

Índice de contenidos

<i>Índice de contenidos</i>	i
<i>Índice de figuras</i>	v
<i>Capítulo 1 Introducción</i>	1
1.1 Motivación	3
1.2 Objetivos	5
1.3 Fases y métodos	6
1.4 Medios	7
1.5 Estructura del documento	7
<i>Capítulo 2 DT-MRI</i>	9
2.1 Introducción	11
2.2 Resonancia magnética por difusión	11
2.3 Estimación del tensor de difusión	12
2.3.1 Cálculo del tensor de difusión	14
2.3.2 Anisotropía y medidas macroestructurales	15
2.3.3 Interpolación	18
2.4 Visualización	20
2.5 Tractografía	23
<i>Capítulo 3 Imagen por tensor de esfuerzo</i>	25
3.1 Introducción	27
3.2 Técnicas para la detección del movimiento en MRI	28
3.2.1 TMRI	29
3.2.2 PCMRI	29
3.2.3 Métodos de campo de gradiente pulsado	30
3.3 Estimación del tensor de esfuerzo	30
3.3.1 Interpolación	31
3.4 Visualización	32
<i>Capítulo 4 Librerías empleadas: ITK, VTK, FLTK</i>	35
4.1 ITK	37
4.1.1 Introducción	37
4.1.2 Características generales	37
4.1.3 Programación genérica	38
4.1.4 Gestión de la memoria	39
4.1.5 Representación de datos	39
4.1.6 El pipeline de datos	40
4.1.7 Filtrado	40
4.1.8 Lectura y escritura en ficheros	40

4.2 VTK	41
4.2.1 Introducción	41
4.2.2 Características generales	41
4.2.3 El modelo de gráficos	42
4.2.4 El pipeline de visualización	42
4.2.5 Gestión de memoria	43
4.2.6 Control implícito de la ejecución	43
4.3 FLTK	44
4.3.1 Introducción	44
4.3.2 Características generales	44
4.3.3 Tratamiento de los eventos	45
4.3.4 FLUID	45
4.4 CMake	46
 <i>Capítulo 5 Saturn y otras interfaces</i>	 47
5.1 Introducción a Saturn	49
5.2 Interfaz de usuario	50
5.3 Código de Saturn	54
5.3.1 Clase UsimagToolBase	54
5.3.2 Clase UsimagToolGUI	55
5.3.3 Clase UsimagToolConsole	55
5.3.4 Clase TensorGUI	56
5.3.5 Clase TensorConsole	56
5.3.6 Clase DTITensor	57
5.3.7 Clase DataTensorElementType	57
5.3.8 Clase VolumesContainer	58
5.3.9 Clase Viewer3D	58
5.3.10 Resto de código	58
5.4 Otras interfaces de visualización	59
 <i>Capítulo 6 Desarrollo de una interfaz de visualización para DTI</i>	 63
6.1 Introducción	65
6.2 Clase vtkTensorGlyphDTI	65
6.2.1 Tipos de dato	67
6.2.2 Variables de clase	68
6.2.3 Métodos	70
6.2.4 Cumplimiento de los objetivos	74
6.2.5 La clase vtkTensorGlyphDTI frente a vtkTensorGlyph	74
6.2.6 Uso de la clase	76
6.3 Código en TensorConsole	76
6.4 Interfaz de usuario	79
6.4.1 Forma de uso	81
6.5 Pruebas	82
6.6 Ejemplos	84
 <i>Capítulo 7 Desarrollo de una interfaz para tensor de esfuerzo</i>	 95

7.1 Introducción	97
7.2 Clase StrainTensor	97
7.3 Tipos de dato	98
7.4 Clase ComputeStrainScalars	99
7.5 Lectura y escritura de ficheros	100
7.6 Clase vtkTensorGlyphStrain	100
7.6.1 Tipos de dato	101
7.6.2 Variables de clase	101
7.6.3 Método GetOutput()	102
7.6.4 Uso de la clase	103
7.7 Métodos en TensorConsole	104
7.8 Interfaz	105
7.8.1 Uso de la interfaz de usuario	106
Referencias	107

