado mediante una llamada al constructor de la clase *cl::CommandQueue*. Dado un *queue cl::CommandQueue*, los *kernel* pueden ser encolados mediante *queue.enqueueNDRangeKernel*. Esto encola un *kernel* para ejecutarlo en un *device* asociado. El número total de elementos en el dominio de ejecución se llama *global work size*, y el número de elementos individuales se llaman *work-items*. El *kernel* se puede ejecutar en un dominio de ejecución 1D, 2D ó 3D (no se le puede pasar al *kernel* un *work_size* de más de 3D) en paralelo. Los *work items* se pueden agrupar en *work-groups* cuando se necesite que se comuniquen entre los *work-items*. Los *work-groups* se definen con una función de subíndice (llamada *local work size*), que define el tamaño de cada dimensión correspondiente a las dimensiones del dominio de ejecución (*global work size*).

Respecto a la ejecución de los *kernel*, hay mucho que considerar, pero por ahora vamos a señalar que para el *kernel* de *Hello World*, cada *work-item* se encarga de una letra de la cadena resultante, y

es suficiente con ejecutar *hw.length()+1*, donde *hw* es la constante *std::string* que habíamos definido al principio del programa. Necesitamos un *work-item* extra para el término nulo (de ahí el +1):

```
cl::CommandQueue queue(context, devices[0], 0, &err);
checkErr(err, "CommandQueue::CommandQueue()");cl::Event event;
err = queue.enqueueNDRangeKernel(kernel,
                                 cl::NullRange,
                                 cl::NDRange(hw.length()+1),
                                 cl::NDRange(1, 1),
                                 NULL,
                                 &event);
checkErr(err, "ComamndQueue::enqueueNDRangeKernel()");
```

El último argumento para la llamada *enqueueNDRangeKernel* anterior era un objeto *cl::Event*, que se puede utilizar para consultar el estado del comando con el que está asociado (por ejemplo, se ha completado). Es compatible con el método *wait()*, que bloquea hasta que el comando se ha completado. Esto es necesario para asegurar que el *kernel* ha terminado de ejecutarse antes de leer el resultado en la memoria del *host* con *queue.enqueueReadBuffer()*. Con el resultado del cálculo de nuevo en la memoria del *host*, es simplemente cuestión de sacar el resultado y salir del programa:

```
    event.wait();
    err = queue.enqueueReadBuffer(outCL,
                                  CL_TRUE,
                                  0,
                                  hw.length()+1,
                                  outH);
    checkErr(err, "ComamndQueue::enqueueReadBuffer()");
    std::cout << outH;
    return EXIT_SUCCESS;
}
```

Finalmente, para dar el programa por completo y se ejecute correctamente, hay que definir el *kernel* (lesson1_kernels.cl). La implementación del *kernel* es sencilla: se obtiene un único índice en la función (puesto que el *global work size* tiene dimensión 1D) usando *get_global_id()*, y escribe su valor en la variable de salida [22]:

```
#pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable
__constant char hw[] = "Hello World\n";
__kernel void hello(__global char * out)
{
    size_t tid = get_global_id(0);
    out[tid] = hw[tid];
}
```

Capítulo 4

Implementación

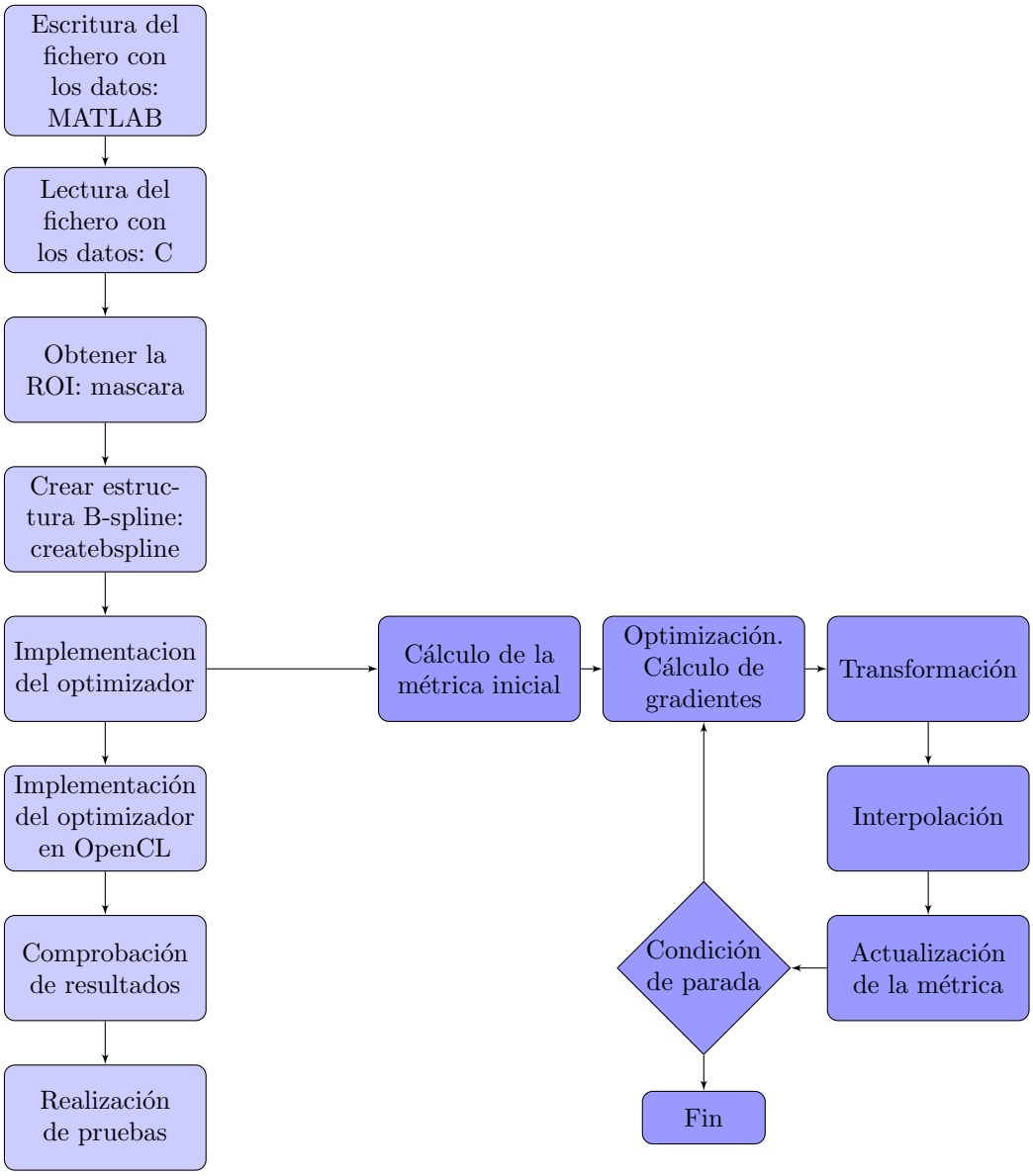
4.1 Implementación en código C y OpenCL

La implementación del código no ha sido realizada en su totalidad en OpenCL, sino solamente las funciones con mayor carga computacional. El resto del código ha sido implementado en código C.

Para la implementación del código, se han realizado varias fases:

- La primera fase que se va a relizar es el preprocesado:
  - En una primera tarea se ha realizado la escritura del fichero con los datos de vistas en eje corto del corazón utilizando MATLAB (ya que serán ficheros *.mat*). Posteriormente ha utilizado el código C necesario para cargar esos datos en memoria.
  - El segundo paso ha sido la implementación de las funciones de preprocesado en código C:
    - Primero se han definido la ROI y los parámetros necesarios mediante la función **mascara**.
    - Se ha obtenido la estructura de la B-spline, que es donde se almacenan las variables relacionadas con las B-spline. Se ha realizado mediante la función **createbsplines**.
- En una segunda fase se ha realizado el optimizador, que es donde se realiza el registrado en sí mediante la función **optimizar**. Dentro del optimizador tenemos varias tareas:
  - Cálculo de la métrica inicial, que obtiene el valor de la métrica antes de empezar el proceso iterativo. Lo realizaremos con la función **metrica**.
  - Cálculo de gradientes, que se realizará mediante la función **gradiente**.
  - Realización de la transformación, que se realizará mediante la función **transformar**
  - Realización de la interpolación, que se realizará mediante la función **interpolar**.
  - Actualización de la métrica, que se realizará mediante la función **metrica**.
  - Comprobación de la condición de parada, y si no la cumple, vuelve al cálculo de los gradientes.
- Implementación en OpenCL en las funciones de mayor carga computacional.
- Por último será la realización de pruebas, tanto para la comprobación como para la visualización de los tiempos de ejecución.

Mediante un diagrama se muestra el esquema que se ha seguido para la implementación:



4.1.1 Escritura del fichero con los datos

Lo primero que tenemos que tener en cuenta, es que en MATLAB los datos se representan por defecto de tipo *double*, aunque es posible que, dependiendo del origen de estos datos, solo tomen valores enteros. Es importante que en el proceso se conserven los datos originales sin pérdida de información. En el caso particular puesto como ejemplo a continuación, los datos toman únicamente valores enteros comprendidos entre 0 y 1023 (10 bits de precisión numérica). Sin embargo, y dado que cuando trabajemos en C manejaremos los datos con precisión *float*, haremos la conversión a este tipo ya en el momento de volcar los datos a un archivo binario.

Se carga en MATLAB el fichero *.mat*:

```
>> load('../datos/INFO.mat');
```

Los datos a guardar se encuentran en la variable 'IC', de tamaño ROWS x COLS x SLICES x FRAMES. Se vuelcan únicamente los datos correspondientes a un *slice* con precisión *float*. Para ello bastará con escribir:

```
>> fid = fopen('../datos/datos.bin','wb')
>> slice = 7;
>> fwrite(fid, IC(:,:,slice,:), 'single')
>> fclose(fid);
```

4.1.2 Lectura del fichero con los datos

Ahora solo falta cargar el fichero en C, donde el tamaño de los datos ha sido obtenido de antemano:

```
float *IC = (float *)malloc(sizeof(float) * ROWS * COLS * FRAMES);
FILE * fid = fopen("../datos/datos.bin", "rb");
fread(IC, sizeof(float), ROWS*COLS*FRAMES, fid);
fclose(fid);
```

En cuanto al manejo de los datos en C y acceso a los elementos de los mismos existen dos opciones:

- Empleando arrays multidimensionales.
- Empleando arrays unidimensionales.

En el primer caso, se debe construir un array multidimensional con las dimensiones correspondientes. Para ello, hay que reservar memoria de forma iterativa para cada una de las dimensiones (temporal, vertical y horizontal). En el ejemplo siguiente se inicializa además con los datos previamente cargados en IC:

```
float*** IC_3D = (float***)malloc(sizeof(float**)*ROWS);
for(int i = 0; i < ROWS; i++)
{
    IC_3D[i] = (float**)malloc(sizeof(float*)*COLS);
    for(int j = 0; j < COLS; j++)
    {
        IC_3D[i][j] = (float *)malloc(sizeof(float)*FRAMES);
        // Aprovechamos para inicializar los datos
        for(int n = 0; n < FRAMES; n++)
        {
            IC_3D[i][j][n] = IC[i + j * ROWS + n * ROWS * COLS];
        }
    }
}
```

En el ejemplo, hemos calculado qué posición en el array 'IC' corresponde con la posición del array tridimensional dada por los índices i, j, n. Para ello, se emplea la fórmula general para un array N-dimensional con un vector de tamaños *tam* cargado previamente (tam sería en tamaño de la matriz de datos):

$$indice(a_0, a_1, a_2..., a_{N-1}) = a_0 + a_1 * tam[0] + a_2 * tam[0] * tam[1] + ... + a_{N-1} * tam[0] * ... * tam[N-2] \quad (4.1)$$

Este método tiene la ventaja de poder acceder a los elementos del array mediante los índices de fila, columna, etc. Por ejemplo, IC_3D[i][j][n]. Como contrapartida, la gestión de la memoria es más compleja. Además, debe tenerse en cuenta que no es posible garantizar que los datos ocupen posiciones contiguas de memoria, por lo que las operaciones de copia, volcado a archivo o envío a la GPU deben realizarse con cuidado.

Una segunda posibilidad es almacenar todos los datos en un único array unidimensional. En este caso, la memoria se reserva una única vez como un bloque contiguo. En el siguiente ejemplo es lo que se ha realizado para cargar los datos en memoria. Se repiten aquí esas líneas por completitud:

```
float *IC = (float *)malloc(sizeof(float) * ROWS * COLS * FRAMES);
fread(IC, sizeof(float), ROWS*COLS*FRAMES, fid);
```

En este caso, cada vez que se quiera acceder a un elemento del array tendremos que emplear la fórmula de la ecuación (4.1). Para facilitar las cosas, se puede implementar dicha fórmula en una función para arrays 2D, 3D, etc.

En el presente proyecto se han empleado arrays unidimensionales, puesto que es más recomendable para la posterior implementación en OpenCL, la reducción del código es bastante notable y la carga computacional se reduce considerablemente, puesto que el empleo de arrays unidimensionales utiliza bloques contiguos de memoria a los que es de más fácil acceso.

4.1.3 Creación de la ROI

Una vez obtenidos los datos, es hora de implementar el registrado. Lo primero que se va a hacer es realizar la función **mascara**, la cual tiene como parámetro de entrada el radio de la ROI introducido por el usuario para que cubra la zona de mayor actividad cardiaca. La función tendrá como parámetros de salida:

- **X:** una matriz de unos y ceros, donde los unos indican el interior de la ROI y los ceros el exterior.
- **areaX:** un número que indica el área del círculo correspondiente a la suma de todos los elementos de la matriz.
- **caja:** una caja que representa una *bouding box* sobre el círculo que le circunscribe.

Se puede ver una representación en la figura 4.1:

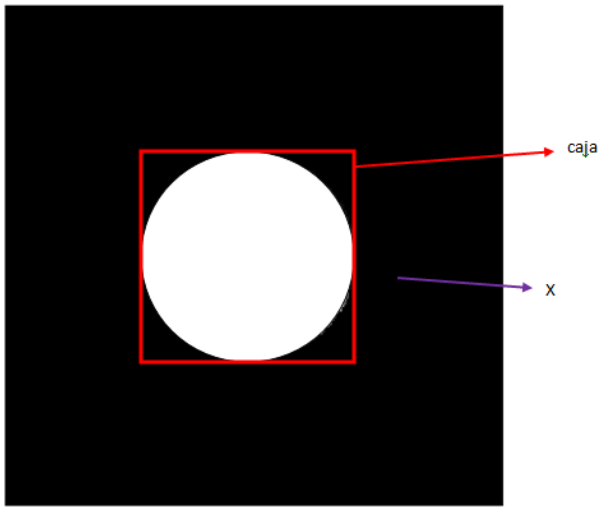


Figura 4.1: Representación de la ROI y la *bounding box*.

La función **mascara** se encarga de la creación de la ROI, para posteriormente realizar el registrado en ella, que será la zona de mayor actividad, reduciendo la carga computacional considerablemente.

4.1.4 Cálculo de la estructura B-spline

Una vez obtenida la máscara, la otra parte del preprocesado va a ser la obtención la estructura B-spline, que va a ser una estructura donde se almacenan las variables relacionadas con las B-spline.

La estructura B-spline, tendrá los siguentes parámetros:

- **pu:** posiciones de los puntos de control.
- **BB y BB1:** matriz de productos B-spline.
- **coef:** coeficientes con offset.
- **BBg y BB1g:** matriz de productos B-spline para el cálculo de los gradientes.
- **coefg:** coeficientes para el cálculo de los gradientes.

En el capítulo 2 se habló de las B-splines de una manera más detallada.
Hay que destacar que los puntos de control no representan los píxeles de la imagen (habrá muchos píxeles con diferentes valores de intensidad entre cada punto de control).

4.1.5 Optimizador

En primer lugar, para el diseño del optimizador se establecerán las coordenadas del espacio de la imagen de referencia 'x', la cual representa las coordenadas de los píxeles de dicha imagen.

Para describir el optimizador, se consideran dos funciones previas, a saber, una función que implemente la transformación, (la función **transformar**) y otra que implemente la interpolación (la función **interpolar**) por simplicidad en la exposición:

- **Transformación:** La función **transformar** tiene como parámetros de entrada la matriz de transformaciones (T) y las coordenadas de los píxeles de la imagen (x) y como parámetro de salida las coordenadas de los píxeles de la nueva imagen deformada (xn). Esta función se encarga de transformar cada punto 'x' en la imagen de referencia al punto que le corresponde en cada imagen de partida (Tn(x), que es el equivalente a 'xn' en el código). Esta transformación es realizada mediante la FFD, empleando las B-spline para la deformación de los puntos de control.
- **Interpolación:** La función **interpolar** tiene como parámetros de entrada 'xn' y la imagen a la que se le va a aplicar la deformación (In), y se encarga de obtener la imagen transformada (IT) empleando interpolación. El motivo de esta interpolación es la necesidad de hallar el valor de la imagen en los nuevos píxeles, esto es, el valor de In(Tn(x)); y como probablemente Tn(x) no coincida con los valores donde se encuentran los puntos de la imagen, se debe interpolar.

  Se realizará una interpolación bilineal, cuyos coeficientes se describen en la ecuación 4.2 y representan los pesos asociados al desplazamiento de los cuatro píxeles vecinos del punto de coordenadas Tn(x) que se va a interpolar.

$$\begin{aligned} \omega_{00} &= (1-\alpha)(1-\beta) \\ \omega_{01} &= (1-\alpha)\beta \\ \omega_{10} &= \alpha(1-\beta) \\ \omega_{11} &= \alpha\beta \end{aligned} \tag{4.2}$$

  Donde α representa la distancia entre los puntos en el eje x y β a la distancia entre los puntos en el eje y. (x',y') son las coordenadas de los nuevos píxeles y (x,y) las coordenadas de los píxeles iniciales:

$$\begin{aligned} \alpha &= x' - x \\ \beta &= y' - y \end{aligned} \tag{4.3}$$

En la figura 4.2 se muestra de forma gráfica el significado de los parámetros anteriores.

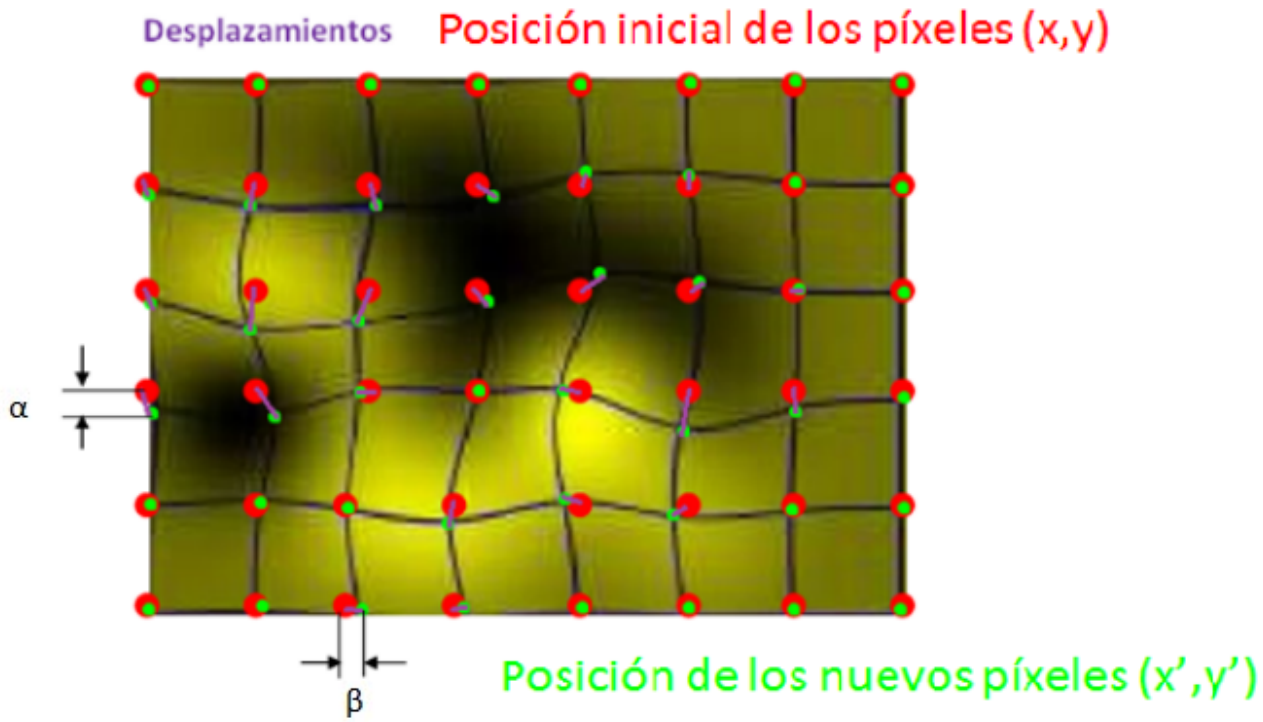


Figura 4.2: Manipulación de los puntos de control.

Una vez implementadas ambas funciones ya se puede realizar el optimizador sin demasiada complejidad. La estrategia de optimización se explicó con más detalle en las secciones 2.3 y 2.4 del documento. Como ya se ha mencionado, el optimizador además de implicar la transformación y la interpolación, implica el cálculo de los gradientes a partir de la métrica (cuyo cálculo se detalla en sección 2.4) y se realizará de manera iterativa. Se actualiza el valor de la métrica en cada iteración y existe un criterio de parada (del que también se ha hablado en la sección 2.4).