Índice de figuras

Capítulo 2	
<i>Figura 2.1. Diferentes geometrías de glifo</i>	22
<i>Figura 2.2. Glifos supercuádricos en función del parámetro γ [16]</i>	22
Capítulo 3	
<i>Figura 3.1. Representación de glifos 2D [29]</i>	33
Capítulo 5	
<i>Figura 5.1. Interfaz de usuario de Saturn</i>	51
<i>Figura 5.2. Paneles de configuración de Saturn</i>	52
<i>Figura 5.3. Visualización 3D en Saturn</i>	53
<i>Figura 5.4. Tractografía en Saturn</i>	53
<i>Figura 5.5. Visualización con Fluid de la interfaz definida en UsimagToolGUI</i>	55
<i>Figura 5.6. Visualización de glifos en 3D Slicer [52]</i>	60
<i>Figura 5.7. Visualización de tractos mediante glifos en 3D Slicer [52]</i>	60
<i>Figura 5.8. Interfaz de visualización de glifos en MedINRIA [54]</i>	62
Capítulo 6	
<i>Figura 6.1. Justificación del factor signo</i>	73
<i>Figura 6.2. Interfaz de usuario para glifos en DTI</i>	79
<i>Figura 6.3. Aspecto al variar los índices de recorte de planos</i>	80
<i>Figura 6.4. Comparación del tiempo de carga en función del número de glifos</i>	83
<i>Figura 6.5. Comparación de la latencia en función de la geometría del glifo</i>	83
<i>Figura 6.6. Latencia en función de la resolución de los glifos</i>	83
<i>Figura 6.7. Visualización de glifos en el plano axial</i>	85
<i>Figura 6.8. Vista con la opacidad reducida hasta cero</i>	85
<i>Figura 6.9. Visualización cercana de los glifos</i>	86
<i>Figura 6.10. Glifos en el plano sagital (X)</i>	86
<i>Figura 6.11. Glifos en el plano coronal (Y)</i>	87
<i>Figura 6.12. Glifos en tres planos</i>	87
<i>Figura 6.13. Distintos tipos de coloreado</i>	88
<i>Figura 6.14. Diferentes geometrías de glifo</i>	89
<i>Figura 6.15. Efecto del factor de escala</i>	90
<i>Figura 6.16. Aplicación del factor de escala a la visualización con cuboides:</i>	91
<i>Figura 6.17. Crop de planos (recorte de planos)</i>	92
<i>Figura 6.18. Discriminación de glifos (FA > 20)</i>	92
<i>Figura 6.19. Efecto del parámetro gamma</i>	93
<i>Figura 6.20. Glifos en tractografía</i>	94
Capítulo 7	
<i>Figura 7.1. Aspecto de la interfaz para tensor de esfuerzo</i>	106

Capítulo 1

Introducción

Este capítulo contiene una introducción al proyecto. Se explica el trasfondo y la motivación del mismo, los objetivos que persigue, las fases de desarrollo y la estructura de este documento.

1.1 Motivación

El campo de la imagen médica ha adquirido una gran importancia en los últimos años. Este conjunto de técnicas proporciona información sobre el interior del cuerpo humano que resulta vital en el diagnóstico, análisis y seguimiento de enfermedades. El avance de la tecnología y el desarrollo de los ordenadores la han convertido en un elemento fundamental de la medicina actual, pese a estar en pleno crecimiento. Una de las áreas en desarrollo es la de la imagen tensorial, que ofrece información que no puede obtenerse con otras modalidades y que puede resultar muy relevante en la práctica clínica.

Una técnica para la obtención de imágenes tensoriales es la resonancia magnética por tensor de difusión, DT-MRI [1]. Esta técnica se basa en el fenómeno de la difusión, el movimiento de la materia debida a movimientos moleculares aleatorios. Esta técnica permite estudiar la difusividad de las moléculas de agua en los tejidos, en particular en la sustancia blanca cerebral. De este modo se puede determinar la estructura interna de la misma. Esta es una ventaja importante respecto de otras técnicas de resonancia magnética, que perciben la sustancia blanca como una masa homogénea.

DT-MRI permite por primera vez la exploración no invasiva de la anatomía estructural del cerebro *in vivo*, y su aplicación clínica incluye el estudio de enfermedades como esclerosis múltiple [11][12], leucoaraiosis, isquemia cerebral, esquizofrenia [13] y epilepsia, y resulta útil en neurooncología y neurocirugía guiada por imagen [14].

La leucoaraiosis, por ejemplo, es un término que hace referencia a los cambios en la difusión en la sustancia blanca, que pueden aparecer en la isquemia crónica y la enfermedad de Alzheimer, entre otros. En la esclerosis múltiple se produce una desmielinización de las fibras que altera las características de la difusión en la sustancia blanca. DT-MRI permite detectar estas y otras alteraciones.

Por otro lado, la adquisición de imágenes tensoriales tiene una relevancia cada vez mayor también en otras áreas, como la cardiología. El estudio de las propiedades mecánicas del corazón proporciona un importante parámetro en diagnóstico y seguimiento de los pacientes, y es posible gracias a la modalidad de imagen por tensor de esfuerzo [30]. En particular, la adquisición de secuencias temporales del corazón a lo largo del ciclo cardíaco permite realizar un seguimiento de la contracción de las paredes del endocardio a través de técnicas de procesado de imagen [22].

Algunas de las técnicas empleadas son MRI (imagen por resonancia magnética), MRI con etiquetado o MRI por contraste de fase. La resonancia

magnética ofrece una gran libertad en la posición y orientación de las imágenes, y las técnicas específicas permiten seguir el movimiento de los tejidos o dar datos cuantitativos del movimiento. La adquisición y el procesado de estas imágenes tienen una importancia creciente dado su poder diagnóstico a través de medidas derivadas del tensor de esfuerzo [28][29][30].

Además, la posibilidad de representar el campo tensorial sobre la imagen cardiaca capacita al profesional para realizar un mejor diagnóstico y seguimiento de los pacientes. Sin embargo, la información tensorial a representar es compleja, y no hay acuerdo sobre la mejor forma de representación de campos tensoriales, por lo que supone un campo de investigación activo en estos momentos [30][32].

Entre las aplicaciones clínicas de esta técnica aparecen enfermedades cardíacas isquémicas como el infarto de miocardio, enfermedades de las válvulas cardíacas como la estenosis, cardiomiopatías como la cardiomiopatía hipertrófica o el estudio de la función diastólica del corazón. Cada una de estas enfermedades se caracteriza por una cierta alteración en el movimiento cardíaco, que puede ser detectada con las técnicas de imagen por tensor de esfuerzo.

Otra interesante aplicación del tensor de esfuerzo aparece en aplicaciones de elastografía. La elastografía consiste en la medición de las propiedades mecánicas de un tejido. En el caso del cáncer de próstata y de mama se ha demostrado que los tejidos cambian sus propiedades elásticas. Estos cambios pueden ser medidos por medio de señales de ultrasonido. La forma estándar de visualizar las propiedades elásticas es a través del esfuerzo axial [33][34]. No obstante la visualización del campo tensorial puede facilitar enormemente el análisis visual del profesional en análisis elastográficos.

La complejidad de las imágenes tensoriales hace que sean necesarias aplicaciones informáticas para su procesado y visualización. Existen actualmente una variedad de estas herramientas, algunas de ellas aún en desarrollo. Una de ellas es 3D Slicer [52], un software de código abierto orientado a la visualización diagnóstica y la cirugía guiada por imagen, y que cuenta con un módulo dedicado a DT-MRI. Otro caso es MediINRIA [54], una herramienta gratuita (aunque no abierta) que incluye dos módulos específicos para DT-MRI, DTI-Track y Tensor Viewer. Por el contrario, es complicado encontrar aplicaciones para visualizar imágenes de tensor de esfuerzo cardíaco.