El esquema general del optimizador nos queda como se muestra en la figura 4.3:

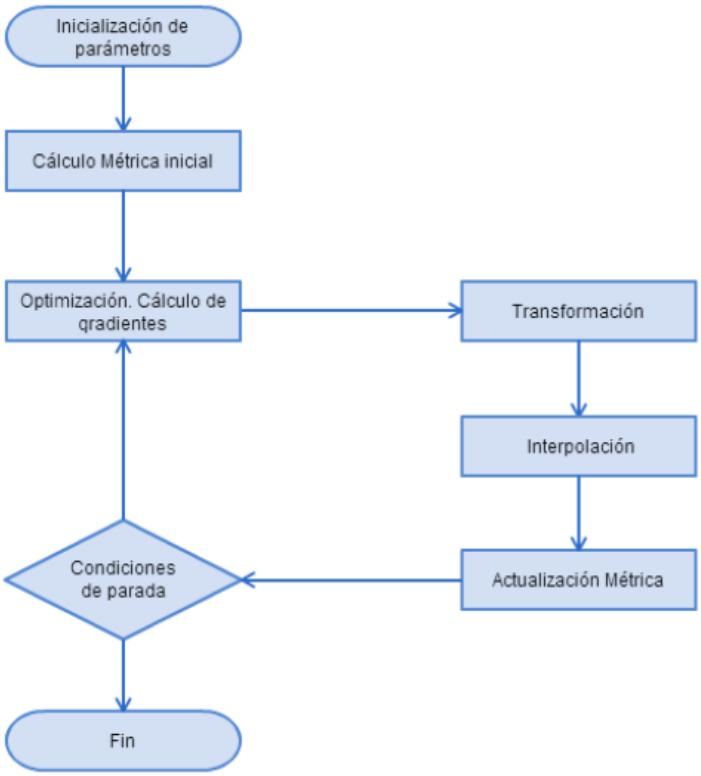
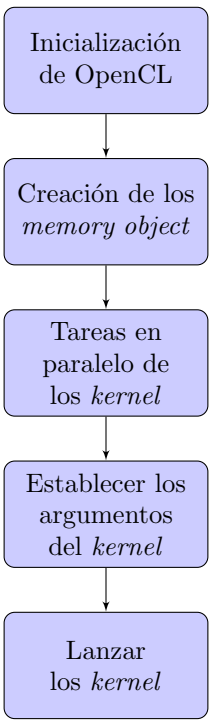


Figura 4.3: Esquema general del registrado.

En el optimizador, también se calculará el coste, se actualizará la matriz de las transformaciones y se actualizará el valor de los pesos tal y como se comentó en la sección 2.4 del presente documento. La implementación en OpenCL se realizará sobre el optimizador descrito, que será el proceso de mayor carga computacional.

4.1.6 Implementación del código

Por último, se va a implementar el optimizador mediante OpenCL. Para ello, se van a seguir una serie de fases para su correcta implementación, las cuales están representadas en el siguiente diagrama:



- Inicialización de OpenCL:

  En una primera fase, se seguirá el procedimiento general para programar una GPU, el cual se explicó con detalle en la sección 3.3. Por completitud, se mostrará el código empleado en el presente proyecto:

```
cl_program program = NULL;
cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret;

FILE *fpcl;

const char fileName[] = "./kernels.cl";
size_t source_size;
char *source_str;

//Load kernel source file
fpcl = fopen(fileName, "rb");
if (!fpcl) {
        fprintf(stderr, "Failed to load kernel.\n");
        exit(1);
}
source_str = (char *)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fpcl);
fclose(fpcl);


//Get platform/device information
ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
```

```
CL_CHECK(ret);

ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT,
    1, &device_id, &ret_num_devices); CL_CHECK(ret);

//Create OpenCL Context
context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
CL_CHECK(ret);

//Create command queue
command_queue = clCreateCommandQueue(context, device_id,
        CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &ret);
CL_CHECK(ret);

// Create kernel from source
program = clCreateProgramWithSource(context, 1,
    (const char **)&source_str, (const size_t *)&source_size, &ret);
CL_CHECK(ret);
cout << "Creacion del programa OK " << endl;

//Build
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
CL_CHECK(ret);
cout << "Compilacion del programa OK " << endl;
```

- Creación de los objetos de memoria:

  En una segunda fase, se han de crear los objetos memoria necesarios para el almacenamiento temporal de las variables a utilizar. Se mostrará un ejemplo para la creación de un objeto de memoria (los demás serán creados e inicializados de la misma manera):

```
dVmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, total_length* sizeof(float),
    NULL, &ret); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, dVmobj, CL_TRUE, 0, total_length *sizeof(
    float), dV,      0, NULL, NULL); CL_CHECK(ret);
```

- Creación de las tareas en paralelo de los *kernel*:

  Para la ejecución de los *kernel* se ha decidido utilizar el método de la creación de tareas en paralelo, utilizando el *kernel* como un vector, el cual tiene asociado a cada elemento una función. A continuación se muestran las funciones asociadas a cada elemento del *kernel* en el presente proyecto, las cuales están descritas en el Anexo. Se crean las tareas en paralelo del kernel OpenCL, donde se indica la función asociada a cada elemento del vector kernel:

```
kernel[0] = clCreateKernel(program, "inicVect", &ret); CL_CHECK(ret);
kernel[1] = clCreateKernel(program, "inicVect", &ret); CL_CHECK(ret);
kernel[2] = clCreateKernel(program, "inicVect", &ret); CL_CHECK(ret);
kernel[3] = clCreateKernel(program, "inicVect", &ret); CL_CHECK(ret);
kernel[4] = clCreateKernel(program, "inicVect3", &ret); CL_CHECK(ret);
kernel[5] = clCreateKernel(program, "inicVect3", &ret); CL_CHECK(ret);
kernel[6] = clCreateKernel(program, "gradiente_imagen", &ret); CL_CHECK(ret);
kernel[7] = clCreateKernel(program, "gradiente", &ret); CL_CHECK(ret);
kernel[8] = clCreateKernel(program, "coste1", &ret); CL_CHECK(ret);
kernel[9] = clCreateKernel(program, "proyeccion", &ret); CL_CHECK(ret);
kernel[10] = clCreateKernel(program, "transformar", &ret); CL_CHECK(ret);
kernel[11] = clCreateKernel(program, "transformar2", &ret); CL_CHECK(ret);
```

```
kernel[12] = clCreateKernel(program, "interpolar", &ret); CL_CHECK(ret);
kernel[13] = clCreateKernel(program, "metrica", &ret); CL_CHECK(ret);
```

En el Anexo se mostrará el código completo para dicha implementación, donde también se mostrará el fichero kernels.cl.

- Establecimiento de los argumentos del *kernel*:

  Los argumentos del *kernel* se establecen usando la API de C++ *kernel.setArg()*, que toma el ínice y el valor para un argumento en particular. A continuación se muestra el establecimiento de un *kernel* en particular, puesto que en los demás *kernel* los argumentos se establecerán de igual forma:

```
ret = clSetKernelArg(kernel[9], 0, sizeof(cl_mem), (void *)&proyH2mobj); CL_CHECK(
    ret);
ret = clSetKernelArg(kernel[9], 1, sizeof(cl_mem), (void *)&Tmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[9], 2, sizeof(cl_mem), (void *)&Difmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[9], 3, sizeof(cl_mem), (void *)&H2mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[9], 4, sizeof(cl_mem), (void *)&media2mobj); CL_CHECK(
    ret);
ret = clSetKernelArg(kernel[9], 5, sizeof(cl_mem), (void *)&Wnmobj); CL_CHECK(ret);
```

- Lanzamiento de los *kernel*:

  Por último se realizará el lanzamiento de los *kernel*. En OpenCL, el procedimiento a realizar sería encolar los *kernel* para ejecutarlos en un *device* asociado. En la sección 3.3 se explica este proceso de forma detallada, el cual se realiza mediante *queue.enqueueNDRangeKernel*:

```
ret = clEnqueueNDRangeKernel(command_queue, // cl_command_queue command_queue
kernel[7],                 // cl_kernel kernel
3,                                 // cl_uint work_dim
NULL,                      // const size_t *global_work_offset
(size_t *)&total_size_dV,  // const size_t *global_work_size
NULL,                       // const size_t *local_work_offset
0,                                 // cl_uint num_events_in_wait_list
NULL,                       // const cl_event *event_wait_list
NULL                       // cl_event *event
);
CL_CHECK(ret);
```

Cabe destacar que tanto la función que va acumulando el coste en cada iteración como la función que actualiza los valores de la métrica, no han sido implementados en OpenCL, debido a dos razones: la primera es que dichas funciones operan con arrays relativamente pequeños, y la carga computacional de éstas es despreciable, y la segunda razón es que al ser funciones que van acumulando valores, es tarea difícil en OpenCL, puesto que al trabajar con datos en paralelo, cada vez que se ejecute el *kernel* no acumulará datos, sino que sumará solamente el hilo más rápido, lo que nos dará un valor prácticamente al azar.

4.2 Consejos para la implementación

En primer lugar, como ya se ha mencionado anteriormente, hay que tener especial cuidado en los tipos de datos que se van a leer y escribir, ya que si no se utiliza una precisión adecuada, por ejemplo, volcando a un fichero valores con una precisión de un byte (de 0 a 255), si esos datos tienen valores mayores, se está perdiendo información.

Otro detalle importante es el uso de memoria dinámica, puesto que trabajando con arrays de tan elevado número de elementos, hay un momento en el que, si se definen tamaños fijos de memoria, se truncan los datos a partir de ahí, y se pierde gran parte de la información.

Cuando se tiene un puntero como argumento de una función (argumento de E/S), y la variable a la que apunta está declarada con tamaño dinámico, en los argumentos de la función se ha de declarar como otro puntero más, es decir, si se tiene un vector dinámico (de una dimensión) y se pasa a una función como argumento de E/S, en la declaración de los argumentos de esa función hay que declararlo como puntero doble, y cuando se utilice dicha variable, se ha de apuntar a la primera dirección de memoria (su empleo sería: (*variable)). Se va a mostrar un ejemplo para que se vea más claro.

Primero se declara la variable como un vector dinámico y se pasa a la función *function* como argumento de E/S:

```
float *variable=(float *)malloc(ROW*COL*FRAME*sizeof(float ));
function(&variable);
```

En la función *function* se va a igualar la posición 4 del vector a 0: *function* como argumento de E/S:

```
void function(float **variable);
{
    (*variable)[3]= 0;
}
```

En cuanto a la comparación de los resultados, una función que ha sido de gran ayuda ha sido la que representa un histograma (función *histogram* de MATLAB), la cual muestra una representación gráfica de una variable en forma de barras, donde la superficie de cada barra es proporcional a la frecuencia de los valores representados. Lo que se ha hecho ha sido la escritura de un fichero binario en C con la variable a comparar, la lectura de ésta en MATLAB, y la comparación con la variable obtenida en MATLAB. Se muestra en el siguiente ejemplo.

Se escribe la variable que se desea comparar en un fichero binario en C:

```
FILE * fid;
fid = fopen("ruta_fichero","wb");
if (fid == NULL) { printf("Error al abrir archivo de volcado" ); exit(-1); }
for(int i = 0; i < n_elementos; i++)
fwrite(&(variable[i]), sizeof(float), 1, fid);
fclose(fid);
```

Se lee la variable en MATLAB y se compara mediante el uso del histograma:

```
fileID = fopen('fichero.bin');
variable_C = fread(fileID, inf,'single'); %precision float
fclose(fileID);
hist(variable_C(:)-variable_,MATLAB(:),1000)
```

Si ambos valores son iguales, se tendría que representar una barra con todos los valores a 0, algo parecido a la figura 4.4, aunque se puede tolerar un cierto error que puede ser debido a la conversión de datos de doble precisión (*double*) empleados en MATLAB a datos de precisión simple (*float*) empleados en C:

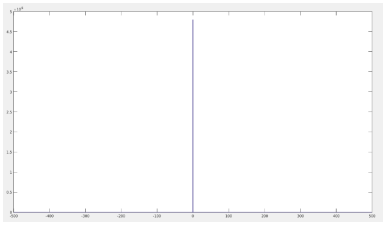


Figura 4.4: Caso ideal de la comparación de dos variables.

Si la variable obtenida no está del todo bien, es decir, que muchos valores coinciden y algunos no, nos puede ser de gran ayuda el uso de la representación visual que nos ofrece MATLAB, con funciones como *imagesc* y *colorbar*, las cuales pueden dar una ayuda visual de qué se está realizando mal.

En cuanto a OpenCL, se ha de tener especial cuidado de no sobreescribir datos, ya que se ejecuta todo en paralelo, y una operación con la misma variable realizada dos veces sobreescribirá el hilo más lento y por consiguiente se ha de prestar atención a las operaciones acumulativas.

Capítulo 5

Resultados

5.1 Comprobación de resultados

En el experimento se van a comparar los resultados obtenidos mediante la escritura en MATLAB de éstos, almacenando en una variable las imágenes transformadas en MATLAB y en otra las obtenidas en C y OpenCL.

Para realizar la comparativa de los datos se hace uso de los percentiles (comando *prctile* de MATLAB) de la resta del valor absoluto de ambas variables antes mencionadas. En la tabla 5.1 se muestra el resumen de la cuantificación de los errores, en la cual se muestra el cálculo de los percentiles 50, 75, 90, 95 y 99 para datos con resoluciones de 400x400x30, 288x288x30 y 256x256x30 píxeles.

Tabla 5.1: Resumen de la cuantificación de errores mediante empleo de percentiles

		percentiles				
tamaño de los datos		**50**	**75**	**90**	**95**	**99**
	400x400x30	0	0	1.8086e-05	1.5740e-04	7.9698e-04
	288x288x30	0	0	1.2258e-04	3.3871e-04	0.0012
	256x256x30	2.1175e-07	7.3364e-06	1.0919e-04	2.6481e-04	9.2074e-04

En la tabla 5.1 se observa que el error es tolerable debido al reducido valor de los percentiles, el cual es nulo o próximo a 0 en todos los casos.

Por otra parte, también es posible realizar una representación visual de los errores en escala de grises, haciendo uso del comando *implay* de MATLAB, efectuando la resta de ambas imágenes transformadas a lo largo de la secuencia y aplicando una normalización. En consecuencia se observa en la figura 5.1 que visualmente el error es despreciable debido a la ausencia de puntos grises.

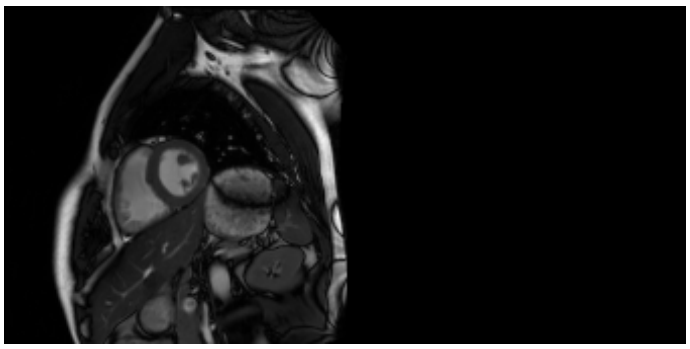


Figura 5.1: Imagen transformada (izquierda) y resta normalizada de ambos resultados (derecha).

Cabe destacar que debido a la ausencia del suavizado se producen pequeños desalineamientos en la ROI. En la figura 5.2 se muestra un pequeño desalineamiento, el cual ha sido explicado con más profundidad en la sección 2.4 del presente documento.

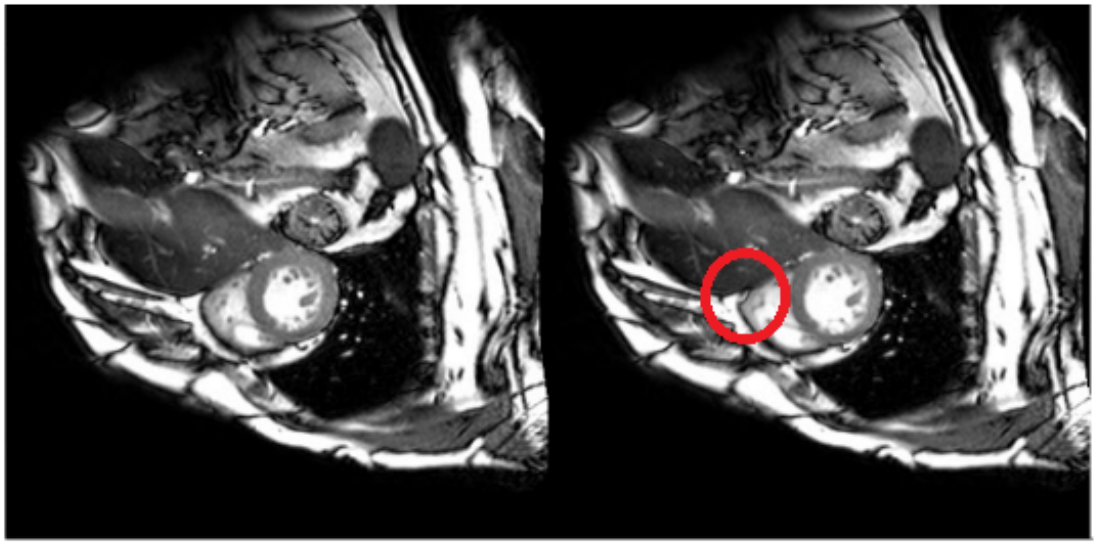


Figura 5.2: Desalineamientos por ausencia del suavizado.

5.2 Tiempos de ejecución

Para la evaluación y diseño de los métodos se dispone de un conjunto de imágenes de MR cardiaca de varios pacientes y las correspondientes segmentaciones manuales de los ventrículos. La adquisición de MR de interés viene dada por las imágenes en 2D de los cortes del corazón. Cada corte forma una imagen en 2D. Los datos están almacenados en ficheros *.mat*, por lo que se tendrá que hacer uso de MATLAB para su obtención.

Se tiene que tener en mente, que el objetivo principal de este proyecto es la mejora de tiempos de ejecución, y el principal objeto de estudio será la comparación de los diferentes tiempos de ejecución utilizando MATLAB, C y OpenCL. Para ello se han medido los tiempos de ejecución del algoritmo en MATLAB, C y OpenCL como función de diversos parámetros del problema, a saber, el radio de la

ROI (columna etiquetada como *radio* en las tablas 5.2, 5.3 y 5.4), el orden de las B-splines (columna etiquetada como *E*), la separación de los puntos de control (columna etiquetada como *Dp*), valor de los pesos (columna etiquetada como *W*) y el número máximo de iteraciones (columna etiquetada como *nmáx*).

Los experimentos se han realizado 10 veces de forma que los resultados de tiempo de ejecución se muestran (en segundos) como media (columna *media*) y desviación típica (columna *std*). Adicionalmente se muestra el valor de la métrica para las ejecuciones en las tres programaciones (columna *métrica*). La métrica es la obtenida en cualquiera de las veces que se ha realizado el experimento, puesto que esta no varía de un experimento a otro.

Tabla 5.2: Imágenes de resolución 400x400 píxeles

radio	**Dp**	**E**	**W**	**nmáx**	**MATLAB**			**C**			**OpenCL**		
					media	**std**	**métrica**	**media**	**std**	**métrica**	**media**	**std**	**métrica**
70	3,3	3	5	60	120.44	1.87	873390	45.60	0.53	873494	3.61	0.05	873268
70	4,4	2	5	60	67.39	0.36	1884321	28.91	0.63	1884312	2.52	0.08	1884312
70	4,4	3	3	60	95.51	0.29	1107600	47.16	0.35	1107627	3.63	0.035	1107654
70	4,4	3	5	30	49.41	0.08	1280300	23.81	0.31	1280302	2.24	0.03	1280302
70	4,4	3	5	60	98.15	0.80	1093000	46.99	0.71	1093006	3.67	0.04	1092998
70	4,4	3	5	80	125.92	0.26	1050100	63.84	0.95	1050137	4.53	0.04	1050145
70	4,4	3	7	60	95.32	0.15	1091600	47.52	1.02	1091637	3.61	0.03	1091637
70	5,5	3	5	60	85.94	0.17	1321700	46.66	0.52	1321715	3.67	0.44	1321715
90	4,4	3	5	60	140.11	0.37	1476800	70.14	1.12	1425250	5.13	0.04	1477839

Tabla 5.3: Imágenes de resolución 288x288 píxeles

radio	**Dp**	**E**	**W**	**nmáx**	**MATLAB**			**C**			**OpenCL**		
					media	**std**	**métrica**	**media**	**std**	**métrica**	**media**	**std**	**métrica**
70	3,3	3	5	60	110.20	1.04	3750954	44.59	0.73	3750930	3.52	0.08	3762817
70	4,4	2	5	60	59.84	0.20	4243200	26.70	0.41	4243245	2.39	0.08	4243241
70	4,4	3	3	60	87.55	0.22	4335400	45.90	0.49	4335395	3.45	0.05	4335413
70	4,4	3	5	30	44.67	0.14	4943600	23.18	0.29	4943626	2.10	0.02	4943623
70	4,4	3	5	60	91.51	1.59	4313272	45.51	0.79	4313272	3.46	0.06	4313274
70	4,4	3	5	80	119.33	0.23	4079000	60.53	0.67	4079018	4.37	0.05	4078940
70	4,4	3	7	60	87.58	0.22	4250200	45.63	0.89	4250262	3.46	0.03	4250257
70	5,5	3	5	60	79.12	0.97	4806567	45.71	0.62	4806556	3.53	0.08	4806557
90	4,4	3	5	60	132.82	0.59	4909700	69.65	1.16	4909695	5.04	0.08	4909212

Tabla 5.4: Imágenes de resolución 256x256 píxeles

radio	Dp	E	W	nmáx	MATLAB			C			OpenCL		
					media	std	métrica	media	std	métrica	media	std	métrica
70	3,3	3	5	60	108.45	0.59	2206100	50.23	0.63	2206145	3.60	0.05	2199500
70	4,4	2	5	60	61.58	0.47	2318400	33.42	0.37	2318390	2.47	0.07	2318389
70	4,4	3	3	60	91.14	0.51	2400400	51.11	0.72	2400390	3.52	0.02	2400389
70	4,4	3	5	30	46.50	0.22	3060400	25.97	0.43	3060417	2.20	0.03	3060419
70	4,4	3	5	60	93.13	0.75	2402900	51.74	0.51	2402867	3.64	0.24	2402866
70	4,4	3	5	80	120.86	0.65	2192500	72.66	3.50	2192514	4.42	0.04	2192515
70	4,4	3	7	60	91.20	0.52	2420300	51.20	0.70	2420311	3.56	0.06	2420311
70	5,5	3	5	60	79.09	0.47	2561800	51.80	0.73	2561792	3.62	0.03	2561790
90	4,4	3	5	60	139.95	0.72	2838100	76.49	0.64	2838120	5.08	0.04	2838100

En cuanto a los resultados, se observa que el mayor aumento de la carga computacional se produce al aumentar el radio, por lo que lo mejor será escoger una ROI que cubra bien la estructura de interés y un pequeño margen.

El aumento del número de iteraciones implica también un aumento de la carga computacional importante. Lo ideal será también escoger un número de iteraciones correcto, es decir, un número a partir del cual las deformaciones relativas fuesen despreciables. La elección del número de iteraciones ya fue mencionada en la sección 2.4 del documento.

En cuanto al valor de los pesos, apenas afecta al tiempo de ejecución, por lo que ese parámetro no debería suponer ningún problema si se desea mejorarlo.

El espaciado entre los puntos de control tampoco afecta demasiado al tiempo de ejecución (aunque sí notablemente en MATLAB), por lo que sí se podría aumentar la resolución disminuyendo el espaciado sin que suponga un tiempo apreciable.

El cambio más apreciable en la disminución de la carga computacional, y con ello el tiempo de ejecución, es la disminución del orden de las B-splines, puesto que, como ya se ha mencionado en la sección 2.5, el cuello de botella es la computación de la B-spline cúbica.

Por lo que se aprecia en las tablas comparativas, los cambios de los parámetros en OpenCL no provocan grandes aumentos en el tiempo de ejecución, por lo que se pueden adaptar perfectamente a un rango amplio de MRI sin necesidad de preocuparse demasiado por el aumento de la carga computacional.

Capítulo 6

Conclusiones y líneas futuras

En el presente proyecto se ha llevado a cabo la optimización de un esquema de compensación del movimiento del corazón humano a partir de una secuencia de resonancia magnética cardiaca. El método de registrado ha sido no rígido empleando un modelo FFD basado en B-splines, utilizando un registrado grupal con una métrica de similitud que minimiza las variaciones en intensidad a lo largo del tiempo y una optimización empleando un método de gradiente descendente con una estimación adaptativa del valor de los pesos. Los resultados parten de MRI de pacientes en apnea, por lo que en este proyecto los desalineamientos causados por la respiración de los pacientes no han sido caso de estudio.

Se ha desarrollado un algoritmo eficiente basado en la manipulación de datos en paralelo empleando la GPU. En el estudio nos hemos dado cuenta de que la utilización de la GPU nos ofrece la posibilidad de variar algunos parámetros importantes para la realización del registrado sin que los tiempos de ejecución aumenten de forma apreciable. Esto nos permitiría aumentar la ROI para asegurarnos de hacer el registrado en la zona de interés o aumentar la resolución disminuyendo el espaciado de los puntos de control entre los más importantes, así como disminuir el valor de los pesos y aumentar el número de iteraciones para hacerlo más preciso.

Para el desarrollo futuro, puede haber algunas implementaciones a desarrollar y mejorar. Lo primero sería la realización del suavizado, con lo que se arreglarían los pequeños desalineamientos en los bordes de la estructura. Se podrían reducir aún más los tiempos de ejecución mediante la implementación en OpenCL de todo el código, principalmente si empleamos la GPU para el cálculo de los parámetros necesarios para la estructura B-spline. Por último, se podrían corregir esos pequeños errores prácticamente despreciables, que pueden ser por la utilización de datos tipo *float*, lo que podría ir acumulando un pequeño error que se produce frente a datos tipo *double*.

Apéndice A

Código para la realización del registrado

- **Función *main*:**

```
#include<iostream>
#include<cstdlib>
#include<cstdio>
#include<string>
#include<cmath>
#ifdef __APPLE__
#include<OpenCL/opencl.h>
#else
#include<CL/cl.h>
#endif


#include "mascara.h"
//#include "baseDatos.h"
#include "createbspline.h"
#include "optimizador.h"
#include <string.h>


#include <time.h>


#define IMG_SIZE 100
#define MAX_SRC_SIZE (0x100000)



using namespace std;


/*void err_check( int err, string err_code ) {
if ( err != CL_SUCCESS ) {
cout << "Error: " << err_code << "(" << err << ")" << endl;
exit(-1);
}
}*/

double standard_deviation(double data[], int n, double mean)
{
double sum_deviation=0.0;
int i=0;
for(i=0; i<n;++i)
sum_deviation+=(data[i]-mean)*(data[i]-mean);
return sqrt(sum_deviation/(n-1));
}
```

```
int main()
{
clock_t start, end;
//double cpu_time_used[10];
double cpu_time_used;
//clock_t t1=clock();
//printf("Dummy Statement\n");



int *tam=(int *)malloc(3*sizeof(int ));
//for(veces=0; veces<10;veces++)
//{
start = clock();



//Obtengo el tama o de los datos
FILE* f = fopen("/nfs/export/pfc/rpalsan/Santi_FFD/size.bin", "rb");
size_t f_bytes = 0;

// Compruebo si se ha abierto bien el archivo
if (f == NULL) {
cout << "Error al abrir el archivo" << endl;
return(-1);
} else {
fseek(f, 0, SEEK_END );
f_bytes = ftell(f);
fseek(f, 0, SEEK_SET);

cout << "Cargado archivo de" << f_bytes << " bytes." << endl;
}


fread(tam, sizeof(int)*3, 1, f);

fclose(f);

int N=tam[2];

float *IC=(float *)malloc(tam[0]*tam[1]*N*sizeof(float ));


//Imagen transformada
float *IT=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float ));

//Obtengo la imagen original
FILE* fp = fopen("/nfs/export/pfc/rpalsan/Santi_FFD/datos.bin", "rb");
size_t fp_bytes = 0;

// Compruebo si se ha abierto bien el archivo
if (fp == NULL) {
cout << "Error al abrir el archivo" << endl;
return(-1);
} else {
fseek(fp, 0, SEEK_END );
fp_bytes = ftell(fp);
fseek(fp, 0, SEEK_SET);

cout << "Cargado archivo de" << fp_bytes << " bytes." << endl;
}
```

```
fread(IC, sizeof(float)*tam[0]*tam[1]*tam[2], 1, fp);

fclose(fp);



printf("lll -> %d %d %d", tam[0], tam[1],tam[2]);


//Inicializo la imagen transformada a la imagen original
for(int i=0; i<tam[0]*tam[1]*tam[2]; i++)
{
IT[i] = (float)IC[i];
}




//indica el radio del circulo de unos
int radio=70;

//Area del circulo correspondiente a la suma de todos los elementos de la matriz
int areaX=0;

//bounding box sobre el circulo que le circunscribe
int *caja=(int *)malloc(2*2*sizeof(int ));


//X define la regi n de interes
int *X=(int *)malloc(tam[0]*tam[1]*sizeof(int ));

mascara(&X,radio, tam, &areaX, caja);




//Obtengo las variables B-spline, almacenadas en la estructura ts
int Dp[2]={4,4};//Separacion entre puntos de control
int E=2;//Orden de las bsplines


//struct Bspline *ts={0};

Bspline_t* ts = creabspline(tam, caja, Dp, E, tam[3]);



//COMPARACION DE LA ESTRUCTURA TS OBTENIDA ECLIPSE CON LA ESTRUCTURA TS OBTENIDA EN
    MATLAB
/*
------ts->coef------
MATLAB:
ts.coef{1} 400x2
ts.coef{2} 400x2


ECLIPSE:
ts->coef1 400x2
ts->coef2 400x2

eclipse                matlab
```

```
ts->coef1[x][y] => ts.coef{1}(x+1,y+1)
ts->coef2[x][y] => ts.coef{2}(x+1,y+1)
*/

//printf("coef1--> %d\n", ts->coef1[183][1]); //ts.coef{1}(184,2)
//printf("coef2--> %d\n", ts->coef2[183][1]); //ts.coef{2}(184,2)


/*
------ts->coefg------
MATLAB:
ts.coefg{1} 39x2
ts.coefg{2} 39x2


ECLIPSE:
ts->coefg1 39x2
ts->coefg2 39x2

eclipse                 matlab
ts->coefg1[x][y] => ts.coefg{1}(x+1,y+1)
ts->coefg2[x][y] => ts.coefg{2}(x+1,y+1)
*/

//printf("coefg1--> %d\n", ts->coefg1[35][1]); //ts.coefg{1}(36,2)
//printf("coefg2--> %d\n", ts->coefg2[35][1]); //ts.coefg{2}(36,2)



/*
------ts->BB------
MATLAB:
ts.BB 400x400x30x2x4x4


ECLIPSE:
ts->BB 400x400x4x4

eclipse                 matlab
ts->BB[x][y][z][t] => ts.BB(y+1,x+1,1-30,1-2,z+1,t+1))

En MATLAB la 3   y 4   componente son repetidas (es un repmat), da igual el valor
    que ponga entre esos
*/

//printf("BB--> %f\n", ts->BB[223][64][3][1]); //ts.BB(65,224,17,2,4,2)


/*
------ts->BB1 y ts->BB2------
MATLAB:
ts.BB1 400x400x30x2x4x4x2


ECLIPSE:
ts->BB1 400x400x4x4
ts->BB2 400x400x4x4

eclipse                 matlab
ts->BB1[x][y][z][t] => ts.BB1(y+1,x+1,1-30,1-2,z+1,t+1,1)) -->BB1 es un 1 en la
    ultima componente
ts->BB2[x][y][z][t] => ts.BB1(y+1,x+1,1-30,1-2,z+1,t+1,2)) -->BB2 es un 2 en la
    ultima componente

En MATLAB la 3   y 4   componente son repetidas (es un repmat), da igual el valor
```

```
    que ponga entre esos
*/


//printf("BB1--> %f\n", ts->BB1[223][64][3][1]); //ts.BB1(65,224,17,2,4,2,1)
//printf("BB2--> %f\n", ts->BB2[223][64][3][1]); //ts.BB1(65,224,17,2,4,2,2)




/*
------ts->BBg------
MATLAB:
ts.BBg 17x17x39x39


ECLIPSE:
ts->BBg 17x17x39x39

eclipse                 matlab
ts->BBg[x][y][z][t] => ts.BBg(x+1,y+1,z+1,t+1))

*/


//printf("BBg--> %f\n", ts->BBg[14][2][26][31]); //ts.BBg(15,3,27,32)


/*
------ts->BB1g-2g------
MATLAB:
ts.BB1g 17x17x30x39x39x2x2


ECLIPSE:
ts->BB1g 17x17x39x39
ts->BB2g 17x17x39x39

eclipse                 matlab
ts->BB1g[x][y][z][t] => ts.BB1g(x+1,y+1,1-30,z+1,t+1,1-2,1)) -->BB1g es un 1 en la
    ultima componente
ts->BB2g[x][y][z][t] => ts.BB1g(x+1,y+1,1-30,z+1,t+1,1-2,2)) -->BB2g es un 2 en la
    ultima componente

En MATLAB la 3   y 6   componente son repetidas (es un repmat), da igual el valor
    que ponga entre esos
*/



//printf("BB1g--> %f\n", ts->BB1g[14][2][26][31]); //ts.BB1g(15,3,17,27,32,2,1)
//printf("BB2g--> %f\n", ts->BB2g[14][2][26][31]); //ts.BB1g(15,3,17,27,32,2,2)



/*
printf("coef1 =  %d\n", ts->coef1[21][1]); //400x2
printf("coef2 =  %d\n", ts->coef2[21][1]); //400x2
printf("coefg1 = %d\n", ts->coefg1[19][0]); //39x2
printf("coefg2 = %d\n", ts->coefg2[19][0]); //39x2
printf("  BB =   %f\n", ts->BB[399][258][2][1]); //400x400x4x4
printf("  BB1 =  %f\n", ts->BB1[399][258][2][1]); //400x400x4x4
printf("  BB2 =  %f\n", ts->BB2[399][258][2][1]); //400x400x4x4
printf("  BBg =  %f\n", ts->BBg[14][11][33][31]); //17x17x39x39
printf("  BB1g = %f\n", ts->BB1g[14][11][33][31]); //17x17x39x39
```

```
printf("  BB2g = %f\n", ts->BB2g[14][11][33][31]); //17x17x39x39
*/


/*En MATLAB:
*
ts.coef{1}(22,2)
ts.coef{2}(22,2)
ts.coefg{1}(20,1)
ts.coefg{2}(20,1)
ts.BB(400,259,17,2,3,2)
ts.BB1(400,259,17,2,3,2,1)
ts.BB1(400,259,17,2,3,2,2)
ts.BBg(15,12,34,32)
ts.BB1g(15,12,17,34,32,2,1)
ts.BB1g(15,12,17,34,32,2,2)

*/


//Hasta aqui las variables que tenemos en ECLIPSE respecto de MATLAB




Datos_t datos;
datos.alineamiento=0;


int W=5;//pesos para cada desplazamiento en cada punto de cada imagen

float Wn=float(W)/float(areaX); //normalizacion

//printf("%.10f\n",Wn);


//Defino los parametros requeridos para el gradiente descendente
int flagW=1;//flag de activacion de la adaptacion de W


Parametros_t parametros;

parametros.nmax=60;
parametros.et=0.01;//0.01 pixel para tener una convergencia plena
parametros.eh=0.005;//es un 0.5% del inicial







//MATLAB     ECLIPSE
//  I           IC       Imagen original (1 corte con 30 instantes temporales) 400
    x400x30
//  T           T        Matriz de transformacion
//  IT          IT       Imagenes transformadas
```

```
optimizador(tam,ts, IC,&IT, datos, &X, parametros, Wn, caja, flagW);



FILE * fid;

fid = fopen("/nfs/export/pfc/rpalsan/Santi_FFD/compara.bin","wb");
if (fid == NULL) { printf("Error al abrir archivo de volcado" ); exit(-1); }
for(int i = 0; i < tam[0]; i++)
for(int c = 0; c < tam[1]; c++)
for(int n = 0; n < tam[2]; n++)
fwrite(&(IT[i+c*tam[0]+n*tam[0]*tam[1]]), sizeof(float), 1, fid);

fclose(fid);









end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

/*FILE * fid_tiempo;


fid_tiempo = fopen("/nfs/export/pfc/rpalsan/Santi_FFD/tiempo1.bin","wb");
if (fid_tiempo == NULL) { printf("Error al abrir archivo de volcado " ); exit(-1); }
fwrite(&(cpu_time_used), sizeof(double), 1, fid_tiempo);

fclose(fid_tiempo);*/







//double suma_tiempo=0;
//double promedio=0;

//double suma_tiempo2=0;
// double std=0;

//for (int l=0; l<veces; l++)
//{
printf("\nTiempo que tarda en ejecutarse todo:   %f\n",cpu_time_used);
// suma_tiempo+=cpu_time_used[l];
//}
//promedio = suma_tiempo/(10);

//printf("\Promedio-->   %f\n", promedio);


//std=standard_deviation(cpu_time_used, 10, promedio);
```

```
//printf("\STD---->   %f\n", std);

printf("\n----------*ACABA_TODO_OPENCL*----------\n");

return(0);

}
```

- **Función *mascara*:**

```
#include <stdio.h>

#include<cmath>
#include <stdlib.h>
#include <math.h>

#include<iostream>


#include "mascara.h"

/*
MASCARA dibuja un circulo de cierto radio con unos en el centro de una matriz

Entradas:
-radio: indica el radio del circulo de unos
-tam: tama o de la matriz donde se dibuja el circulo

Salidas:
-X: matriz devuelta donde los unos indican el interior del circulo y los ceros el
    exterior
-areaX: Area del circulo correspondiente a la suma de todos los elementos de la
    matriz
-caja: bounding box sobre el circulo que le circunscribe
*/



void mascara (int **X,int radio,int tam[], int *areaX, int *caja)
{
//precomputo el radio al cuadrado
int radio2=radio*radio;
//coordenadas del centro
float centro[2]={float((tam[1])+1)/2, (float(tam[0])+1)/2};

float C[tam[0]][tam[1]];
float D[tam[0]][tam[1]];


//trazo la malla de distancias al centro para cada pixel
//[C D]=meshgrid((1-centro(1):tam(2)-centro(1)).^2,(1-centro(2):tam(1)-centro(2))
    .^2);
for(int i=0; i<tam[1]; i++)
{
for(int j=0; j<tam[0]; j++)
{
C[j][i]=(1-centro[0] + i)*(1-centro[0] + i);
}
}

for (int i=0; i<tam[0]; i++)
```

```
{
for(int j=0; j<tam[1]; j++)
{
D[i][j]=(1-centro[1] + i)*(1-centro[1] + i);
}
}




//valores interiores a la ROI -> 1, exteriores ->0

int num_1=0;

for (int i=0; i<tam[0]; i++)
{
for(int j=0; j<tam[1]; j++)
{
if((C[i][j]+D[i][j])<=radio2)
{
(*X)[i+j*tam[0]]=1;
/*
if(i==199)
printf("i--> %d     j--> %d\n",i,j);*/
num_1++;
}
}
}


//para almacenar los indices de todas las posiciones de los unos en X
int fil_1[num_1];
int col_1[num_1];
int min_fil1=tam[0];
int max_fil1=0;
int min_col1=tam[1];
int max_col1=0;


for (int i=0; i<tam[0]; i++)//recorre filas
{
for(int j=0; j<tam[1]; j++)//recorre columnas
{
if((*X)[i+j*tam[0]]==1)
{
//obtener el minimo indice de las filas
if(i<min_fil1)
min_fil1=i;
//obtener el minimo indice de las columnas
if(j<min_col1)
        min_col1=j;
//obtener el maximo indice de las filas
if(i>max_fil1)
        max_fil1=i;
//obtener el maximo indice de las columnas
if(j>max_col1)
        max_col1=j;


fil_1[*areaX]=i;
col_1[*areaX]=j;
(*areaX)++;
}
}
```

```
}


//Caja: valores limite de la ROI
caja[0+0]=min_fil1+1;
caja[0+1*2]=max_fil1+1;
caja[1+0]=min_col1+1;
caja[1+1*2]=max_col1+1;


//areaX: area de la ROI


}
```