El Laboratorio de Procesado de Imagen (LPI) de la Universidad de Valladolid está desarrollando la herramienta Saturn (también conocida como UsimagTool) para el tratamiento de imágenes tensoriales con fines médicos [50]. Esta aplicación es código abierto y cuenta, entre otras cosas, con diversas opciones de procesado de imagen y con la posibilidad de realizar tractografía.

Saturn permite visualizar magnitudes escalares extraídas de las imágenes tensoriales DT-MRI, como las medidas de anisotropía o los autovalores. El objetivo de este proyecto es desarrollar una interfaz para la visualización de los tensores de difusión mediante representaciones tridimensionales o glifos, describen visualmente las características de la difusión en cada punto. Asimismo, también se pretende desarrollar una interfaz para visualizar imágenes de tensor de esfuerzo, una posibilidad que en estos momentos no existe en Saturn.

1.2 Objetivos

En vista de las motivaciones expresadas en el apartado anterior, se establecen para este proyecto los siguientes objetivos:

- Estudio de la estimación de campos tensoriales en diversas técnicas de imagen médica. Resulta de especial interés conocer los procesos de adquisición y estimación tensorial en DT-MRI y tensor de esfuerzo, así como sus fundamentos teóricos.
- Familiarización con las librerías ITK, VTK y FLTK. ITK es empleada en Saturn para la representación de la información y el procesado de imágenes. VTK se emplea en visualización, por lo que es fundamental en este proyecto. FLTK y la aplicación FLUID se usan para la creación de interfaces gráficas de usuario.
- Familiarización con la herramienta Saturn, tanto a nivel de usuario como de desarrollador.
- Estudio de visualización de campos tensoriales existentes.
- Implementación de módulos integrados en Saturn para la visualización de campos tensoriales, tanto en DT-MRI como para tensor de esfuerzo. Estos módulos deben seguir el modelo del resto de la aplicación e integrarse en la interfaz de la misma.

1.3 Fases y métodos

Para el adecuado desarrollo del proyecto y la consecución de los objetivos anteriores, se establecen las siguientes fases de trabajo:

- Estudio y comprensión de las distintas modalidades médicas de estimación tensorial. Estas serán fundamentales en el desarrollo de métodos de visualización. Se realizará a través de libros y artículos disponibles en el Laboratorio de Procesado de Imagen y a través de la biblioteca virtual de la universidad.
- Familiarización con VTK, ITK y FLTK. Inicialmente con un propósito general, a modo de primer acercamiento, y posteriormente orientado a la visualización de campos tensoriales. Para ello se recurrirá a la literatura disponible, así como al mismo código fuente de las librerías. También se probarán distintos ejemplos prácticos, que afiancen los conocimientos adquiridos.
- Familiarización con la herramienta Saturn. En un primer momento desde la perspectiva de usuario, para conocer su interfaz y sus funcionalidades, y posteriormente a nivel de código fuente. Esto último es fundamental, ya que el desarrollo de este proyecto debe integrarse en Saturn a nivel de código y de interfaz.
- Estudio de las interfaces y modelos de visualización. Esto permitirá estudiar de qué forma se aborda el problema en otros grupos de trabajo. Se trabajará con distintas interfaces de libre distribución que se encuentran disponibles en el laboratorio o a través de Internet.
- Desarrollo de métodos de visualización, a partir de los conceptos extraídos de la literatura y de aportaciones propias de este proyecto.
- Implementación en Saturn de los métodos desarrollados. La integración permitirá aprovechar la interfaz existente de Saturn, y la aplicación de los métodos a casos prácticos.
- Pruebas de funcionamiento de los módulos desarrollados, así como de cada una de las funcionalidades que implementa.