- **Función *createbspline*:**

```
#include "createbspline.h"

//http://stackoverflow.com/questions/1824363/dynamic-allocation-deallocation-of-2d-3
    d-arrays

#include <stdio.h>

#include<cmath>
#include <stdlib.h>
#include <math.h>

#include<iostream>

#include "spline.h"


using namespace std;



/*
CREABSPLINE almacena los elementos necesarios para la transformacion bspline, tanto
    las matrices de productos B-spline (hasta la derivada
primera) como los coeficientes asociados, asi como la ubicacion de los puntos de
    control como su densidad, el orden del bspline, de manera que
faciliten un posterior procesado

Entradas
-tam: tama o de la matriz inicial
-caja: determina la zona donde se operara posteriormente
-Dp: Separacion entre puntos de control
-E: orden del bspline
-N: numero de imagenes

Salidas
-ts: estructura donde se almacenan las variables relacionadas con el bspline
*/



//struct Bspline creabspline(int tam[],int caja[2][2],int Dp[],int E,int N)
Bspline_t* creabspline(int tam[],int *caja,int Dp[],int E,int N)
{


//primero reservo memoria para la estructura, y luego para los miembros que lo
    necesiten
```

```
//struct Bspline *ts = malloc(sizeof(*ts)); // reservamos para la estructura;

//struct Bspline *ts;

//Bspline_t* ts = malloc(sizeof(Bspline_t));

Bspline_t* ts = (Bspline_t*)malloc(sizeof(Bspline_t));

//ts = (struct Bspline *)malloc(sizeof(struct Bspline));
//Definicion de los parametros requeridos (coord. xy)
int tamxy[4]={tam[0], tam[1], tam[0], tam[1]};

int L=2;//numero de dimensiones de la imagen
//printf("%d     ",tamxy[0]);
ts->tipo='b';
ts->nt=2;
ts->Dp[0]=Dp[0];
ts->Dp[1]=Dp[1];
ts->E=E;
ts->c[0]=(1+float(tamxy[0]))/2;
ts->c[1]=(1+float(tamxy[1]))/2;
ts->rp[0]=(ts->E+1)*ts->Dp[0]/2;//radio de influencia
ts->rp[1]=(ts->E+1)*ts->Dp[1]/2;//radio de influencia



//ubicacion en la malla de cada punto de control
for(int i=0; i<L; i++)//dimensiones de la imagen
{
for(int j=0; j<2; j++)//min-max
{
//  floor((abs(ts.c(l)- caja(l,b))   +ts.rp(l))/ts.Dp(l));
ts->C[i][j]=floor((abs(ts->c[i]-(caja[i+j*2]+1))+ts->rp[i])/ts->Dp[i]);

}

}



//RESERVA DE MEMORIA DINMICA PARA ts.pu1 y ts.pu2

int xpu1 =ts->C[0][0]+ts->C[0][1]+1, ypu1 = ts->C[0][0]+ts->C[0][1]+1;
ts->pu1=(float *)malloc(xpu1*ypu1*sizeof(float ));


int xpu2 = ts->C[1][0]+ts->C[1][1]+1, ypu2 = ts->C[1][0]+ts->C[1][1]+1;
ts->pu2=(float *)malloc(xpu2*ypu2*sizeof(float ));



//ordenacion de cada punto de control y ubicacion de cada punto
for(int i=0; i<(ts->C[0][0]+ts->C[0][1]+1); i++)
{
for(int j=0; j<(ts->C[0][0]+ts->C[0][1]+1); j++)
{
//ts->u1[j][i]=-ts->C[0][0]+i;
ts->pu1[j+i*xpu1]=ts->c[0]+ts->Dp[0]*(-ts->C[0][0]+i);
}
}

for(int i=0; i<(ts->C[1][0]+ts->C[1][1]+1); i++)
{
for(int j=0; j<(ts->C[1][0]+ts->C[1][1]+1); j++)
{
```

```
//ts.u1[j][i]=-ts.C[0][0]+i;
ts->pu2[i+j*xpu2]=ts->c[1]+ts->Dp[1]*(-ts->C[1][0]+i);

}

}









int xcoef1 =tam[0], ycoef1 = ts->nt;
ts->coef1=(int *)malloc(xcoef1*ycoef1*sizeof(int ));


int xcoef2 =tam[1], ycoef2 = ts->nt;
ts->coef2=(int *)malloc(xcoef2*ycoef2*sizeof(int ));



//coeficientes y matriz de productos para la transformacion
if (ts->E %2 != 0)//orden 2
{
for (int j=0; j<2; j++)
{
for (int i=0; i<tam[0]; i++)
{
ts->coef1[i+j*xcoef1]=floor((i+1-ts->c[0])/ts->Dp[0])+((ts->E+pow(-1,j+1))/2)*pow
    (-1,j+1);

}
}
for (int j=0; j<2; j++)
{
for (int i=0; i<tam[1]; i++)
{
ts->coef2[i+j*xcoef2]=floor((i+1-ts->c[1])/ts->Dp[1])+((ts->E+pow(-1,j+1))/2)*pow
    (-1,j+1);

}
}
}


else if (ts->E %2 == 0)//orden 1 y 3
{
for (int j=0; j<2; j++)
{
for (int i=0; i<tam[0]; i++)
{
ts->coef1[i+j*xcoef1]=round((i+1-ts->c[0])/ts->Dp[0])+(ts->E/2)*pow(-1,j+1);

}
}
for (int j=0; j<2; j++)
{
for (int i=0; i<tam[1]; i++)
{
ts->coef2[i+j*xcoef2]=round((i+1-ts->c[1])/ts->Dp[1])+(ts->E/2)*pow(-1,j+1);;

}
}
}
```

```
//inicializacion de matriz auxiliar para el calculo (2->derivada)

int zaux1 = ts->nt, xaux1 = tamxy[0], yaux1 = ts->E+1;
float *BBAux1=(float *)malloc(xaux1*yaux1*zaux1*sizeof(float ));

int zaux2 = 2, xaux2 = tamxy[1], yaux2 = ts->E+1;
float *BBAux2=(float *)malloc(xaux2*yaux2*zaux2*sizeof(float ));



int xbis1[tamxy[0]];
int xbis2[tamxy[1]];

//BBAux1[0][0][0]=0;
//printf("%f     ",BBAux1[0][0][0]);




for(int i=0; i<tamxy[0]; i++)
{
xbis1[i]=i+1;



for(int j=ts->coef1[i+0]; j<=ts->coef1[i+1*xcoef1]; j++)
{
BBAux1[i+(j-ts->coef1[i+0])*xaux1+0]= bsplineN(((xbis1[i]-ts->c[0])/ts->Dp[0])-j,ts
    ->E);


//printf("%f ",BBAux1[i][j-ts.coef1[i][0]][0]);



BBAux1[i+(j-ts->coef1[i+0])*xaux1+1*xaux1*yaux1]=b1splineN(((xbis1[i]-ts->c[0])/ts->
    Dp[0])-j,ts->E);
//printf("%f     ",BBAux1[i][j-ts.coef1[i][0]][1]);
}
//printf("\n");

}

for(int i=0; i<tamxy[1]; i++)
{
xbis2[i]=i+1;



for(int j=ts->coef2[i+0]; j<=ts->coef2[i+1*xcoef2]; j++)
{
BBAux2[i+(j-ts->coef2[i+0])*xaux2+0]= bsplineN(((xbis2[i]-ts->c[1])/ts->Dp[1])-j,ts
    ->E);



//printf("%f     ",BBAux2[i][j-ts.coef2[i][0]][0]);
```

```
BBAux2[i+(j-ts->coef2[i+0])*xaux2+1*xaux2*yaux2]=b1splineN(((xbis2[i]-ts->c[1])/ts->
    Dp[1])-j,ts->E);
//printf("%f     ",BBAux2[i][j-ts.coef2[i][0]][1]);
}
}




//matriz de productos bspline Bprimax*By y Bprimay*Bx

//int BB2[ts.E+1][ts.E+1][tamxy[0]][tamxy[1]];
//int BB1aux[tamxy[1]][tamxy[0]][2][1][ts.E+1][ts.E+1];

//printf("%d     ",ts.E);




int t1 = ts->E+1, z1 = ts->E+1, x1 = tamxy[0], y1 = tamxy[1];
ts->BB=(float *)malloc(x1*y1*z1*t1*sizeof(float));
ts->BB1=(float *)malloc(x1*y1*z1*t1*sizeof(float));
ts->BB2=(float *)malloc(x1*y1*z1*t1*sizeof(float));



for(int m=0; m<tamxy[0]; m++)
{
//printf("%d     ",ts.E);
for(int n=0; n<tamxy[1]; n++)
{
//productos matriciales
for(int t=0; t<ts->E+1; t++)
{
for(int k=0; k<ts->E+1; k++)
{
ts->BB[m+n*x1+k*x1*y1+t*x1*y1*z1]= BBAux2[m+t*xaux2+0] * BBAux1[n+k*xaux1+0];


ts->BB1[m+n*x1+k*x1*y1+t*x1*y1*z1] = ts->Dp[0] * BBAux2[m+t*xaux2+0] * BBAux1[n+k*
    xaux1+1*xaux1*yaux1];//Tener en cuenta los efectos de borde del operador
    derivada.

ts->BB2[m+n*x1+k*x1*y1+t*x1*y1*z1] = ts->Dp[1] * BBAux2[m+t*xaux2+1*xaux2*yaux2] *
    BBAux1[n+k*xaux1+0];

}
}
}
}




int coef1_caja = ts->coef1[caja[0+0]+0];
for(int i=0; i<tamxy[0]; i++)
{
for(int j=0; j<2; j++)
```

```
{
ts->coef1[i+j*xcoef1] = ts->coef1[i+j*xcoef1] +1 - coef1_caja; //coeficientes con
    offset

}

}



int coef2_caja = ts->coef2[caja[1+0]+0];
for(int i=0; i<tamxy[1]; i++)
{
for(int j=0; j<2; j++)
{
ts->coef2[i+j*xcoef2] = ts->coef2[i+j*xcoef2] +1 - coef2_caja; //coeficientes con
    offset
}
}




//       int BB1aux[tamxy[1]][tamxy[0]][2][1][ts.E+1][ts.E+1];


/*
* permute 6D
for (int i=0;i<(ts.E+1);i++)
{
for (int j=0;j<(ts.E+1);j++)
{
for (int n=0; n<1; n++)
{
for (int l=0; l<2; l++)
{
for (int k=0; k<tamxy[0]; k++)
{
for (int t=0; t<tamxy[1]; t++)
{
BB1aux[t][k][l][n][j][i]=BB1[i][j][k][t][l];

}
}
}
}
}
}*/






//------------------coeficientes y matriz de productos para el calculo del gradiente

int limxy[2]={ts->C[0][0]+ts->C[0][1]+1 , ts->C[1][0]+ts->C[1][1]+1};



int xcoefg1 =limxy[0], ycoefg1 = ts->nt;
ts->coefg1=(int *)malloc(xcoefg1*ycoefg1*sizeof(int ));


int xcoefg2 =limxy[1], ycoefg2 = ts->nt;
```

```
ts->coefg2=(int *)malloc(xcoefg2*ycoefg2*sizeof(int ));




float xbis3[limxy[0]];
float xbis4[limxy[1]];

for(int j=0; j<limxy[0]; j++)
{
xbis3[j]=ts->pu1[0+j*xpu1];
for(int i=0; i<2; i++)
{
ts->coefg1[j+i*xcoefg1]= ceil(ts->pu1[0+j*xpu1] - ceil(ts->rp[0])) + i*2*ceil(ts->rp
    [0]);
//printf("%d j--> %d i--> %d ",ts->coefg1[j][i],j,i);
//printf("%d ",ts->pu1[0][i]);

}
//printf("\n");


}


for(int j=0; j<limxy[1]; j++)
{
xbis4[j]=ts->pu2[j+0];
//printf("%d\n", xbis4[j]);
for(int i=0; i<2; i++)
{
ts->coefg2[j+i*xcoefg2]=ceil(ts->pu2[j+0] - ceil(ts->rp[1])) + i*2*ceil(ts->rp[1]);
}
}







int dim2_Aux3=2*ceil(ts->rp[0])+1;
int dim2_Aux4=2*ceil(ts->rp[1])+1;



int zaux3 = ts->nt, xaux3 = limxy[0], yaux3 = dim2_Aux3;
float *BBAux3=(float *)malloc(xaux3*yaux3*zaux3*sizeof(float ));


int zaux4 = ts->nt, xaux4 = limxy[1], yaux4 = dim2_Aux4;
float *BBAux4=(float *)malloc(xaux4*yaux4*zaux4*sizeof(float ));



for(int i=0; i<limxy[0];i++)
{
for(int j=ts->coefg1[i+0]; j<=ts->coefg1[i+1*xcoefg1]; j++)
{
BBAux3[i+(j-ts->coefg1[i+0])*xaux3+0]= bsplineN(((j-xbis3[i])/ts->Dp[0]),ts->E);



//printf("%f    ",BBAux3[i][j-ts.coefg1[i][0]][0]);
```

```
//printf("%f\n",xbis3[i]);

//printf("%d\n",ts.coefg1[i][0]);

BBAux3[i+(j-ts->coefg1[i+0])*xaux3+1*xaux3*yaux3]= b1splineN(((j-xbis3[i])/ts->Dp
    [0]),ts->E);
//printf("%f    ",BBAux3[i][j-ts.coefg1[i][0]][1]);
}
//printf("\n");

}




for(int i=0; i<limxy[1];i++)
{
for(int j=ts->coefg2[i+0]; j<=ts->coefg2[i+1*xcoefg2]; j++)
{
BBAux4[i+(j-ts->coefg2[i+0])*xaux4+0]= bsplineN(((j-xbis4[i])/ts->Dp[1]),ts->E);



//printf("%f    ",BBAux4[i][j-ts.coefg1[i][0]][0]);
//printf("(i=%d,j=%d)-->%f    ",i,j-ts.coefg2[i][0],BBAux4[i][j-ts.coefg1[i][0]][0]);
//printf("%f\n",xbis3[i]);

//printf("%d\n",ts.coefg1[i][0]);

BBAux4[i+(j-ts->coefg2[i+0])*xaux4+1*xaux4*yaux4]= b1splineN(((j-xbis4[i])/ts->Dp
    [1]),ts->E);
//printf("%f    ",BBAux4[i][j-ts.coefg1[i][0]][1]);
}
//printf("\n");

}



//  Array 4 Dimensions
int x2 = 2*ceil(ts->rp[0])+1, y2 = 2*ceil(ts->rp[1])+1, z2 = limxy[0], t2 = limxy
    [1];
ts->BBg=(float *)malloc(x2*y2*z2*t2*sizeof(float));
ts->BB1g=(float *)malloc(x2*y2*z2*t2*sizeof(float));
ts->BB2g=(float *)malloc(x2*y2*z2*t2*sizeof(float));




for(int m=0; m<(2*ceil(ts->rp[0])+1); m++)
{
//printf("%d    ",ts.E);
for(int n=0; n<(2*ceil(ts->rp[1])+1); n++)
{
//productos matriciales
for(int t=0; t<limxy[1]; t++)
{
for(int k=0; k<limxy[0]; k++)
{
ts->BBg[n+m*x2+k*x2*y2+t*x2*y2*z2] = BBAux4[t+m*xaux4+0] * BBAux3[k+n*xaux3+0];



ts->BB2g[n+m*x2+k*x2*y2+t*x2*y2*z2] = 2*ts->Dp[0] * BBAux4[t+m*xaux4+0] * BBAux3[k+n
```

```
    *xaux3+1*xaux3*yaux3];
ts->BB1g[n+m*x2+k*x2*y2+t*x2*y2*z2] = 2*ts->Dp[0] * BBAux4[t+m*xaux4+1*xaux4*yaux4]
    * BBAux3[k+n*xaux3+0];

}
}
}
}









for(int j=0; j<limxy[0]; j++)
{
for(int i=0; i<2; i++)
if(ts->coefg1[j+i*xcoefg1]<1)
ts->coefg1[j+i*xcoefg1]=1;

else if(ts->coefg1[j+i*xcoefg1]>tamxy[0])
ts->coefg1[j+i*xcoefg1]=tamxy[0];




}


for(int j=0; j<limxy[1]; j++)
{
for(int i=0; i<2; i++)
{
if(ts->coefg2[j+i*xcoefg2]<1)
ts->coefg2[j+i*xcoefg2]=1;

else if(ts->coefg2[j+i*xcoefg2]>tamxy[1])
ts->coefg2[j+i*xcoefg2]=tamxy[1];
}

}


return ts;




}
```

- **Función *spline*:**

```
//aqu  voy a crear todas las funciones relacionadas con las B-Splines



#include "spline.h"
#include<cmath>
#include <stdlib.h>
#include <stdio.h>

/*funcion y = bspline1( x )
BSPLINE1 funcion B-spline de orden 1
```

```
Entradas
x: valores de entrada a modificar

Salidas
y: valores bspline correspondientes
*/

float bspline1(float x)
{
float y;

//precomputacion del valor absoluto
//float val=abs(x);

//rango de valores no nulos del bspline
if (x < 0 && x >= -1)
y = x + 1;

else if (x < 1 && x >= 0)
y = 1 - x;

else
y=0;



return y;
}



/*funcion y = bspline2( x )
BSPLINE2 funcion B-spline de orden 2

Entradas
x: valores de entrada a modificar

Salidas
y: valores bspline correspondientes
*/

float bspline2(float x)
{
float y;

//precomputacion del valor absoluto
//float val=abs(x);


//rango de valores no nulos del bspline
if (x < -0.5 && x>= -1.5)
y = (4*x*x + 12*x + 9)/8;

else if (x < 0.5 && x >= -0.5)
y = (-4*x*x + 3)/4;

else if (x < 1.5 && x >= 0.5)
y = (4*x*x - 12*x + 9)/8;

else
y=0;


return y;
```

```
}



/*funcion y = bspline3( x )
BSPLINE3 funcion B-spline de orden 3

Entradas
x: valores de entrada a modificar

Salidas
y: valores bspline correspondientes
*/

float bspline3(float x)
{
float y;

//precomputacion del valor absoluto
//float val=fabs(x);

//rango de valores no nulos del bspline
if (x < -1 && x >= -2)
y = (x*x*x + 6*x*x + 12*x + 8)/6;

else if (x <0 && x >= -1)
y = (-3*x*x*x - 6*x*x + 4)/6;

else if (x < 1 && x >= 0)
y = (3*x*x*x - 6*x*x + 4)/6;

else if (x < 2 && x >= 1)
y=(-x*x*x + 6*x*x - 12*x + 8)/6;

else
y=0;


return y;
}

/*function y = bspline0(x)
BSPLINE0 funcion B-spline de orden 0

Entradas
x: valores de entrada a modificar

Salidas
y: valores bspline correspondientes
*/

float bspline0(float x)
{
float y;
//precomputacion del valor absoluto
//float val=abs(x);


//rango de valores no nulos del bspline
if (x < 0.5 && x >= -0.5)
y=1;

else
y=0;
```

```
return y;
}


/*----------VAMOS CON LAS DERIVADAS DE LAS B-SPLINES----------*/

/*funcion y = bspline0(x)
B1SPLINEN funciOn derivada primera del B-spline de orden n

Entradas
x: valores de entrada a modificar
n: orden del bspline

Salidas
y: valores bspline correspondientes
*/

float b1spline1(float x)
{
float y;

//precomputacion del valor absoluto
//float val=abs(x);


//rango de valores no nulos del bspline
if (x < 0 && x >= -1)
y=1;

else if (x < 1 && x >= 0)
y=-1;

else
y=0;


return y;
}



/*
funcion y = bspline2( x )
BSPLINE2 funcion B-spline de orden 2

Entradas
x: valores de entrada a modificar

Salidas
y: valores bspline correspondientes
*/

float b1spline2(float x)
{
float y;

//precomputacion del valor absoluto
//float val=abs(x);


//rango de valores no nulos del bspline
if (x < -0.5 && x >= -1.5)
y = (2*x + 3)/2;
```

```
else if (x < 0.5 && x >= -0.5)
y = -2*x;

else if (x < 1.5 && x >= 0.5)
y = (42*x - 3)/2;

else
y = 0;


return y;
}



/*funcion y = b1spline3( x )
B1SPLINE3 funcion primera derivada B-spline de orden 3

Entradas
x: valores de entrada a modificar

Salidas
y: valores bspline correspondientes

*/

float b1spline3(float x)
{
float y;

//precomputacion del valor absoluto
//float val=fabs(x);

//rango de valores no nulos del bspline
if (x < -1 && x >= -2)
y = (x*x + 4*x + 4)/2;


else if (x < 0 && x >= -1)
y = (-3*x*x - 4*x)/2;

else if (x < 1 && x >= 0)
y = (3*x*x - 4*x)/2;

else if (x < 2 && x >= 1)
y = (-x*x + 4*x - 4)/2;

else
y=0;



return y;
}

/*
funcion y=bsplineN(x,n)
BSPLINEN funcion B-spline de orden n

Entradas
x: valores de entrada a modificar
n: orden del bspline

Salidas
y: valores bspline correspondientes
```

```
*/

float bsplineN(float x,int n)
{
float y;
//seleccion del orden por parametro
if (n==1)
y=bspline1(x);
if (n==2)
y=bspline2(x);
if (n==3)
y=bspline3(x);

return y;
}


/*
funcion y=b1splineN(x,n)
B1SPLINEN funcion derivada primera del B-spline de orden n

Entradas
x: valores de entrada a modificar
n: orden del bspline

Salidas
y: valores bspline correspondientes
*/


float b1splineN(float x,int n)
{
float y;
//seleccion del orden por parametro
if (n==1)
y=b1spline1(x);
if (n==2)
y=b1spline2(x);
if (n==3)
y=b1spline3(x);

return y;
}
```

- **Función *optimizador* en C:**

```
#include<iostream>
#include<cstdlib>
#include<cstdio>
#include<string>
#include<cmath>
#ifdef __APPLE__
#include<OpenCL/opencl.h>
#else
#include<CL/cl.h>
#endif


#include "mascara.h"
//#include "baseDatos.h"
#include "createbspline.h"
#include "optimizador.h"
#include <string.h>
```

```
#define IMG_SIZE 100
#define MAX_SRC_SIZE (0x100000)

#define MAX_SOURCE_SIZE (0x100000)




using namespace std;


void err_check( int err, string err_code ) {
if ( err != CL_SUCCESS ) {
cout << "Error: " << err_code << "(" << err << ")" << endl;
exit(-1);
}
}
// <20160404-Javier> Te pego aqu  esta macro  til  para comprobar errores en
    llamadas de OpenCL
#define CL_CHECK(_expr)                                                         \
do {                                                                              \
if ((cl_int)_expr == CL_SUCCESS)                                                \
break;                                                                        \
fprintf(stderr, "OpenCL Error: '%s' returned %d!\n", #_expr, (int)_expr);     \
abort();                                                                      \
} while (0)


// OpenCL kernel. Each work item takes care of one element of c
/*
const char *kernelSource =                                          "\n" \
"#pragma OPENCL EXTENSION cl_khr_fp64 : enable                       \n" \
"__kernel void vecAdd(  __global double *a,                          \n" \
"                       __global double *b,                          \n" \
"                       __global double *c,                          \n" \
"                       const unsigned int n)                       \n" \
"{                                                                  \n" \
"    //Get our global thread ID                                     \n" \
"    int id = get_global_id(0);                                     \n" \
"                                                                   \n" \
"    //Make sure we do not go out of bounds                         \n" \
"    if (id < n)                                                    \n" \
"        c[id] = a[id] + b[id];                                     \n" \
"}                                                                  \n" \
                                    "\n" ;
*/

/*
const char *kernelSource =                                          "\n" \
"#pragma OPENCL EXTENSION cl_khr_fp64 : enable                       \n" \
"__kernel void inicVect(  __global float *a,                         \n" \
"                         const unsigned int tam)                   \n" \
"{                                                                  \n" \
"    //Get our global thread ID                                     \n" \
"    int id = get_global_id(0);                                     \n" \
"                                                                   \n" \
"    //Make sure we do not go out of bounds                         \n" \
"    if (id < tam)
```

```
                                                \n" \
"            a[id] = 0;                                                  \n" \
"}                                                                       \n" \
"}                                                                       \n" \
                                    "\n" ;
*/


//OPTIMIZADOR se encarga del bucle principal del algoritmo de registrado

//Entradas
//Defino los parametros requeridos para el gradiente; struct %parametros
//-nmax=50;
//-et=0.01;%0.01 pixel para tener una convergencia plena.
//-eh=0.005;%es un 0.5% del inicial

//estructura de datos para optimizar; struct datos
//interpolacion='linear';
//metric='varianza';

//estructura de la transformacion; struct ts

//x malla original de meshgrid original
//T matriz de transformacion
//I imagenes originales
//X  y caja es la mascara que define la ROI
//ts parametros de transformacion (guarda la matriz bspline)
//term terminos de suavidad; pesos y derivadas temporales y espaciales
//Wn matriz de paso
//flagW flag que activa la adaptacion de Wn

//Salidas
//-H vector que guarda las metricas de cada iteracion
//-tiempo vector de tiempo total al finalizar cada iteracion
//-T matriz de transformacion final
//-IT imagenes transformadas
//-xn transformacion de la malla del meshgrid

/*
x = x1 y x2    (x(:,:,1)=x1      x(:,:,2)=x2)
ts=ts
term=term
T= optim->T que no se lo paso y lo creo dentro
I=IC
datos=datos
X=X
parametros=parametros
Wn=Wn
caja=caja
flagW=flagW
*/

//[ ~, ~, T, IT, xn] = optimizador( x,ts,term,T,I,datos,X,parametros,Wn,caja,flagW )
    ;

void optimizador(int tam[], Bspline_t* ts,
float *IC, float **IT,
Datos_t datos, int **X, Parametros_t parametros,
float Wn, int *caja, int flagW)
{
```

```
//-------------------------------------PARA LA TRANSFORMACION-------------------------------------

int   N = tam[2];
float *xn1=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float ));
float *xn2=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float ));
float *xn1Aux=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float ));
float *xn2Aux=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float ));
float *dx1=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float ));
float *dx2=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float ));
float *dy=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float ));


float *x1=(float *)malloc(tam[0]*tam[1]*sizeof(float ));
float *x2=(float *)malloc(tam[0]*tam[1]*sizeof(float ));

float *media=(float *)malloc(tam[0]*tam[1]*sizeof(float ));

float *V=(float *)malloc(tam[0]*tam[1]*sizeof(float ));



//<20160404-Javier> Inicializa x1 y x2
for (int i=0; i<tam[0]; i++) {
for(int j=0; j<tam[1]; j++) {
x1[j+i*tam[0]] = i+1;
x2[j+i*tam[0]] = j+1;
}
}

for(int n=0;n<tam[2];n++)
{
for(int i=0;i<tam[0];i++)
{
for(int j=0;j<tam[1];j++)
{
// x1[j+i*tam[0]]=i+1; //<20160404-Javier> Esto lo estas haciendo mas veces de las
    necesarias (para cada n). Por eso lo saco fuera.
xn1[j+i*tam[0]+n*tam[0]*tam[1]] = x1[j+i*tam[0]];
xn2[j+i*tam[0]+n*tam[0]*tam[1]] = x2[j+i*tam[0]];  //<20160404-Javier> Tambin
    puedes inicializar xn2
}
//printf("\n");
}
}

//              for(int n=0;n<tam[2];n++)
//              {
//                      for(int i=0;i<tam[0];i++)
//                      {
//                              for(int j=0;j<tam[1];j++)
//                              {
//                                      x2[j+i*tam[0]]=j+1;
//                                      xn2[j+i*tam[0]+n*tam[0]*tam[1]] = x2[j+i*tam
    [0]];
//                              }
//                      }
//              }


int xT = ts->C[1][0] + ts->C[1][1] + 1, yT = ts->C[0][0] + ts->C[0][1] + 1, zT = N,
    tT = ts->nt;
float *T=(float *)malloc(xT*yT*zT*tT*sizeof(float ));
```

```
int long_r1 = caja[0+1*2] - caja[0+0] + 1;

//int *r1 = (int *)malloc(sizeof(int)*long_r1);


int long_r2 = caja[1+1*2] - caja[1+0] + 1;


int xTAux = long_r1, yTAux = long_r2, zTAux = N, tTAux = ts->nt, pTAux = (ts->E + 1)
    , qTAux = (ts->E + 1);
float *TAux=(float *)malloc(xTAux*yTAux*zTAux*tTAux*pTAux*qTAux*sizeof(float ));




int xlim=2*ceil(ts->rp[0])+1, ylim=2*ceil(ts->rp[1])+1, zlim=ts->C[0][0]+ts->C
    [0][1]+1, tlim=ts->C[1][0]+ts->C[1][1]+1;

int lim[4]={xlim, ylim, zlim, tlim};//17 17 39 39

int xdV = lim[0], ydV = lim[1], zdV = tam[2], tdV = lim[2], pdV = lim[3], qdV = ts
    ->nt;
float *dV=(float *)malloc(xdV*ydV*zdV*tdV*pdV*qdV*sizeof(float ));







int r1[long_r1];
int r2[long_r2];

int k1=0;

for (int i=caja[0+0]; i<=caja[0+1*2];i++)
{
r1[k1]=i;
//printf("%d  ",r1[k1]);
k1++;

}

int k2=0;

for (int i=caja[1+0]; i<=caja[1+1*2];i++)
{
r2[k2]=i;
//printf("%d  ",r2[k2]);
k2++;
}

//calculo de un 2.5% de la imagen
int margen1 = ceil(float(tam[0])/40);
int margen2 = ceil(float(tam[1])/40);
int margen3 = ceil(float(tam[2])/40);

//int tdH = ts->nt, zdH = tam[2], xdH = lim[3], ydH = lim[2];
//float *dH=(float *)malloc(xdH*ydH*zdH*tdH*sizeof(float ));
```

```
int max=parametros.nmax;

float *H1 = (float *)malloc((max+1) * sizeof(float ));
float *TDif = (float *)malloc((max+1) * sizeof(float ));
float *HDif = (float *)malloc((max+1) * sizeof(float ));

//float  H1[max] = {0};
//float  TDif[60] = {0};
//float  HDif[60] = {0};


int xH = lim[3], yH = lim[2], zH = N, tH = ts->nt;
float *H2=(float *)malloc(xH*yH*zH*tH*sizeof(float ));
float *proyH2=(float *)malloc(xH*yH*zH*tH*sizeof(float ));
float *Dif=(float *)malloc(xH*yH*zH*tH*sizeof(float ));



int zmedia2 = ts->nt, xmedia2 = lim[3], ymedia2 = lim[2];
float *media2=(float *)malloc(xmedia2*ymedia2*zmedia2*sizeof(float ));



//float metrica (float *V, float* IT, Datos_t datos, Term_t* term, int caja[2][2],
    int tam[], float*media, float* dy)
coste(&H1[0],&H2,metrica (V, IC,datos, caja, tam, media, &dy,ts),(*X),0,tam,ts,lim,
    dV);



//printf("H1--> %f\n",H1[0]);

//tengo del main que IT=IC


//parametros de condicion de parada:
float p2 = parametros.et*tam[2]*ts->nt;
float p4 = H1[0]*parametros.eh;

int iter=1;


int condicion_parada=0;




while(condicion_parada==0)
{



for(int i=0; i<lim[0]; i++)
{
for(int j=0; j<lim[1]; j++)
{
for(int k=0; k<tam[2]; k++)
{
```

```
for(int l=0; l<lim[2]; l++)
{
for(int n=0; n<lim[3]; n++)
{
        dV[i+j*lim[0]+k*lim[0]*lim[1]+l*lim[0]*lim[1]*tam[2]+n*lim[0]*lim[1]*tam[2]*
            lim[2]+0]=0;
        dV[i+j*lim[0]+k*lim[0]*lim[1]+l*lim[0]*lim[1]*tam[2]+n*lim[0]*lim[1]*tam[2]*
            lim[2]+1*lim[0]*lim[1]*tam[2]*lim[2]*lim[3]]=0;

        H2[n+l*lim[3]+k*lim[3]*lim[2]+0]=0;
        H2[n+l*lim[3]+k*lim[3]*lim[2]+1*lim[3]*lim[2]*tam[2]]=0;

        proyH2[n+l*lim[3]+k*lim[3]*lim[2]+0]=0;
        proyH2[n+l*lim[3]+k*lim[3]*lim[2]+1*lim[3]*lim[2]*tam[2]]=0;

        Dif[n+l*lim[3]+k*lim[3]*lim[2]+0]=0;
        Dif[n+l*lim[3]+k*lim[3]*lim[2]+1*lim[3]*lim[2]*tam[2]]=0;
}
}

}

}
}


gradiente(&dV, dx1, dx2, dy, (*IT), ts, datos, x1, x2, T, tam, lim);




printf("H1--> %f",H1[0]);




for(int i=0; i<tam[0]; i++)
{
for(int j=0; j<tam[1]; j++)
{
for(int k=0; k<tam[2]; k++)
{

xn1[j+i*tam[0]+k*tam[0]*tam[1]] = x1[j+i*tam[0]];
xn2[j+i*tam[0]+k*tam[0]*tam[1]] = x2[j+i*tam[0]];
dx1[j+i*tam[0]+k*tam[0]*tam[1]]=0;
dx2[j+i*tam[0]+k*tam[0]*tam[1]]=0;


(*IT)[i+j*tam[0]+k*tam[0]*tam[1]] = IC[i+j*tam[0]+k*tam[0]*tam[1]];

}

}

}


coste(&H1[iter],&H2,V,(*X),1,tam,ts,lim,dV);
```

```
proyeccion(H2, &proyH2,tam,lim, media2, Wn,&T,&Dif);




transformar (xn1Aux, xn2Aux,&xn1, &xn2, &TAux, ts, T, caja, 1, tam, long_r1, long_r2
    , r1, r2,lim);


interpolar (&(*IT), &xn1, &xn2, IC, caja, margen1, margen2, margen3, tam);




coste(&H1[iter],&H2,metrica (V, (*IT),datos, caja, tam, media, &dy,ts),(*X),0,tam,ts
    ,lim,dV);

//printf("\n\nH1--> %f %f %f\n",H1[0],H1[1],H1[2]);


//void evolucion (float *TDif, float HDif, int *Wn, float ****T, float H[], int
    iter, int flagW)




evolucion(&TDif[iter-1], &HDif[iter-1], &Wn, Dif, H1, iter, flagW,lim,tam);


printf("\n\nH1--> %f, TDif--> %f HDif--> %f\n",H1[iter],TDif[iter-1],HDif[iter-1]);



if(((max)==iter)||((TDif[iter-1]==p2)&&(HDif[iter-1]==p4)))
//if(iter==1||((TDif[iter-1]==p2)&&(HDif[iter-1]==p4)))
{
condicion_parada=1;
}



printf("\niter %d\n",iter);
iter=iter+1;



}//fin while
```

```
}
```

- **Función *metrica*:**

```
#include "createbspline.h"

//Optim_t* optimizador(int tam[], int ***x1,int ***x2, Bspline_t* ts, Term_t*
    term, float* IC, Datos_t datos, int ***X, Parametros_t parametros, float Wn,
     int caja[2][2], int flagW)


// METRICA calcula una metrica dada de una matriz de imagenes
//Entradas
//-IT: imagenes sobre las que se calcula la metrica
//-datos: estructura que almacena el tipo de metrica usada
//-term: estructura que almacena los terminos de suavidad a tener en cuenta para
     la metrica total
//-caja: determina la zona de las imagenes donde se operara

//Salidas
//-V: metrica de las imagenes en cada punto, del tama o de las imagenes de
    entrada
//void metrica (float***V, float*** IT, Datos_t datos, Term_t* term, int caja
    [2][2], int tam[], float**media, float**** dy)
float* metrica (float *V, float* IT, Datos_t datos, int *caja, int tam[], float*
    media, float** dy, Bspline_t* ts)

{

//case 'varianza'
//varianza de las imagenes
//if(datos.alineamiento==0)


//al pasarsela a coste(), hay que inicializarla a 0 en algun momento para que no
     se acumule V, ya que va a ser un bucle




for(int i=0; i<tam[0]; i++)
{
for(int j=0; j<tam[1]; j++)
{
V[i+j*tam[0]]=0;
}
}


float suma=0;
for(int i=0; i<tam[0]; i++)
{
for(int j=0; j<tam[1]; j++)
{
suma=0;
for(int k=0; k<tam[2]; k++)
{
//suma[i][j] += IT[i][j][k];
suma+= IT[i+j*tam[0]+k*tam[0]*tam[1]];
```

```
}
media[i+j*tam[0]]=suma/tam[2];

}
}




for(int i=0; i<tam[0]; i++)
{
for(int j=0; j<tam[1]; j++)
{
for(int k=0; k<tam[2]; k++)
{
(*dy)[i+j*tam[0]+k*tam[0]*tam[1]]=(2.0/tam[2])*(IT[i+j*tam[0]+k*tam[0]*tam[1]]-
    media[i+j*tam[0]]);
V[i+j*tam[0]] = V[i+j*tam[0]] + (IT[i+j*tam[0]+k*tam[0]*tam[1]]-media[i+j*tam
    [0]])*(IT[i+j*tam[0]+k*tam[0]*tam[1]]-media[i+j*tam[0]])/tam[2];//calculo de
     la varianza

}
}
}




//Se aden los terminos de suavidad a la metrica

//V(caja(2,1):caja(2,2),caja(1,1):caja(1,2))=V(caja(2,1):caja(2,2),caja(1,1):
    caja(1,2))+sum(sum((term.landa(1)*sum((term.dtaux(caja(2,1):caja(2,2),caja
    (1,1):caja(1,2)).^2),5)+(term.dtaut(caja(2,1):caja(2,2),caja(1,1):caja(1,2))
    .^2)*term.landa(3)),4),3);
/*for(int i=caja[1][0]-1; i<caja[1][1]; i++)
{
for(int j=caja[0][0]-1; j<caja[0][1]; j++)
{
for(int n=0; n<tam[2]; n++)
{
for(int l=0; l<ts->nt; n++)
{
for(int m=0; m<ts->nt; m++)
{
V[i+j*tam[0]]+=(term->lambda[0]) * (term->dtaux[i+j*tam[0]+n*tam[0]*tam[1]+l*tam
    [0]*tam[1]*tam[2]+m*tam[0]*tam[1]*tam[2]*ts->nt]) * (term->dtaux[i+j*tam[0]+
    n*tam[0]*tam[1]+l*tam[0]*tam[1]*tam[2]+m*tam[0]*tam[1]*tam[2]*ts->nt]) + (
    term->dtaut[i+j*tam[0]+n*tam[0]*tam[1]+l*tam[0]*tam[1]*tam[2]]) * (term->
    dtaut[i+j*tam[0]+n*tam[0]*tam[1]+l*tam[0]*tam[1]*tam[2]])*(term->lambda[2]);

}
}
}

//(*V)[i][j]+=
}
}*/
return V;
}
```

- **Función *coste*:**

```
#include "createbspline.h"



void coste(float *H1, float **H2, float *V,int *X, int flag_coste,int tam[],
    Bspline_t* ts,int lim[], float *dV)
{
*H1=0;

//printf("\n%d %d\n",lim[0],lim[1]);



for(int i=0; i<tam[0]; i++)
{
for(int j=0; j<tam[1]; j++)
{
*H1=*H1 + V[i+j*tam[0]]*X[i+j*tam[0]];
}
}




for(int k=0; k<lim[3]; k++)
{
for(int l=0; l<lim[2]; l++)
{
for(int n=0; n<tam[2]; n++)
{
for(int i=ts->coefg1[l+0]; i<=ts->coefg1[l+1*lim[2]]; i++)
{
for(int j=ts->coefg2[k+0]; j<=ts->coefg2[k+1*lim[3]]; j++)
{

(*H2)[k+l*lim[3]+n*lim[3]*lim[2]+0]+= dV[j-ts->coefg2[k+0]+(i-ts->coefg1[l+0])*
    lim[0]+n*lim[0]*lim[1]+l*lim[0]*lim[1]*tam[2]+k*lim[0]*lim[1]*tam[2]*lim
    [2]+0]*X[j-1+(i-1)*tam[0]];
(*H2)[k+l*lim[3]+n*lim[3]*lim[2]+1*lim[3]*lim[2]*tam[2]]+=dV[j-ts->coefg2[k+0]+(
    i-ts->coefg1[l+0])*lim[0]+n*lim[0]*lim[1]+l*lim[0]*lim[1]*tam[2]+k*lim[0]*
    lim[1]*tam[2]*lim[2]+1*lim[0]*lim[1]*tam[2]*lim[2]*lim[3]]*X[j-1+(i-1)*tam
    [0]];

}

}
}

}
}

}
```

- **Función *gradiente*:**

```
#include "createbspline.h"

//GRADIENTE se encarga del ensamblado de cada una de las partes
//involucradas en el calculo del gradiente, tanto el gradiente de la imagen
//como el de la metrica y el de la transformacion, teniendo en cuenta los
//terminos de suavidad
```

```
//Entradas
//-x: malla de pixeles inicial
//-IT: matriz de imagenes
//-ts: estructura de almacenamiento de las variables bspline, involucradas
//-solo en la transformacion bspline
//-datos: estructura que almacena el tipo de transformacion y la metrica
    utilizada
//-T: matriz de transformacion con los desplazamientos
//-term: estructura que almacena todas las variables implicadas en el calculo de
     los terminos de suavidad espacial y temporal, solo para bsplines

//Salidas
//-dV: gradiente total de las imagenes empleado en el metodo del gradiente
    descendente

void gradiente (float**dV, float* dx1, float* dx2, float* dy,  float* IT,
    Bspline_t* ts, Datos_t datos, float *x1, float *x2, float *T,  int tam[],
    int lim[])
{


gradiente_imagen(&dx1, &dx2, IT, tam);


//dy seria el gradiente_metrica
//ts.BBg seria el gradiente_transformacion

/*
------ts->BBg------
MATLAB:
ts->BBg 17x17x39x39


ECLIPSE:
dtheta 17x17x30x39x39

eclipse                matlab
ts->BBg[x][y][z][t] => ts.BBg(x+1,y+1,1-30,z+1,t+1))

*/


//elseif ndims(T)==4 %bsplines

for(int k=0; k<lim[2]; k++)
{
for(int l=0; l<lim[3]; l++)
{
for(int i=ts->coefg1[k+0]; i<=ts->coefg1[k+1*lim[2]]; i++)
{
for(int j=ts->coefg2[l+0]; j<=ts->coefg2[l+1*lim[3]]; j++)
{

for(int n=0; n<tam[2]; n++)
{


(*dV)[j-(ts->coefg2[l+0])+(i-(ts->coefg1[k+0]))*lim[0]+n*lim[0]*lim[1]+k*lim[0]*
    lim[1]*tam[2]+l*lim[0]*lim[1]*tam[2]*lim[2]+0]= dy[j-1+(i-1)*tam[0]+n*tam
    [0]*tam[1]]*dx1[j-1+(i-1)*tam[0]+n*tam[0]*tam[1]]*ts->BBg[j-(ts->coefg2[l
    +0])+(i-(ts->coefg1[k+0]))*lim[0]+k*lim[0]*lim[1]+l*lim[0]*lim[1]*lim[2]];
(*dV)[j-(ts->coefg2[l+0])+(i-(ts->coefg1[k+0]))*lim[0]+n*lim[0]*lim[1]+k*lim[0]*
    lim[1]*tam[2]+l*lim[0]*lim[1]*tam[2]*lim[2]+1*lim[0]*lim[1]*tam[2]*lim[2]*
    lim[3]]=dy[j-1+(i-1)*tam[0]+n*tam[0]*tam[1]]*dx2[j-1+(i-1)*tam[0]+n*tam[0]*
    tam[1]]*ts->BBg[j-(ts->coefg2[l+0])+(i-(ts->coefg1[k+0]))*lim[0]+k*lim[0]*
```

```
    lim[1]+l*lim[0]*lim[1]*lim[2]];

}
}
}

}
}

}
```

- **Función *gradiente_imagen*:**

```
#include "createbspline.h"



void gradiente_imagen (float** dx1, float** dx2, float* IT,int tam[])
{


//para los bordes
int h=1;


//bordes para las variaciones en y
for(int i=0; i<tam[1]; i++)
{

for(int k=0; k<tam[2]; k++)
{
//borde superior
(*dx2)[0+i*tam[0]+k*tam[0]*tam[1]]=IT[1+i*tam[0]+k*tam[0]*tam[1]]-IT[0+i*tam
    [0]+k*tam[0]*tam[1]];
//borde inferior
(*dx2)[tam[0]-1+i*tam[0]+k*tam[0]*tam[1]]=IT[tam[0]-1+i*tam[0]+k*tam[0]*tam
    [1]]-IT[tam[0]-2+i*tam[0]+k*tam[0]*tam[1]];

//borde izquierdo y derecho
if(i>0 && i<(tam[0]-1))
{
//borde izquierdo
(*dx2)[i+0+k*tam[0]*tam[1]]=(IT[i+1+0+k*tam[0]*tam[1]]-IT[i-1+0+k*tam[0]*tam
    [1]])/(2);
//borde derecho
(*dx2)[i+(tam[1]-1)*tam[0]+k*tam[0]*tam[1]]=(IT[i+1+(tam[1]-1)*tam[0]+k*tam
    [0]*tam[1]]-IT[i-1+(tam[1]-1)*tam[0]+k*tam[0]*tam[1]])/(2);
}
}

}


//bordes para las variaciones en x
for(int i=0; i<tam[0]; i++)
{

for(int k=0; k<tam[2]; k++)
{
//borde izquierdo
(*dx1)[i+0+k*tam[0]*tam[1]]=(IT[i+1*tam[0]+k*tam[0]*tam[1]]-IT[i+0+k*tam[0]*
    tam[1]])/(h);
//borde derecho
```

```
(*dx1)[i+(tam[1]-1)*tam[0]+k*tam[0]*tam[1]]=(IT[i+(tam[1]-1)*tam[0]+k*tam
    [0]*tam[1]]-IT[i+(tam[1]-2)*tam[0]+k*tam[0]*tam[1]])/(h);

//borde superior e inferior
if(i>0 && i<(tam[0]-1))
{
//borde superior
(*dx1)[0+i*tam[0]+k*tam[0]*tam[1]]=(IT[0+(i+1)*tam[0]+k*tam[0]*tam[1]]-IT
    [0+(i-1)*tam[0]+k*tam[0]*tam[1]])/(2);
//borde inferior
(*dx1)[tam[1]-1+i*tam[0]+k*tam[0]*tam[1]]=(IT[tam[1]-1+(i+1)*tam[0]+k*tam
    [0]*tam[1]]-IT[tam[1]-1+(i-1)*tam[0]+k*tam[0]*tam[1]])/(2);
}
}

}




//para lo que no son los bordes
h=2;


for(int i=1; i<tam[0]-1; i++)
{
for(int j=1; j<tam[1]-1; j++)
{

for(int k=0; k<tam[2]; k++)
{

(*dx2)[i+j*tam[0]+k*tam[0]*tam[1]]=(IT[i+1+j*tam[0]+k*tam[0]*tam[1]]-IT[i-1+
    j*tam[0]+k*tam[0]*tam[1]])/(h);


(*dx1)[i+j*tam[0]+k*tam[0]*tam[1]]=(IT[i+(j+1)*tam[0]+k*tam[0]*tam[1]]-IT[i
    +(j-1)*tam[0]+k*tam[0]*tam[1]])/(h);
}
}

}



}
```

- **Función *proyeccion*:**

```
#include "createbspline.h"



void proyeccion(float *H2, float **proyH,int tam[], int lim[],float *media,
    float Wn, float **T, float **Dif)
{
float suma0=0;
float suma1=0;
for(int i=0; i<lim[3]; i++)
{
for(int j=0; j<lim[2]; j++)
{
suma0=0;
suma1=0;
```

```
for(int k=0; k<tam[2]; k++)
{
//suma[i][j] += IT[i][j][k];
suma0+= H2[i+j*lim[3]+k*lim[3]*lim[2]+0];

suma1+= H2[i+j*lim[3]+k*lim[3]*lim[2]+1*lim[3]*lim[2]*tam[2]];


media[i+j*lim[3]+0]=suma0/tam[2];
media[i+j*lim[3]+1*lim[3]*lim[2]]=suma1/tam[2];
}


}
}




for(int i=0; i<lim[3]; i++)
{
for(int j=0; j<lim[2]; j++)
{
for(int k=0; k<tam[2]; k++)
{
(*proyH)[i+j*lim[3]+k*lim[3]*lim[2]+0]=H2[i+j*lim[3]+k*lim[3]*lim[2]+0]-media[i+
    j*lim[3]+0];




(*proyH)[i+j*lim[3]+k*lim[3]*lim[2]+1*lim[3]*lim[2]*tam[2]]=H2[i+j*lim[3]+k*lim
    [3]*lim[2]+1*lim[3]*lim[2]*tam[2]]-media[i+j*lim[3]+1*lim[3]*lim[2]];

(*Dif)[i+j*lim[3]+k*lim[3]*lim[2]+0]=Wn*(*proyH)[i+j*lim[3]+k*lim[3]*lim[2]+0];
(*Dif)[i+j*lim[3]+k*lim[3]*lim[2]+1*lim[3]*lim[2]*tam[2]]=Wn*(*proyH)[i+j*lim
    [3]+k*lim[3]*lim[2]+1*lim[3]*lim[2]*tam[2]];
//actualizo T
(*T)[i+j*lim[2]+k*lim[2]*lim[3]+0]=(*T)[i+j*lim[2]+k*lim[2]*lim[3]+0]-(*Dif)[i+j
    *lim[2]+k*lim[2]*lim[2]+0];
(*T)[i+j*lim[2]+k*lim[2]*lim[3]+1*lim[2]*lim[3]*tam[2]]=(*T)[i+j*lim[2]+k*lim
    [2]*lim[3]+1*lim[2]*lim[3]*tam[2]]-(*Dif)[i+j*lim[3]+k*lim[3]*lim[2]+1*lim
    [3]*lim[2]*tam[2]];

}
}
}

}
```