1.4 Medios

Para el desarrollo de este proyecto, el Laboratorio de Procesado de Imagen cuenta con los siguientes medios:

- Ordenador personal.
- Literatura y artículos accesibles a través de la biblioteca virtual de la Universidad de Valladolid.
- Librerías software como ITK, VTK y FLTK, todas de código abierto y disponibles en la red.

Además, se cuenta con los medios ofrecidos por el Laboratorio de Procesado de Imagen (LPI):

- Ordenadores personales y estaciones de trabajo.
- Varios servidores con alta capacidad de cálculo.
- Aplicación Saturn: código fuente, documentación y ejecutables.
- Banco de imágenes de DT-MRI.
- Banco de imágenes MRI cardiaca.
- Banco de imágenes de elastografía.

1.5 Estructura del documento

La estructura de este documento sigue las fases indicadas anteriormente. El capítulo 2 contiene un repaso al estado del arte sobre DT-MRI. En él, se explican los fundamentos teóricos de la difusión, los principios de la técnica y el método de estimación del tensor, así como la metodología habitual para visualizar la imagen.

De forma similar, en el capítulo 3 se explica lo concerniente a la imagen por tensor de esfuerzo. Entre otras cosas, se habla de las diferentes técnicas de

adquisición de la imagen, de la estimación del tensor y de los diferentes modos de visualización.

En el cuarto capítulo se introducen las librerías empleadas en Saturn, y por extensión utilizadas en este proyecto. ITK es empleada por Saturn para la representación de datos y el procesado de imágenes. VTK es utilizada para visualización, y FLTK para el diseño de interfaces gráficas. Por último, aparece también la herramienta CMake, utilizada por estas librerías y por Saturn para garantizar la compatibilidad con las distintas plataformas.

El capítulo 5 explica el origen, funcionalidades y estructura interna de la herramienta Saturn. Este capítulo pretende introducir las características de la aplicación, de las que se va a hacer uso en el desarrollo de los distintos módulos. Asimismo, también se presentan otras interfaces de visualización de imagen médica, como Slicer y MedINRIA.

El capítulo 6 trata sobre el primero de los módulos desarrollados, correspondiente a DT-MRI. Este módulo permite la visualización del campo tensorial mediante representaciones geométricas o glifos. El capítulo explica las distintas partes que componen el módulo (interfaz gráfica, clases específicas, código adicional). Asimismo, contiene una serie de ejemplos de su uso, un apartado de pruebas de funcionamiento y un manual de usuario.

El capítulo 7 explica el desarrollo de la segunda parte del proyecto, correspondiente a tensor de esfuerzo. Del mismo modo, se explican los elementos que la componen, las nuevas interfaces desarrolladas y las funcionalidades implementadas, así como las pruebas de funcionamiento correspondientes y un manual de uso.

El capítulo 8 contiene las conclusiones del proyecto. Se realiza un repaso global al trabajo realizado, y se concluye con las líneas de trabajo futuras.

Capítulo 2

DT-MRI

Este capítulo contiene una introducción a las técnicas de imagen de resonancia magnética por tensor de difusión, DT-MRI. En este capítulo se explican los fundamentos de la técnica, el método para obtener el tensor de difusión, la forma de visualizarlo, y algunas aplicaciones clínicas de esta modalidad de imagen.

2.1 Introducción

Existen múltiples técnicas para la obtención de imágenes cerebrales. Hasta hace algunas décadas, la única forma de ver el cerebro era la observación directa. La primera técnica de observación del cerebro apareció con los rayos X. La tomografía computerizada [1] utiliza los rayos X para obtener una serie de imágenes, que son combinadas y procesadas por un ordenador para obtener una imagen tridimensional. Una segunda modalidad de imagen es la tomografía por emisión de positrones (PET). Se le administra al paciente una inyección de glucosa radioactiva, que activa diferentes áreas del cerebro. La PET permite medir esta actividad, mostrando qué áreas presentan una mayor actividad al realizar diferentes operaciones mentales.