- **Función *transformar*:**

```
#include "createbspline.h"

#include <stdio.h>

#include<cmath>
#include <stdlib.h>
#include <math.h>

#include<iostream>

#include "spline.h"
```

```
//void optimizador(int tam[], int ***x1,int ***x2, Bspline_t* ts, Term_t* term,
    float* IC, Datos_t datos, int ***X, Parametros_t parametros, float Wn, int
    caja[2][2], int flagW)

void transformar (float *xn1Aux, float *xn2Aux, float **xn1, float **xn2, float
    **TAux, Bspline_t* ts, float *T, int *caja, int flagopt, int tam[], int
    longr1, int longr2, int r1[], int r2[],int lim[])
{

int N=tam[2];



int laux=0;
int kaux=0;



for (int i = 0; i < longr2; i++)
{
kaux = ts->coef2[r2[i]-1+0]-1;
for (int j = 0; j < longr1; j++)
{
laux = ts->coef1[r1[j]-1+0]-1;



for (int n = 0; n < N; n++)
{
for(int l = 0; l <= ts->E; l++)


{
for(int k = 0; k <= ts->E; k++)
{


(*TAux)[i+j*longr1+n*longr1*longr2+0+l*longr1*longr2*tam[2]*ts->nt+k*longr1*
    longr2*tam[2]*ts->nt*(ts->E+1)] = T[k+kaux+(l+laux)*lim[2]+n*lim[2]*lim
    [3]+0];

(*TAux)[i+j*longr1+n*longr1*longr2+1*longr1*longr2*tam[2]+l*longr1*longr2*tam
    [2]*ts->nt+k*longr1*longr2*tam[2]*ts->nt*(ts->E+1)] = T[k+kaux+(l+laux)*lim
    [2]+n*lim[2]*lim[3]+1*lim[2]*lim[3]*tam[2]];


(*xn1)[i+r1[0]-1+(j+r2[0]-1)*tam[0]+n*tam[0]*tam[1]] +=(*TAux)[i+j*longr1+n*
    longr1*longr2+0+l*longr1*longr2*tam[2]*ts->nt+k*longr1*longr2*tam[2]*ts->nt
    *(ts->E+1)]* ts->BB[i+r1[0]-1+(j+r2[0]-1)*tam[0]+l*tam[0]*tam[1]+k*tam[0]*
    tam[1]*(ts->E+1)];
(*xn2)[i+r1[0]-1+(j+r2[0]-1)*tam[0]+n*tam[0]*tam[1]] = (*xn2)[i+r1[0]-1+(j+r2
    [0]-1)*tam[0]+n*tam[0]*tam[1]] + (*TAux)[i+j*longr1+n*longr1*longr2+1*
    longr1*longr2*tam[2]+l*longr1*longr2*tam[2]*ts->nt+k*longr1*longr2*tam[2]*ts
    ->nt*(ts->E+1)] * ts->BB[i+r1[0]-1+(j+r2[0]-1)*tam[0]+l*tam[0]*tam[1]+k*tam
    [0]*tam[1]*(ts->E+1)];

}

}

}
```

```
}


}









}
```

- **Función *interpolar*:**

```
#include "createbspline.h"

#include <stdio.h>

#include<cmath>
#include <stdlib.h>
#include <math.h>

#include<iostream>

#include "spline.h"


void interpolar (float **IT, float **xn1, float **xn2, float *I, int *caja,
    unsigned int margen1,unsigned int margen2,unsigned int margen3, int tam[])
{

float alpha=0;
float beta=0;

float w00=0;
float w01=0;
float w10=0;
float w11=0;



//para recorrer la primera componente de xn[esta][][]
for(int i= (caja[1+0]-margen1)-1; i<(caja[1+1*2]+margen1);i++)
{

//para recorrer la segunda componente de xn[][esta][]
for (int j=(caja[0+0]-margen2)-1; j<(caja[0+1*2]+margen2);j++)

{

//para recorrer la tercera componente de xn[][][esta]
for (int n=0; n<tam[2];n++)
{
int xn1_f=floor ((*xn1)[i+j*tam[0]+n*tam[0]*tam[1]])-1;
int xn2_f=floor ((*xn2)[i+j*tam[0]+n*tam[0]*tam[1]])-1;


int xn1_c=1+xn1_f;
int xn2_c=1+xn2_f;
```

```
alpha = (*xn1)[i+j*tam[0]+n*tam[0]*tam[1]]-1 - xn1_f;
beta  = (*xn2)[i+j*tam[0]+n*tam[0]*tam[1]]-1 - xn2_f;


//definimos los pesos, que es el desplazamiento producido en cada punto
w00 = (1-alpha)*(1-beta);
w10 = (1-alpha)*(beta);
w01 = (alpha)*(1-beta);
w11 = (alpha)*(beta);



(*IT)[i+j*tam[0]+n*tam[0]*tam[1]] = w00 * I[int(xn2_f)+(int(xn1_f))*tam[0]+n*tam
    [0]*tam[1]] + w01 * I[int(xn2_f)+(int(xn1_c))*tam[0]+n*tam[0]*tam[1]] + w10
    * I[int(xn2_c)+(int(xn1_f))*tam[0]+n*tam[0]*tam[1]] + w11 * I[int(xn2_c)+(
    int(xn1_c))*tam[0]+n*tam[0]*tam[1]];



}
}
}




}
```

- **Función *evolucion*:**

```
#include "createbspline.h"

#include <math.h>


void evolucion (float *TDif, float *HDif, float *Wn,  float *T, float H[], int
    iter, int flagW, int lim[], int tam[])
{
//caso B-splines


for(int i=0; i<lim[3]; i++)
{
for(int j=0; j<lim[2]; j++)
{
for(int k=0; k<tam[2]; k++)
{
(*TDif)+=T[i+j*lim[2]+k*lim[2]*lim[3]+0]*T[i+j*lim[2]+k*lim[2]*lim[3]+0];
(*TDif)+=T[i+j*lim[2]+k*lim[2]*lim[3]+1*lim[2]*lim[3]*tam[2]]*T[i+j*lim[2]+k*lim
    [2]*lim[3]+1*lim[2]*lim[3]*tam[2]];
}
}
}
(*TDif)=sqrt((*TDif));
(*HDif)=H[iter-1]-H[iter];
//adaptacion de Wn, flagW la activa

if (flagW)
{
if ((*HDif)>0)
(*Wn)=(*Wn)*1.2;
else
(*Wn)=(*Wn)/2;

}
```

```
}
```

- **Función *optimizador en OpenCL*:**

```
#include<iostream>
#include<cstdlib>
#include<cstdio>
#include<string>
#include<cmath>
#ifdef __APPLE__
#include<OpenCL/opencl.h>
#else
#include<CL/cl.h>
#endif


#include "mascara.h"
//#include "baseDatos.h"
#include "createbspline.h"
#include "optimizador.h"
#include <string.h>




#define IMG_SIZE 100
#define MAX_SRC_SIZE (0x100000)

#define MAX_SOURCE_SIZE (0x100000)




using namespace std;


void err_check( int err, string err_code ) {
if ( err != CL_SUCCESS ) {
cout << "Error: " << err_code << "(" << err << ")" << endl;
exit(-1);
}
}
// <20160404-Javier> Te pego aqu esta macro til para comprobar errores en
    llamadas de OpenCL
#define CL_CHECK(_expr)                                                         \
do {                                                                              \
if ((cl_int)_expr == CL_SUCCESS)                                                \
break;                                                                        \
fprintf(stderr, "OpenCL Error: '%s' returned %d!\n", #_expr, (int)_expr);     \
abort();                                                                      \
} while (0)




//OPTIMIZADOR se encarga del bucle principal del algoritmo de registrado

//Entradas
//Defino los parametros requeridos para el gradiente; struct %parametros
```

```
//-nmax=50;
//-et=0.01;%0.01 pixel para tener una convergencia plena.
//-eh=0.005;%es un 0.5% del inicial


//term terminos de suavidad; pesos y derivadas temporales y espaciales

//Salidas



void optimizador(int tam[], Bspline_t* ts,
float *IC, float **IT,
Datos_t datos, int **X, Parametros_t parametros,
float Wn, int *caja, int flagW)
{

//X  y caja es la mascara que define la region de interes
//interpolacion='linear';
//metric='varianza';
//IC: imagenes originales
//ts: parametros de transformacion (guarda la matriz bspline)
//Wn: matriz de paso
//flagW: flag que activa la adaptacion de Wn
//Wn: matriz de paso

//-----------------------------------PARA LA TRANSFORMACION-----------------------------
//Numero de instantes temporales
int    N = tam[2];

//x es la malla original de los puntos de control
float *x1=(float *)malloc(tam[0]*tam[1]*sizeof(float));
float *x2=(float *)malloc(tam[0]*tam[1]*sizeof(float));

//xn es la transformacion de la malla original de los puntos de control, es decir,
    los puntos de control despues de cada desplazamiento
float *xn1=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float));
float *xn2=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float));

//T es la matriz de transformaciones
int xT = ts->C[1][0] + ts->C[1][1] + 1, yT = ts->C[0][0] + ts->C[0][1] + 1, zT = N,
    tT = ts->nt;
float *T=(float *)malloc(xT*yT*zT*tT*sizeof(float));

//-----------------------------------PARA El GRADIENTE-----------------------------
//dx es el gradiente de la imagen
float *dx1=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float));
float *dx2=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float));

//dy es gradiente de la metrica
float *dy=(float *)malloc(tam[0]*tam[1]*tam[2]*sizeof(float));

//ts->BB sera el gradiente de la transformacion


//tama o de ts->BBg, que es la matriz de productos B-spline
int xlim=2*ceil(ts->rp[0])+1, ylim=2*ceil(ts->rp[1])+1, zlim=ts->C[0][0]+ts->C
    [0][1]+1, tlim=ts->C[1][0]+ts->C[1][1]+1;
int lim[4]={xlim, ylim, zlim, tlim};//17 17 39 39


//dV sera el gradiente toal de las imagenes empleado en el metodo del gradiente
    descendente
int xdV = lim[0], ydV  = lim[1], zdV  = tam[2], tdV = lim[2], pdV = lim[3], qdV = ts
    ->nt;
```

```
float *dV=(float *)malloc(xdV*ydV*zdV*tdV*pdV*qdV*sizeof(float ));



//-----------------------------------PARA LA METRICA-------------------------------
//media para el calculo de la metrica
float *media=(float *)malloc(tam[0]*tam[1]*sizeof(float ));
//metrica de las imagenes en cada punto, del tama o de las imagenes de entrada
float *V=(float *)malloc(tam[0]*tam[1]*sizeof(float ));







//H2 es el vector que guarda las metricas de cada iteracion




int long_r1 = caja[0+1*2] - caja[0+0] + 1;

//int *r1 = (int *)malloc(sizeof(int)*long_r1);


int long_r2 = caja[1+1*2] - caja[1+0] + 1;


int xTAux = long_r1, yTAux = long_r2, zTAux = N, tTAux = ts->nt, pTAux = (ts->E + 1)
    , qTAux = (ts->E + 1);
float *TAux=(float *)malloc(xTAux*yTAux*zTAux*tTAux*pTAux*qTAux*sizeof(float ));









int r1[long_r1];
int r2[long_r2];

int k1=0;

for (int i=caja[0+0]; i<=caja[0+1*2];i++)
{
r1[k1]=i;
//printf("%d  ",r1[k1]);
k1++;

}

int k2=0;

for (int i=caja[1+0]; i<=caja[1+1*2];i++)
{
r2[k2]=i;
//printf("%d  ",r2[k2]);
k2++;
}
```

```
//calculo de un 2.5% de la imagen
unsigned int margen1 = ceil(float(tam[0])/40);
unsigned int margen2 = ceil(float(tam[1])/40);
unsigned int margen3 = ceil(float(tam[2])/40);




int max=parametros.nmax;

float H=0;

//H1 va a ser el vector que guarda las metricas en cada iteracion
float *H1 = (float *)malloc((max+1) * sizeof(float ));

//TDif es la diferencia de la norma de la matriz de transformacion


float *TDif = (float *)malloc((max+1) * sizeof(float ));
//Hdif es la diferencia entre la metrica de las dos ultimas iteraciones
float *HDif = (float *)malloc((max+1) * sizeof(float ));



//H2 es el array multidimensional a proyectar
int xH = lim[3], yH = lim[2], zH = N, tH = ts->nt;
float *H2=(float *)malloc(xH*yH*zH*tH*sizeof(float ));
//proyH2 es la proyeccion del array sobre su media
float *proyH2=(float *)malloc(xH*yH*zH*tH*sizeof(float ));
//Dif es lo que voy a restar a T para obtener la nueva matriz T de la transformacion
    .
//Vamos a obtener unos nuevos pesos Wn en cada iteracion, pues Dif=Wn*proyH2, y con
    ello definimos  T=T-Dif
float *Dif=(float *)malloc(xH*yH*zH*tH*sizeof(float ));


//media2 para obtener la proyeccion proyH2
int zmedia2 = ts->nt, xmedia2 = lim[3], ymedia2 = lim[2];
float *media2=(float *)malloc(xmedia2*ymedia2*zmedia2*sizeof(float ));



//float metrica (float *V, float* IT, Datos_t datos, Term_t* term, int caja[2][2],
    int tam[], float*media, float* dy)
coste(&H1[0],&H2,metrica (V, IC,datos, caja, tam, media, &dy,ts),(*X),0,tam,ts,lim,
    dV);




//parametros de condicion de parada:
float p2 = parametros.et*tam[2]*ts->nt;
float p4 = H1[0]*parametros.eh;
int iter=1;
int condicion_parada=0;




//Inicializa x1 y x2
for (int i=0; i<tam[0]; i++) {
for(int j=0; j<tam[1]; j++) {
x1[j+i*tam[0]] = i+1;
```

```
x2[j+i*tam[0]] = j+1;
}
}

//Inicializa xn1 y xn2
for(int n=0;n<tam[2];n++)
{
for(int i=0;i<tam[0];i++)
{
for(int j=0;j<tam[1];j++)
{
xn1[j+i*tam[0]+n*tam[0]*tam[1]] = x1[j+i*tam[0]];
xn2[j+i*tam[0]+n*tam[0]*tam[1]] = x2[j+i*tam[0]];
}
}
}




//-------------------------------------------OPENCL
    ---------------------------------------

unsigned int tam0 = tam[0];
unsigned int tam1 = tam[1];
unsigned int tam2 = tam[2];

unsigned int lim0 = lim[0];
unsigned int lim1 = lim[1];
unsigned int lim2 = lim[2];
unsigned int lim3 = lim[3];

unsigned int longr1 = long_r1;
unsigned int longr2 = long_r2;

unsigned int tsE = ts->E+1;
unsigned int nt = ts->nt;



unsigned int total_length =lim[0]*lim[1]*tam[2]*lim[2]*lim[3]*ts->nt;
unsigned int total_length2=lim[3]*lim[2]*tam[2]*ts->nt;
unsigned int total_length3=lim[3]*lim[2]*ts->nt;
unsigned int length_xn=tam0*tam1*tam2;
unsigned int length_x=tam[0]*tam[1];
unsigned int length_coefg1=lim[2]*ts->nt;
unsigned int length_coefg2=lim[3]*ts->nt;
unsigned int length_BBg=lim[0]*lim[1]*lim[2]*lim[3];
unsigned int length_TAux=long_r2*long_r1*tam[2]*ts->nt*(ts->E+1)*(ts->E+1);
unsigned int length_BB=tam[0]*tam[1]*(ts->E+1)*(ts->E+1);


cl_platform_id platform_id = NULL;
cl_device_id device_id = NULL;
cl_context context = NULL;
cl_command_queue command_queue = NULL;
cl_mem dVmobj = NULL;
cl_mem dV2mobj = NULL;

cl_mem H2mobj = NULL;
cl_mem proyH2mobj = NULL;
cl_mem Difmobj = NULL;
cl_mem Tmobj = NULL;
cl_mem media2mobj = NULL;
```

```
cl_mem ITmobj = NULL;
cl_mem xn1mobj = NULL;
cl_mem xn2mobj = NULL;
cl_mem ICmobj = NULL;
cl_mem x1mobj = NULL;
cl_mem x2mobj = NULL;

cl_mem TAuxmobj = NULL;
cl_mem r1mobj = NULL;
cl_mem r2mobj = NULL;

cl_mem BBmobj = NULL;


cl_mem dx1mobj = NULL;
cl_mem dx2mobj = NULL;
cl_mem dymobj = NULL;
cl_mem coefg1mobj = NULL;
cl_mem coefg2mobj = NULL;
cl_mem coef1mobj = NULL;
cl_mem coef2mobj = NULL;
cl_mem BBgmobj = NULL;
cl_mem Xmobj = NULL;


cl_mem Vmobj = NULL;
cl_mem mediamobj = NULL;

cl_mem cajamobj = NULL;


cl_mem Wnmobj = NULL;
cl_float Wm = Wn;

//cl_mem Hmobj = NULL;
//cl_float HH = H;


cl_program program = NULL;
cl_kernel kernel[15] = {NULL, NULL, NULL, NULL, NULL, NULL, NULL,NULL,NULL, NULL,
    NULL,NULL,NULL,NULL,NULL,NULL};
cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret;
//cl_int ret1;



FILE *fpcl;

//const char fileName[] = "./taskParallel.cl";
const char fileName[] = "./kernels.cl";
size_t source_size;
char *source_str;

//Load kernel source file
fpcl = fopen(fileName, "rb");
if (!fpcl) {
fprintf(stderr, "Failed to load kernel.\n");
exit(1);
}
source_str = (char *)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fpcl);
fclose(fpcl);
```

```
//Get platform/device information
ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms); CL_CHECK(ret);
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &
    ret_num_devices); CL_CHECK(ret);

//Create OpenCL Context
context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret); CL_CHECK(ret);

//Create command queue
command_queue = clCreateCommandQueue(context, device_id,
    CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &ret); CL_CHECK(ret);


// Create buffer object
dVmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, total_length* sizeof(float),
    NULL, &ret); CL_CHECK(ret);
dV2mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, total_length* sizeof(float),
    NULL, &ret); CL_CHECK(ret);
H2mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, total_length2*sizeof(float),
    NULL, &ret); CL_CHECK(ret);
proyH2mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, total_length2*sizeof(float),
     NULL, &ret); CL_CHECK(ret);
Difmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, total_length2*sizeof(float),
    NULL, &ret); CL_CHECK(ret);
Tmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, total_length2*sizeof(float), NULL
    , &ret); CL_CHECK(ret);
media2mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, total_length3*sizeof(float),
     NULL, &ret); CL_CHECK(ret);
ITmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_xn*sizeof(float), NULL, &
    ret); CL_CHECK(ret);
xn1mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_xn*sizeof(float), NULL,
    &ret); CL_CHECK(ret);
xn2mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_xn*sizeof(float), NULL,
    &ret); CL_CHECK(ret);
ICmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_xn*sizeof(float), NULL, &
    ret); CL_CHECK(ret);
x1mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_x*sizeof(float), NULL, &
    ret); CL_CHECK(ret);
x2mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_x*sizeof(float), NULL, &
    ret); CL_CHECK(ret);
Vmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_x*sizeof(float), NULL, &
    ret); CL_CHECK(ret);
mediamobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_x*sizeof(float), NULL,
     &ret); CL_CHECK(ret);
BBmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_BB*sizeof(float), NULL, &
    ret); CL_CHECK(ret);
TAuxmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_TAux*sizeof(float),
    NULL, &ret); CL_CHECK(ret);
r1mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, longr1*sizeof(int), NULL, &ret);
     CL_CHECK(ret);
r2mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, longr2*sizeof(int), NULL, &ret);
     CL_CHECK(ret);
dx1mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_xn*sizeof(float), NULL,
    &ret); CL_CHECK(ret);
dx2mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_xn*sizeof(float), NULL,
    &ret); CL_CHECK(ret);
dymobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_xn*sizeof(float), NULL, &
    ret); CL_CHECK(ret);
coefg1mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_coefg1*sizeof(int),
    NULL, &ret); CL_CHECK(ret);
coefg2mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_coefg2*sizeof(int),
    NULL, &ret); CL_CHECK(ret);
coef1mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, tam[0]*ts->nt*sizeof(int),
    NULL, &ret); CL_CHECK(ret);
```

```
coef2mobj = clCreateBuffer(context, CL_MEM_READ_WRITE, tam[1]*ts->nt*sizeof(int),
    NULL, &ret); CL_CHECK(ret);
BBgmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, length_BBg*sizeof(float), NULL,
     &ret); CL_CHECK(ret);
Xmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, tam0*tam1*sizeof(int), NULL, &ret
    ); CL_CHECK(ret);
cajamobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 2*2*sizeof(int), NULL, &ret);
    CL_CHECK(ret);
Wnmobj= clCreateBuffer( context, CL_MEM_READ_WRITE, sizeof(cl_float), NULL, &ret );
//Hmobj= clCreateBuffer( context, CL_MEM_READ_WRITE, sizeof(cl_float), NULL, &ret );


// Copy input data to memory buffer
ret = clEnqueueWriteBuffer(command_queue, dVmobj, CL_TRUE, 0, total_length *sizeof(
    float), dV,      0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, dV2mobj, CL_TRUE, 0, total_length *sizeof(
    float), dV,      0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, H2mobj, CL_TRUE, 0, total_length2*sizeof(
    float), H2,      0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, proyH2mobj, CL_TRUE, 0, total_length2*
    sizeof(float), proyH2, 0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, Difmobj, CL_TRUE, 0, total_length2*sizeof(
    float), Dif,     0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, Tmobj, CL_TRUE, 0, total_length2*sizeof(
    float), T,       0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, media2mobj, CL_TRUE, 0, total_length3*
    sizeof(float), media2,      0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, ICmobj, CL_TRUE, 0, length_xn*sizeof(float
    ), IC,   0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, ITmobj, CL_TRUE, 0, length_xn*sizeof(float
    ), IC,   0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, x1mobj, CL_TRUE, 0, length_x*sizeof(float)
    , x1,     0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, x2mobj, CL_TRUE, 0, length_x*sizeof(float)
    , x2 ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, dx1mobj, CL_TRUE, 0, length_xn*sizeof(
    float), dx1 ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, dx2mobj, CL_TRUE, 0, length_xn*sizeof(
    float), dx2 ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, dymobj, CL_TRUE, 0, length_xn*sizeof(float
    ), dy ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, coefg1mobj, CL_TRUE, 0, length_coefg1*
    sizeof(int), ts->coefg1 ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, coefg2mobj, CL_TRUE, 0, length_coefg2*
    sizeof(int), ts->coefg2 ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, coef1mobj, CL_TRUE, 0, tam[0]*ts->nt*
    sizeof(int), ts->coef1 ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, coef2mobj, CL_TRUE, 0, tam[1]*ts->nt*
    sizeof(int), ts->coef2 ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, BBgmobj, CL_TRUE, 0, length_BBg*sizeof(
    float), ts->BBg ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, BBmobj, CL_TRUE, 0, length_BB*sizeof(float
    ), ts->BB ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, TAuxmobj, CL_TRUE, 0, length_TAux*sizeof(
    float), TAux ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, Xmobj, CL_TRUE, 0, tam0*tam1*sizeof(int),
    (*X) ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, r1mobj, CL_TRUE, 0, long_r1*sizeof(int),
    r1 ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, r2mobj, CL_TRUE, 0, long_r2*sizeof(int),
    r2 ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer(command_queue, cajamobj, CL_TRUE, 0, 2*2*sizeof(int),
    caja ,    0, NULL, NULL); CL_CHECK(ret);
ret = clEnqueueWriteBuffer( command_queue, Wnmobj, CL_TRUE, 0, sizeof(cl_float), &
    Wnn, 0, NULL, NULL );
```

```
//ret = clEnqueueWriteBuffer( command_queue, Hmobj, CL_TRUE, 0, sizeof(cl_int), &HH,
     0, NULL, NULL );

// Create kernel from source
program = clCreateProgramWithSource(context, 1, (const char **)&source_str, (const
    size_t *)&source_size, &ret);
CL_CHECK(ret);
cout << "Creacion del programa OK" << endl;

//Build
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);  CL_CHECK(ret);
cout << "Compilacion del programa OK" << endl;


// Create task parallel OpenCL kernel
kernel[0] = clCreateKernel(program, "inicVect", &ret); CL_CHECK(ret);
kernel[1] = clCreateKernel(program, "inicVect", &ret); CL_CHECK(ret);
kernel[2] = clCreateKernel(program, "inicVect", &ret); CL_CHECK(ret);
kernel[3] = clCreateKernel(program, "inicVect", &ret); CL_CHECK(ret);
kernel[4] = clCreateKernel(program, "inicVect3", &ret); CL_CHECK(ret);
kernel[5] = clCreateKernel(program, "inicVect3", &ret); CL_CHECK(ret);
kernel[6] = clCreateKernel(program, "gradiente_imagen", &ret); CL_CHECK(ret);
kernel[7] = clCreateKernel(program, "gradiente", &ret); CL_CHECK(ret);
kernel[8] = clCreateKernel(program, "coste1", &ret); CL_CHECK(ret);
kernel[9] = clCreateKernel(program, "proyeccion", &ret); CL_CHECK(ret);
kernel[10] = clCreateKernel(program, "transformar", &ret); CL_CHECK(ret);
kernel[11] = clCreateKernel(program, "transformar2", &ret); CL_CHECK(ret);
kernel[12] = clCreateKernel(program, "interpolar", &ret); CL_CHECK(ret);
kernel[13] = clCreateKernel(program, "metrica", &ret); CL_CHECK(ret);

// Set OpenCL kernel arguments
//los de inicVect
ret = clSetKernelArg(kernel[0], 0, sizeof(cl_mem), (void *)&dVmobj); CL_CHECK(ret);

ret = clSetKernelArg(kernel[1], 0, sizeof(cl_mem), (void *)&H2mobj); CL_CHECK(ret);

ret = clSetKernelArg(kernel[2], 0, sizeof(cl_mem), (void *)&proyH2mobj); CL_CHECK(
    ret);

ret = clSetKernelArg(kernel[3], 0, sizeof(cl_mem), (void *)&Difmobj); CL_CHECK(ret);


//los de inicVect3
ret = clSetKernelArg(kernel[4], 0, sizeof(cl_mem), (void *)&xn1mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[4], 1, sizeof(cl_mem), (void *)&x1mobj); CL_CHECK(ret);


ret = clSetKernelArg(kernel[5], 0, sizeof(cl_mem), (void *)&xn2mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[5], 1, sizeof(cl_mem), (void *)&x2mobj); CL_CHECK(ret);


ret = clSetKernelArg(kernel[6], 0, sizeof(cl_mem), (void *)&dx1mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[6], 1, sizeof(cl_mem), (void *)&dx2mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[6], 2, sizeof(cl_mem), (void *)&ITmobj); CL_CHECK(ret);


ret = clSetKernelArg(kernel[7], 0, sizeof(cl_mem), (void *)&dVmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[7], 1, sizeof(cl_mem), (void *)&dx1mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[7], 2, sizeof(cl_mem), (void *)&dx2mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[7], 3, sizeof(cl_mem), (void *)&dymobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[7], 4, sizeof(cl_mem), (void *)&ITmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[7], 5, sizeof(cl_mem), (void *)&coefg1mobj); CL_CHECK(
    ret);
ret = clSetKernelArg(kernel[7], 6, sizeof(cl_mem), (void *)&coefg2mobj); CL_CHECK(
    ret);
```

```
ret = clSetKernelArg(kernel[7], 7, sizeof(cl_mem), (void *)&BBgmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[7], 8, sizeof(unsigned int), &tam0); CL_CHECK(ret);
ret = clSetKernelArg(kernel[7], 9, sizeof(unsigned int), &tam1); CL_CHECK(ret);
ret = clSetKernelArg(kernel[7], 10, sizeof(unsigned int), &tam2); CL_CHECK(ret);
ret = clSetKernelArg(kernel[7], 11, sizeof(unsigned int), &lim0); CL_CHECK(ret);
ret = clSetKernelArg(kernel[7], 12, sizeof(unsigned int), &lim1); CL_CHECK(ret);
ret = clSetKernelArg(kernel[7], 13, sizeof(unsigned int), &lim2); CL_CHECK(ret);
ret = clSetKernelArg(kernel[7], 14, sizeof(unsigned int), &lim3); CL_CHECK(ret);


ret = clSetKernelArg(kernel[8], 0, sizeof(cl_mem), (void *)&H2mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[8], 1, sizeof(cl_mem), (void *)&dVmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[8], 2, sizeof(cl_mem), (void *)&Xmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[8], 3, sizeof(cl_mem), (void *)&coefg1mobj); CL_CHECK(
    ret);
ret = clSetKernelArg(kernel[8], 4, sizeof(cl_mem), (void *)&coefg2mobj); CL_CHECK(
    ret);
ret = clSetKernelArg(kernel[8], 5, sizeof(unsigned int), &tam0); CL_CHECK(ret);
ret = clSetKernelArg(kernel[8], 6, sizeof(unsigned int), &tam1); CL_CHECK(ret);
ret = clSetKernelArg(kernel[8], 7, sizeof(unsigned int), &tam2); CL_CHECK(ret);
ret = clSetKernelArg(kernel[8], 8, sizeof(unsigned int), &lim0); CL_CHECK(ret);
ret = clSetKernelArg(kernel[8], 9, sizeof(unsigned int), &lim1); CL_CHECK(ret);
ret = clSetKernelArg(kernel[8], 10, sizeof(unsigned int), &lim2); CL_CHECK(ret);
ret = clSetKernelArg(kernel[8], 11, sizeof(unsigned int), &lim3); CL_CHECK(ret);


ret = clSetKernelArg(kernel[9], 0, sizeof(cl_mem), (void *)&proyH2mobj); CL_CHECK(
    ret);
ret = clSetKernelArg(kernel[9], 1, sizeof(cl_mem), (void *)&Tmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[9], 2, sizeof(cl_mem), (void *)&Difmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[9], 3, sizeof(cl_mem), (void *)&H2mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[9], 4, sizeof(cl_mem), (void *)&media2mobj); CL_CHECK(
    ret);
ret = clSetKernelArg(kernel[9], 5, sizeof(cl_mem), (void *)&Wnmobj); CL_CHECK(ret);



ret = clSetKernelArg(kernel[10], 0, sizeof(cl_mem), (void *)&TAuxmobj); CL_CHECK(ret
    );
ret = clSetKernelArg(kernel[10], 1, sizeof(cl_mem), (void *)&Tmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[10], 2, sizeof(cl_mem), (void *)&r1mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[10], 3, sizeof(cl_mem), (void *)&r2mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[10], 4, sizeof(cl_mem), (void *)&coef1mobj); CL_CHECK(
    ret);
ret = clSetKernelArg(kernel[10], 5, sizeof(cl_mem), (void *)&coef2mobj); CL_CHECK(
    ret);
ret = clSetKernelArg(kernel[10], 6, sizeof(unsigned int), &lim0); CL_CHECK(ret);
ret = clSetKernelArg(kernel[10], 7, sizeof(unsigned int), &lim1); CL_CHECK(ret);
ret = clSetKernelArg(kernel[10], 8, sizeof(unsigned int), &lim2); CL_CHECK(ret);
ret = clSetKernelArg(kernel[10], 9, sizeof(unsigned int), &lim3); CL_CHECK(ret);
ret = clSetKernelArg(kernel[10], 10, sizeof(unsigned int), &tsE); CL_CHECK(ret);
ret = clSetKernelArg(kernel[10], 11, sizeof(unsigned int), &nt); CL_CHECK(ret);
ret = clSetKernelArg(kernel[10], 12, sizeof(unsigned int), &longr1); CL_CHECK(ret);
ret = clSetKernelArg(kernel[10], 13, sizeof(unsigned int), &longr2); CL_CHECK(ret);


ret = clSetKernelArg(kernel[11], 0, sizeof(cl_mem), (void *)&xn1mobj); CL_CHECK(ret)
    ;
ret = clSetKernelArg(kernel[11], 1, sizeof(cl_mem), (void *)&xn2mobj); CL_CHECK(ret)
    ;
ret = clSetKernelArg(kernel[11], 2, sizeof(cl_mem), (void *)&TAuxmobj); CL_CHECK(ret
    );
ret = clSetKernelArg(kernel[11], 3, sizeof(cl_mem), (void *)&r1mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[11], 4, sizeof(cl_mem), (void *)&r2mobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[11], 5, sizeof(cl_mem), (void *)&BBmobj); CL_CHECK(ret);
```

```
ret = clSetKernelArg(kernel[11], 6, sizeof(unsigned int), &tsE); CL_CHECK(ret);
ret = clSetKernelArg(kernel[11], 7, sizeof(unsigned int), &nt); CL_CHECK(ret);
ret = clSetKernelArg(kernel[11], 8, sizeof(unsigned int), &tam0); CL_CHECK(ret);
ret = clSetKernelArg(kernel[11], 9, sizeof(unsigned int), &tam1); CL_CHECK(ret);

ret = clSetKernelArg(kernel[12], 0, sizeof(cl_mem), (void *)&ITmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[12], 1, sizeof(cl_mem), (void *)&xn1mobj); CL_CHECK(ret)
    ;
ret = clSetKernelArg(kernel[12], 2, sizeof(cl_mem), (void *)&xn2mobj); CL_CHECK(ret)
    ;
ret = clSetKernelArg(kernel[12], 3, sizeof(cl_mem), (void *)&ICmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[12], 4, sizeof(cl_mem), (void *)&cajamobj); CL_CHECK(ret
    );
ret = clSetKernelArg(kernel[12], 5, sizeof(unsigned int), &margen1); CL_CHECK(ret);
ret = clSetKernelArg(kernel[12], 6, sizeof(unsigned int), &margen2); CL_CHECK(ret);
ret = clSetKernelArg(kernel[12], 7, sizeof(unsigned int), &margen3); CL_CHECK(ret);
ret = clSetKernelArg(kernel[12], 8, sizeof(unsigned int), &tam0); CL_CHECK(ret);
ret = clSetKernelArg(kernel[12], 9, sizeof(unsigned int), &tam1); CL_CHECK(ret);


ret = clSetKernelArg(kernel[13], 0, sizeof(cl_mem), (void *)&Vmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[13], 1, sizeof(cl_mem), (void *)&dymobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[13], 2, sizeof(cl_mem), (void *)&ITmobj); CL_CHECK(ret);
ret = clSetKernelArg(kernel[13], 3, sizeof(cl_mem), (void *)&mediamobj); CL_CHECK(
    ret);
ret = clSetKernelArg(kernel[13], 4, sizeof(unsigned int), &tam2); CL_CHECK(ret);




//para pasarle los tama os y dimensiones de los work groups
size_t total_length_vec[] = {total_length, total_length2,total_length2,total_length2
    ,length_xn,total_length};
size_t total_size_xn[] = {longr2, longr1, tam2};

int tam0_interpolar = (caja[1+1*2]+margen1)-((caja[1+0]-margen1)-1);
int tam1_interpolar =  (caja[0+1*2]+margen2)-((caja[0+0]-margen2)-1);

size_t total_size_interpolar[] = {tam0_interpolar, tam1_interpolar, tam2};
size_t total_size_x[] = {tam0, tam1};
size_t total_size_xn1[] = {length_x, tam2};
size_t total_size_xn2[] = {length_x ,tam2};
size_t total_size_IT[] = {tam0, tam1, tam2};
size_t total_size_dV[] = {lim0*lim1, tam2, lim2*lim3};
size_t total_size_H2[] = {lim3, lim2, tam2};
size_t total_size_TAux[] = {longr2*longr1, tam2, (ts->E+1)*(ts->E+1)};

while(condicion_parada==0)
{

for (int i=0; i < 4; i++)
{

ret = clEnqueueNDRangeKernel(
command_queue,        // cl_command_queue command_queue
kernel[i],            // cl_kernel kernel
1,                                  // cl_uint work_dim
NULL,                      // const size_t *global_work_offset
&total_length_vec[i],// const size_t *global_work_size
NULL,                          // const size_t *local_work_offset
0,                                  // cl_uint num_events_in_wait_list
NULL,                          // const cl_event *event_wait_list
NULL                       // cl_event *event
);
 CL_CHECK(ret);
```

```
}



ret = clEnqueueNDRangeKernel(
command_queue,
kernel[4],
2,
NULL,
(size_t *)&total_size_xn1,
NULL,
0,
NULL,
NULL
);
CL_CHECK(ret);


 ret = clEnqueueNDRangeKernel(
 command_queue,
 kernel[5],
 2,
 NULL,
 (size_t *)&total_size_xn2,
 NULL,
 0,
 NULL,
 NULL
 );
 CL_CHECK(ret);



 ret = clEnqueueNDRangeKernel(
 command_queue,
 kernel[6],
 3,
 NULL,
 (size_t *)&total_size_IT,
 NULL,
 0,
 NULL,
 NULL
 );
 CL_CHECK(ret);



 ret = clEnqueueNDRangeKernel(
 command_queue,
 kernel[7],
 3,
 NULL,
 (size_t *)&total_size_dV,
 NULL,
 0,
 NULL,
 NULL
 );
 CL_CHECK(ret);



 ret = clEnqueueNDRangeKernel(
```

```
command_queue,
kernel[8],
3,
NULL,
(size_t *)&total_size_H2,
NULL,
0,
NULL,
NULL
);
CL_CHECK(ret);


ret = clEnqueueNDRangeKernel(
command_queue,
kernel[8],
3,
NULL,
(size_t *)&total_size_H2,
NULL,
0,
NULL,
NULL
);
CL_CHECK(ret);


ret = clEnqueueNDRangeKernel(
command_queue,
kernel[9],
3,
NULL,
(size_t *)&total_size_H2,
NULL,
0,
NULL,
NULL
);
CL_CHECK(ret);




ret = clEnqueueNDRangeKernel(
command_queue,
kernel[10],
3,
NULL,
(size_t *)&total_size_TAux,
NULL,
0,
NULL,
NULL
);
CL_CHECK(ret);


ret = clEnqueueNDRangeKernel(
command_queue,
kernel[11],
3,
NULL,
(size_t *)&total_size_xn,
NULL,
0,
```

```
NULL,
NULL
);
CL_CHECK(ret);


 ret = clEnqueueNDRangeKernel(
 command_queue,
 kernel[12],
 3,
 NULL,
 (size_t *)&total_size_interpolar,
 NULL,
 0,
 NULL,
 NULL
 );
 CL_CHECK(ret);


 ret = clEnqueueNDRangeKernel(
 command_queue,
 kernel[13],
 2,
 NULL,
 (size_t *)&total_size_x,
 NULL,
 0,
 NULL,
 NULL
 );
 CL_CHECK(ret);


ret = clEnqueueReadBuffer(command_queue, Difmobj, CL_TRUE, 0, total_length2*sizeof(
    float), Dif, 0, NULL, NULL); CL_CHECK(ret);

ret = clEnqueueReadBuffer(command_queue, ITmobj, CL_TRUE, 0, length_xn*sizeof(float)
    , (*IT), 0, NULL, NULL); CL_CHECK(ret);


ret = clEnqueueReadBuffer(command_queue, Vmobj, CL_TRUE, 0, length_x*sizeof(float),
    V, 0, NULL, NULL); CL_CHECK(ret);




//Dos funciones que da problemas en openCl ya que van acumulando, pero no se aprecia
     el tiempo que tarda, ya que son bucles con pocas iteraciones
coste(&H1[iter],&H2,V,(*X),0,tam,ts,lim,dV);
evolucion(&TDif[iter-1], &HDif[iter-1], &Wn, Dif, H1, iter, flagW,lim,tam);

//actualizo los nuevos pesos
Wnn = Wn;
ret = clEnqueueWriteBuffer( command_queue, Wnmobj, CL_TRUE, 0, sizeof(cl_float), &
    Wnn, 0, NULL, NULL );


//Comparacion: Tdif con p2 y Hdif con p4 y nmax con iter
//vamos con la condicion de parada
//if(((max)==iter)||((TDif[iter-1]==p2)&&(HDif[iter-1]==p4)))
if(iter==1||((TDif[iter-1]==p2)&&(HDif[iter-1]==p4)))
{
condicion_parada=1;
}
```

```
printf("\niter %d\n",iter);
iter=iter+1;









}//fin while


printf("\n\nH1--> %f, TDif--> %f HDif--> %f\n",H1[iter-1],TDif[iter-2],HDif[iter-2])
    ;


//libero
ret = clFlush(command_queue);
ret = clFinish(command_queue);
ret = clReleaseKernel(kernel[0]);
ret = clReleaseKernel(kernel[1]);
ret = clReleaseKernel(kernel[2]);
ret = clReleaseKernel(kernel[3]);
ret = clReleaseKernel(kernel[4]);
ret = clReleaseKernel(kernel[5]);
ret = clReleaseKernel(kernel[6]);
ret = clReleaseKernel(kernel[7]);
ret = clReleaseKernel(kernel[8]);
ret = clReleaseKernel(kernel[9]);
ret = clReleaseKernel(kernel[10]);
ret = clReleaseKernel(kernel[11]);
ret = clReleaseKernel(kernel[12]);
ret = clReleaseKernel(kernel[13]);
ret = clReleaseKernel(kernel[14]);

ret = clReleaseProgram(program);
ret = clReleaseMemObject(dVmobj);
ret = clReleaseMemObject(dV2mobj);
ret = clReleaseMemObject(Xmobj);
ret = clReleaseMemObject(H2mobj);
ret = clReleaseMemObject(proyH2mobj);
ret = clReleaseMemObject(Difmobj);
ret = clReleaseMemObject(Tmobj);
ret = clReleaseMemObject(media2mobj);
ret = clReleaseMemObject(ITmobj);
ret = clReleaseMemObject(xn1mobj);
ret = clReleaseMemObject(xn2mobj);
ret = clReleaseMemObject(r1mobj);
ret = clReleaseMemObject(r2mobj);
ret = clReleaseMemObject(Vmobj);
ret = clReleaseMemObject(mediamobj);
ret = clReleaseMemObject(TAuxmobj);
ret = clReleaseMemObject(ICmobj);
ret = clReleaseMemObject(x1mobj);
ret = clReleaseMemObject(x2mobj);
ret = clReleaseMemObject(dx1mobj);
ret = clReleaseMemObject(dx2mobj);
ret = clReleaseMemObject(dymobj);
ret = clReleaseMemObject(coefg1mobj);
ret = clReleaseMemObject(coefg2mobj);
ret = clReleaseMemObject(coef1mobj);
ret = clReleaseMemObject(coef2mobj);
```

```
ret = clReleaseMemObject(BBgmobj);
ret = clReleaseMemObject(Wnmobj);
ret = clReleaseMemObject(BBmobj);
ret = clReleaseMemObject(cajamobj);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseContext(context);
}
```