En tercer lugar destaca la imagen por resonancia magnética [2]. La resonancia magnética es una técnica típicamente no invasiva (aunque no siempre), que utiliza campos magnéticos para estudiar la estructura y composición del cuerpo a analizar. Existen diferentes tipos de resonancia magnética, en función de la secuencia de pulsos que utilizan y del tipo de información que se obtiene: resonancia magnética anatómica, funcional, de difusión, de perfusión, espectroscópica, etc. Este trabajo se centra en la resonancia magnética por difusión.

2.2 Resonancia magnética por difusión

La difusión es el proceso por el cual la materia es transportada de un lugar a otro de un sistema gracias a movimientos moleculares aleatorios, en un proceso análogo al de la transferencia de calor por conducción. El médico alemán Adolf Fick (1855) describió por primera vez este fenómeno [3], afirmando que las diferencias en la concentración local de un soluto elevarán el flujo neto de moléculas del soluto de las zonas con una concentración alta a aquellas con una concentración baja, en lo que se conoce como movimiento browniano (en honor al botanista Robert Brown).

En imagen médica, la imagen de resonancia magnética convencional permite identificar fácilmente los centros funcionales del cerebro (córtex y núcleo). Sin embargo, con esta técnica la sustancia blanca del cerebro aparece homogénea, sin dar ninguna señal de una disposición compleja de los tractos de fibras.

En la sustancia blanca, la movilidad del agua está restringida por los axones que están orientados según los tractos de fibras. Esta difusión anisotrópica se debe a las múltiples membranas de mielina comprimidas que

comprenden el axón. Pese a que la mielinización no es esencial para la anisotropía de la difusión en los nervios, la mielina suele considerarse como la mayor barrera a la difusión en tractos de fibras mielinizados.

Una primera técnica para estudiar la difusión del agua en tejidos es la imagen potenciada en difusión (Diffusion Weighted Imaging, o DWI) [4], con la que se obtiene un solo coeficiente de difusión aparente (o ADC) para cada voxel. Esta medida basta para identificar las características de difusión en tejidos donde la difusividad medida es en mayor medida independiente de la orientación.

Pero en tejidos anisotrópicos esto no es suficiente. Los medios anisotrópicos son aquellos que tienen diferentes propiedades físicas (en este caso, de difusión) dependiendo de la dirección. Un ejemplo de medio anisotrópico puede ser una fibra textil. Este es el caso del músculo esquelético y cardiaco, y la sustancia blanca cerebral, donde se sabe que la difusividad medida depende de la orientación del tejido. En estos tejidos, un solo coeficiente no puede caracterizar la movilidad del agua, dependiente de la orientación. El modelo de difusión que puede describir la difusión anisotrópica sustituye el coeficiente escalar de difusión por un tensor simétrico de difusión efectiva o aparente del agua, \mathbf{D} . Esta técnica recibe el nombre de Imagen de Resonancia Magnética por Tensor de Difusión (Diffusion Tensor - Magnetic Resonance Imaging, o DT-MRI) [5].

Por ello, la obtención de la difusión anisotrópica en el cerebro ha allanado el camino para la exploración no invasiva de la anatomía estructural *in vivo*. La resonancia magnética por tensor de difusión efectivo del agua en tejidos puede proporcionar información biológica y clínicamente relevante que no ofrecen otras modalidades de imagen. Esta información incluye parámetros que pueden ayudar a caracterizar la composición del tejido, las propiedades físicas de sus constituyentes, la microestructura de los tejidos y su arquitectura. Por otra parte, esta información se obtiene de forma no invasiva, ya que no requiere el uso de agentes de contraste exógenos.