- **Fichero *kernel.cl*:**

```
__kernel void inicVect(__global float *a)
{
int id = get_global_id(0);
a[id] = 0.0;
}

__kernel void inicVect2(__global float *a,
                        __global float *b)
{
//Get our global thread ID
int id = get_global_id(0);
a[id] = b[id];
}



__kernel void inicVect3(__global float *a,
                        __global float *b)
{
//Get our global thread ID
int id_fil = get_global_id(0);
int id_col = get_global_id(1);
int filas = get_global_size(0); //longitud 400x400=160000
a[id_fil + id_col*filas] = b[id_fil];
}

__kernel void gradiente_imagen (__global float *dx1,
                                __global float *dx2,
                                __global float *IT)
{
int tam0 = get_global_size(0);
int tam1 = get_global_size(1);
int tam2 = get_global_size(2);

int i=get_global_id(0);
int j=get_global_id(1);
int k=get_global_id(2);

//para los bordes
int h=1;


//bordes para las variaciones en y (dx2)
//borde superior
if (i==0)
dx2[0+j*tam0+k*tam0*tam1]=IT[1+j*tam0+k*tam0*tam1]-IT[0+j*tam0+k*tam0*tam1];

//borde inferior
else if(i==tam0-1)
dx2[tam0-1+j*tam0+k*tam0*tam1]=IT[tam0-1+j*tam0+k*tam0*tam1]-IT[tam0-2+j*tam0+k*
    tam0*tam1];

//para lo que no son los bordes
else
```

```
dx2[i+j*tam0+k*tam0*tam1]=(IT[i+1+j*tam0+k*tam0*tam1]-IT[i-1+j*tam0+k*tam0*tam1
    ])/(2);


//bordes para las variaciones en x (dx1)
//borde izquierdo
if (j==0)
dx1[i+0+k*tam0*tam1]=(IT[i+1*tam0+k*tam0*tam1]-IT[i+0+k*tam0*tam1])/(h);

//borde derecho
else if(j==tam1-1)
dx1[i+(tam1-1)*tam0+k*tam0*tam1]=(IT[i+(tam1-1)*tam0+k*tam0*tam1]-IT[i+(tam1-2)*
    tam0+k*tam0*tam1])/(h);

//para lo que no son los bordes
else
dx1[i+j*tam0+k*tam0*tam1]=(IT[i+(j+1)*tam0+k*tam0*tam1]-IT[i+(j-1)*tam0+k*tam0*
    tam1])/(2);
}


__kernel void gradiente(__global float *dV,
                        __global float *dx1,
                        __global float *dx2,
                        __global float *dy,
                        __global float *IT,
                        __global int *coefg1,
                        __global int *coefg2,
                        __global float *BBg,
                        const unsigned int tam0,
                        const unsigned int tam1,
                        const unsigned int tam2,
                        const unsigned int lim0,
                        const unsigned int lim1,
                        const unsigned int lim2,
                        const unsigned int lim3)


{

int iaux = get_global_id(0);
float x=(float)iaux;
int j=iaux%lim0;
int i=iaux/lim0;
int n = get_global_id(1);
int kaux = get_global_id(2);
int k=kaux%lim2;
int l=kaux/lim2;

dV[j+i*lim0+n*lim0*lim1+k*lim0*lim1*tam2+l*lim0*lim1*tam2*lim2+0]=dy[j-1+coefg2[
    l+0]+(i-1+coefg1[k+0])*tam0+n*tam0*tam1]*dx1[j-1+coefg2[l+0]+(i-1+coefg1[k
    +0])*tam0+n*tam0*tam1]*BBg[j+i*lim0+k*lim0*lim1+l*lim0*lim1*lim2];

dV[j+i*lim0+n*lim0*lim1+k*lim0*lim1*tam2+l*lim0*lim1*tam2*lim2+1*lim0*lim1*tam2*
    lim2*lim3]=dy[j-1+coefg2[l+0]+(i-1+coefg1[k+0])*tam0+n*tam0*tam1]*dx2[j-1+
    coefg2[l+0]+(i-1+coefg1[k+0])*tam0+n*tam0*tam1]*BBg[j+i*lim0+k*lim0*lim1+l*
    lim0*lim1*lim2];
}


__kernel void coste1  (__global float *H2,
                       __global float *dV,
                       __global int *X,
                       __global int *coefg1,
                       __global int *coefg2,
```

```
                        const unsigned int tam0,
                        const unsigned int tam1,
                        const unsigned int tam2,
                        const unsigned int lim0,
                        const unsigned int lim1,
                        const unsigned int lim2,
                        const unsigned int lim3)



{
int k = get_global_id(0);
int l = get_global_id(1);
int n = get_global_id(2);

float h1=0.0;
float h2=0.0;
for(int i=coefg1[l+0]; i<=coefg1[l+1*lim2]; i++)
{
for(int j=coefg2[k+0]; j<=coefg2[k+1*lim3]; j++)
{
h1+= dV[j-coefg2[k+0]+(i-coefg1[l+0])*lim0+n*lim0*lim1+l*lim0*lim1*tam2+k*lim0*
    lim1*tam2*lim2+0]*X[j-1+(i-1)*tam0];
h2+= dV[j-coefg2[k+0]+(i-coefg1[l+0])*lim0+n*lim0*lim1+l*lim0*lim1*tam2+k*lim0*
    lim1*tam2*lim2+1*lim0*lim1*tam2*lim2*lim3]*X[j-1+(i-1)*tam0];

}
}
H2[k+l*lim3+n*lim3*lim2+0]=h1;
H2[k+l*lim3+n*lim3*lim2+1*lim3*lim2*tam2]=h2;
}

__kernel void proyeccion  (__global float *proyH,
                           __global float *T,
                           __global float *Dif,
                           __global float *H2,
                           __global float *media,
                           __global float *Wn)
{
int lim3 = get_global_size(0);
int lim2 = get_global_size(1);
int tam2 = get_global_size(2);

int i=get_global_id(0);
int j=get_global_id(1);
int k=get_global_id(2);

float suma0=0.0;
float suma1=0.0;
media[i+j*lim3+0]=0.0;
media[i+j*lim3+1*lim3*lim2]=0.0;

for(int k1=0; k1<tam2; k1++)
{
        suma0+= H2[i+j*lim3+k1*lim3*lim2+0];
        suma1+= H2[i+j*lim3+k1*lim3*lim2+1*lim3*lim2*tam2];
}
media[i+j*lim3+0]=suma0/tam2;
media[i+j*lim3+1*lim3*lim2]=suma1/tam2;

proyH[i+j*lim3+k*lim3*lim2+0]=H2[i+j*lim3+k*lim3*lim2+0]-media[i+j*lim3+0];
proyH[i+j*lim3+k*lim3*lim2+1*lim3*lim2*tam2]=H2[i+j*lim3+k*lim3*lim2+1*lim3*lim2
    *tam2]-media[i+j*lim3+1*lim3*lim2];

Dif[i+j*lim3+k*lim3*lim2+0]=(*Wn)*proyH[i+j*lim3+k*lim3*lim2+0];
```

```
Dif[i+j*lim3+k*lim3*lim2+1*lim3*lim2*tam2]=(*Wn)*proyH[i+j*lim3+k*lim3*lim2+1*
    lim3*lim2*tam2];
//actualizo T
T[i+j*lim2+k*lim2*lim3+0]=T[i+j*lim2+k*lim2*lim3+0]-Dif[i+j*lim2+k*lim2*lim2+0];
T[i+j*lim2+k*lim2*lim3+1*lim2*lim3*tam2]=T[i+j*lim2+k*lim2*lim3+1*lim2*lim3*tam2
    ]-Dif[i+j*lim3+k*lim3*lim2+1*lim3*lim2*tam2];
}


__kernel void transformar  (__global float *TAux,
                            __global float *T,
                            __global int *r1,
                            __global int *r2,
                            __global int *coef1,
                            __global int *coef2,
                            const unsigned int lim0,
                            const unsigned int lim1,
                            const unsigned int lim2,
                            const unsigned int lim3,
                            const unsigned int tsE,
                            const unsigned int nt,
                            const unsigned int longr1,
                            const unsigned int longr2)



{
int iaux = get_global_id(0);
float x=(float)iaux;
int i=iaux%longr2;

int j=iaux/longr2;
int n = get_global_id(1);
int kaux = get_global_id(2);
int l=kaux%tsE;
int k=kaux/tsE;

int tam2 = get_global_size(1);

int kaux2 = coef2[r2[i]-1+0]-1;
int laux2 = coef1[r1[j]-1+0]-1;

TAux[i+j*longr1+n*longr1*longr2+0+l*longr1*longr2*tam2*nt+k*longr1*longr2*tam2*
    nt*tsE] = T[k+coef2[r2[i]-1+0]-1+(l+coef1[r1[j]-1+0]-1)*lim2+n*lim2*lim3+0];
TAux[i+j*longr1+n*longr1*longr2+1*longr1*longr2*tam2+l*longr1*longr2*tam2*nt+k*
    longr1*longr2*tam2*nt*tsE] =T[k+coef2[r2[i]-1+0]-1+(l+coef1[r1[j]-1+0]-1)*
    lim2+n*lim2*lim3+1*lim2*lim3*tam2];

}


__kernel void transformar2  (__global float *xn1,
                             __global float *xn2,
                             __global float *TAux,
                             __global int *r1,
                             __global int *r2,
                             __global float *BB,
                             const unsigned int tsE,
                             const unsigned int nt,
                             const unsigned int tam0,
                             const unsigned int tam1)
{
int longr2 = get_global_size(0);
int longr1 = get_global_size(1);
int tam2 = get_global_size(2);
```

```
int i=get_global_id(0);
int j=get_global_id(1);
int n=get_global_id(2);

for(int l = 0; l < tsE; l++)
{
for(int k = 0; k < tsE; k++)
{
xn1[i+r1[0]-1+(j+r2[0]-1)*tam0+n*tam0*tam1] +=TAux[i+j*longr1+n*longr1*longr2+0+
    l*longr1*longr2*tam2*nt+k*longr1*longr2*tam2*nt*tsE]* BB[i+r1[0]-1+(j+r2
    [0]-1)*tam0+l*tam0*tam1+k*tam0*tam1*tsE];

xn2[i+r1[0]-1+(j+r2[0]-1)*tam0+n*tam0*tam1] += TAux[i+j*longr1+n*longr1*longr2
    +1*longr1*longr2*tam2+l*longr1*longr2*tam2*nt+ k*longr1*longr2*tam2*nt*tsE]
    * BB[i+r1[0]-1+(j+r2[0]-1)*tam0+l*tam0*tam1+k*tam0*tam1*tsE];

}
}
}


__kernel void interpolar  ( __global float *IT,
                            __global float *xn1,
                            __global float *xn2,
                            __global float *I,
                            __global int *caja,
                            const unsigned int margen1,
                            const unsigned int margen2,
                            const unsigned int margen3,
                            const unsigned int tam0,
                            const unsigned int tam1)
{
int i=get_global_id(0);
int j=get_global_id(1);
int n=get_global_id(2);

float alpha=0;
float beta=0;

float w00=0;
float w01=0;
float w10=0;
float w11=0;


int xn1_f= floor (xn1[i + (caja[1+0]-margen1-1)+(j + (caja[0+0]-margen2-1))*tam0
    +n*tam0*tam1])-1;
int xn2_f=floor (xn2[i + (caja[1+0]-margen1-1)+(j + (caja[0+0]-margen2)-1)*tam0+
    n*tam0*tam1])-1;

int xn1_c=1+xn1_f;
int xn2_c=1+xn2_f;

alpha = xn1[i + (caja[1+0]-margen1-1)+(j + (caja[0+0]-margen2-1))*tam0+n*tam0*
    tam1]-1 - xn1_f;
beta  = xn2[i + (caja[1+0]-margen1-1)+(j + (caja[0+0]-margen2-1))*tam0+n*tam0*
    tam1]-1 - xn2_f;

//definimos los pesos, que es el desplazamiento producido en cada punto
w00 = (1-alpha)*(1-beta);
w10 = (1-alpha)*(beta);
w01 = (alpha)*(1-beta);
w11 = (alpha)*(beta);
```

```
IT[i + (caja[1+0]-margen1-1)+(j + (caja[0+0]-margen2-1))*tam0+n*tam0*tam1] = w00
     * I[(int)(xn2_f)+((int)(xn1_f))*tam0+n*tam0*tam1] + w01 * I[(int)(xn2_f)+((
    int)(xn1_c))*tam0+n*tam0*tam1] + w10 * I[(int)(xn2_c)+((int)(xn1_f))*tam0+n*
    tam0*tam1] + w11 * I[(int)(xn2_c)+((int)(xn1_c))*tam0+n*tam0*tam1];


}
__kernel void metrica  (__global float *V,
                        __global float *dy,
                        __global float *IT,
                        __global float *media,
                        const unsigned int tam2)
{
int tam0 = get_global_size(0);
int tam1 = get_global_size(1);

int i=get_global_id(0);
int j=get_global_id(1);

V[i+j*tam0]=0;
media[i+j*tam0]=0;
float suma=0.0;
for(int k=0; k<tam2; k++)
{
suma+= IT[i+j*tam0+k*tam0*tam1];
}
media[i+j*tam0]=suma/tam2;
for(int k=0; k<tam2; k++)
{
dy[i+j*tam0+k*tam0*tam1]=(2.0/tam2)*(IT[i+j*tam0+k*tam0*tam1]-media[i+j*tam0]);
V[i+j*tam0] = V[i+j*tam0]  + (IT[i+j*tam0+k*tam0*tam1]-media[i+j*tam0])*(IT[i+j*
    tam0+k*tam0*tam1]-media[i+j*tam0])/tam2;
}

}
```

En cuanto a los ficheros de cabecera:

- **Cabecera con el prototipo de las funciones: *createbspline.h***:

```
#include "baseDatos.h"

//struct Bspline creabspline(int tam[],int caja[2][2],int Dp[],int E,int N);


Bspline_t* creabspline(int tam[],int *caja,int Dp[],int E,int N);

void optimizador(int tam[], Bspline_t* ts, float *IC,float **IT, Datos_t datos, int
    **X, Parametros_t parametros, float Wn, int *caja, int flagW);

//void metrica (float***V, float*** IT, Datos_t datos, Term_t* term, int caja[2][2],
     int tam[], float**media,float**** dy);
float* metrica (float *V, float* IT, Datos_t datos, int *caja, int tam[], float*
    media, float** dy,Bspline_t* ts);

void transformar (float *xn1Aux, float *xn2Aux, float **xn1, float **xn2, float **
    TAux, Bspline_t* ts, float *T, int *caja, int flagopt, int tam[], int longr1,
    int longr2, int r1[], int r2[], int lim[]);

void interpolar (float **IT, float **xn1, float **xn2, float *I, int *caja,unsigned
    int margen1,unsigned int margen2,unsigned int margen3, int tam[]);
```

```
void gradiente (float**dV, float* dx1, float* dx2, float* dy,  float* IT, Bspline_t*
     ts, Datos_t datos, float *x1, float *x2, float *T, int tam[], int lim[]);


void gradiente_imagen (float** dx1, float** dx2, float* IT,int tam[]);


void coste(float *H1, float **H2, float *V,int *X, int flag_coste,int tam[],
    Bspline_t* ts,int lim[], float *dV);

void proyeccion(float *H2, float **proyH,int tam[], int lim[],float *media, float Wn
    , float **T, float **Dif);

void evolucion (float *TDif, float *HDif, float *Wn,  float *T, float H[], int iter,
     int flagW,int lim[], int tam[]);
```

- **Base de datos: *baseDatos.h*:**

```
#include "baseDatos.h"

//struct Bspline creabspline(int tam[],int caja[2][2],int Dp[],int E,int N);


Bspline_t* creabspline(int tam[],int *caja,int Dp[],int E,int N);

void optimizador(int tam[], Bspline_t* ts, float *IC,float **IT, Datos_t datos, int
    **X, Parametros_t parametros, float Wn, int *caja, int flagW);

//void metrica (float***V, float*** IT, Datos_t datos, Term_t* term, int caja[2][2],
     int tam[], float**media,float**** dy);
float* metrica (float *V, float* IT, Datos_t datos, int *caja, int tam[], float*
    media, float** dy,Bspline_t* ts);

void transformar (float *xn1Aux, float *xn2Aux, float **xn1, float **xn2, float **
    TAux, Bspline_t* ts, float *T, int *caja, int flagopt, int tam[], int longr1,
    int longr2, int r1[], int r2[], int lim[]);

void interpolar (float **IT, float **xn1, float **xn2, float *I, int *caja,unsigned
    int margen1,unsigned int margen2,unsigned int margen3, int tam[]);

void gradiente (float**dV, float* dx1, float* dx2, float* dy,  float* IT, Bspline_t*
     ts, Datos_t datos, float *x1, float *x2, float *T, int tam[], int lim[]);


void gradiente_imagen (float** dx1, float** dx2, float* IT,int tam[]);


void coste(float *H1, float **H2, float *V,int *X, int flag_coste,int tam[],
    Bspline_t* ts,int lim[], float *dV);

void proyeccion(float *H2, float **proyH,int tam[], int lim[],float *media, float Wn
    , float **T, float **Dif);

void evolucion (float *TDif, float *HDif, float *Wn,  float *T, float H[], int iter,
     int flagW,int lim[], int tam[]);
```

- **Cabecera de la máscara: *mascara.h*:**

```
void mascara (int **X,int radio,int tam[], int *areaX, int *caja);
```

- **Cabecera de las B-spline: *spline.h*:**

```
#include<cmath>
#include <stdlib.h>

float bspline1(float x);
float bspline2(float x);
float bspline3(float x);
float bspline0(float x);

float b1spline1(float x);
float b1spline2(float x);
float b1spline3(float x);


float bsplineN(float x,int n);
float b1splineN(float x,int n);
```

Referencia

Bibliografía

[1] Joep Perk, Guy De Backer, Helmut Gohlke, Ian Graham, Željko Reiner, Monique Verschuren, Christian Albus, Pascale Benlian, Gudrun Boysen, Renata Cifkova, et al. European guidelines on cardiovascular disease prevention in clinical practice (version 2012). *European heart journal*, 33(13): 1635–1701, 2012.

[2] Helen B Hubert, Manning Feinleib, Patricia M McNamara, and William P Castelli. Obesity as an independent risk factor for cardiovascular disease: a 26-year follow-up of participants in the framingham heart study. *Circulation*, 67(5):968–977, 1983.

[3] S Sans, H Kesteloot, and D obot Kromhout. The burden of cardiovascular diseases mortality in europe. *European heart journal*, 18(8):1231–1248, 1997.

[4] Hui Wang and Amir A Amini. Cardiac motion and deformation recovery from mri: a review. *Medical Imaging, IEEE Transactions on*, 31(2):487–503, 2012.

[5] Santiago Rodrigo Sanz Estébanez. A groupwise ffd registration method to compensate heart motion. 2014.

[6] Lucilio Cordero-Grande, Susana Merino-Caviedes, Santiago Aja-Fernández, and Carlos Alberola-Lopez. Groupwise elastic registration by a new sparsity-promoting metric: application to the alignment of cardiac magnetic resonance perfusion images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(11):2638–2650, 2013.

[7] CT Metz, Stefan Klein, Michiel Schaap, Theo van Walsum, and Wiro J Niessen. Nonrigid registration of dynamic medical imaging data using nd+ t b-splines and a groupwise optimization approach. *Medical image analysis*, 15(2):238–249, 2011.

[8] Mathieu De Craene, Gemma Piella, Oscar Camara, Nicolas Duchateau, Etelvino Silva, Adelina Doltra, Jan D'hooge, Josep Brugada, Marta Sitges, and Alejandro F Frangi. Temporal diffeomorphic free-form deformation: Application to motion and strain estimation from 3d echocardiography. *Medical Image Analysis*, 16(2):427–450, 2012.

[9] Richard Lorne Ehman, MT McNamara, M Pallack, H Hricak, and CB Higgins. Magnetic resonance imaging with respiratory gating: techniques and advantages. *American journal of Roentgenology*, 143(6):1175–1182, 1984.

[10] Javier Royuela-del Val, Lucilio Cordero-Grande, Federico Simmross-Wattenberg, Marcos Martín-Fernández, and Carlos Alberola-López. Nonrigid groupwise registration for motion estimation and compensation in compressed sensing reconstruction of breath-hold cardiac cine mri. *Magnetic resonance in medicine*, 2015.

[11] Marc Modat, Gerard R Ridgway, Zeike A Taylor, Manja Lehmann, Josephine Barnes, David J Hawkes, Nick C Fox, and Sébastien Ourselin. Fast free-form deformation using graphics processing units. *Computer methods and programs in biomedicine*, 98(3):278–284, 2010.

[12] MATLAB. *version 8.5.0 (R2015a)*. The MathWorks Inc., 2015.

[13] Eclipse. *version 4.5.1 (2015)*. 2015.

[14] Wikipedia. Endocardio — wikipedia, la enciclopedia libre, 2015. URL `https://es.wikipedia.org/w/index.php?title=Endocardio&oldid=84017809`. [Internet; descargado 24-mayo-2016].

[15] Wikipedia. Epicardio — wikipedia, la enciclopedia libre, 2013. URL `https://es.wikipedia.org/w/index.php?title=Epicardio&oldid=66140786`. [Internet; descargado 24-mayo-2016].

[16] William R Crum, Thomas Hartkens, and DLG Hill. Non-rigid image registration: theory and practice. *The British Journal of Radiology*, 2014.

[17] Daniel Rueckert, Luke I Sonoda, Carmel Hayes, Derek LG Hill, Martin O Leach, and David J Hawkes. Nonrigid registration using free-form deformations: application to breast mr images. *Medical Imaging, IEEE Transactions on*, 18(8):712–721, 1999.

[18] Miguel Ángel Martín Fernández et al. Una contribución al registrado articulado: aplicación a la determinación de la maduración ósea mediante análisis de imágenes radiográficas. 2012.

[19] Sameh Hamrouni, Nicolas Rougon, and Françoise Prêteux. Multi-feature information-theoretic image registration: application to groupwise registration of perfusion mri exams. In *Biomedical Imaging: From Nano to Macro, 2011 IEEE International Symposium on*, pages 574–577. IEEE, 2011.

[20] Wikipedia. Ejecución fuera de orden — wikipedia, la enciclopedia libre, 2013. URL `https://es.wikipedia.org/w/index.php?title=Ejecuci%C3%B3n_fuera_de_orden&oldid=65278407`. [Internet; descargado 9-junio-2016].

[21] Federico Simmross Wattenberg. Introducción a la programación heterogénea cpu-gpu en opencl. 2016.

[22] Khronos Group. Intro opencl tutorial. October 2012.