2.3 Estimación del tensor de difusión

Torrey [6] fue el primero en incorporar la difusión anisotrópica translacional en las ecuaciones de transporte de magnetización de Bloch, que describen la interacción del vector de magnetización de un material en presencia de un campo magnético externo constante. Siguieron soluciones analíticas de esta ecuación para especies de libre difusión y para difusión en geometrías restringidas. Unos diez años después de esta introducción, Stejskal

y Tanner [7] resolvieron la ecuación de Bloch-Torrey para el caso de difusión anisotrópica libre en el principal marco de referencia. Sin embargo, la fórmula de Stejskal-Tanner no puede ser usada habitualmente para medir un tensor de difusión usando resonancia magnética nuclear (NMR) o imagen por resonancia magnética (MRI) por diferentes razones. Primero, esta fórmula relaciona un tensor de difusión dependiente del tiempo con la señal NMR, así que se debe establecer una relación entre el tensor de difusión dependiente del tiempo y un tensor de difusión efectivo. Segundo, antes de la aparición de la MRI, en que fue derivada la ecuación de Stejskal-Tanner, se asumía que una muestra anisotrópica homogénea podía ser reorientada físicamente con el imán para alinear sus ejes principales con el sistema de coordenadas del laboratorio. Después del desarrollo de la MRI, sin embargo, esta suposición ya no se sostiene. Los materiales a estudio son a menudo medios heterogéneos cuyas 'fibras' o ejes principales generalmente no son conocidos a priori y pueden variar de una parte a otra de la muestra.

El tensor de difusión efectivo, \mathbf{D} (o funciones de él), es estimado a partir de una serie de imágenes potenciadas en difusión (DWI), usando una relación entre la medida de la atenuación del eco en cada voxel, y la secuencia de gradientes de campo magnético aplicada. Al igual que en la imagen de difusión (DI), donde se calcula un factor escalar b para cada DWI, en DT-MRI se calcula una matriz simétrica \mathbf{b} para cada DWI. Mientras que el escalar b resume el factor atenuante de la señal MR de todos los gradientes de imagen y de difusión en una dirección, la matriz \mathbf{b} resume el efecto atenuante de todos los gradientes de forma de onda aplicados en las tres direcciones x, y, z .

En DI se usa un conjunto de DWIs y sus correspondientes factores b escalares para estimar un ADC (coeficiente de difusión aparente) a lo largo de una dirección particular usando regresión lineal. En DT-MRI, primero se define un tensor de difusión efectivo (por analogía con la definición de un coeficiente de difusión aparente), a partir del cual se puede derivar una fórmula que relate el tensor de difusión efectivo con la medida del eco. En DT-MRI, se calcula una matriz \mathbf{b} simétrica para cada DWI, que resume el efecto de todas las formas de onda de gradiente aplicadas en las tres direcciones x, y, z . Entonces se usa cada DWI y su correspondiente matriz \mathbf{b} para estimar \mathbf{D} .

En DT-MRI, se calcula para cada voxel un tensor que describe la difusión local del agua, a partir de medidas de la difusión en varias direcciones [8]. A diferencia de DI, DT-MRI es una técnica tridimensional; se deben aplicar gradientes de difusión en al menos seis direcciones no colineales, no coplanares para conseguir suficiente información para estimar los seis elementos independientes del tensor de difusión.

Para medir la difusión se usa la secuencia de imagen de Stejskal-Tanner. Esta secuencia usa dos fuertes pulsos de gradiente, posicionados de forma

simétrica alrededor de un pulso de reorientación de 180° , permitiendo una medida controlada de la difusión. El primer pulso de gradiente induce un desplazamiento de fase de todos los espines; el segundo pulso invierte este desplazamiento, cancelándolo para los espines estáticos. Los espines que durante este periodo hayan sufrido un cambio de situación debido al movimiento browniano experimentarán diferentes desplazamientos de fase por los dos pulsos de gradiente, lo que significará que no están completamente realineados, y resultará en una pérdida de señal.

La difusión puede entonces calcularse según la siguiente ecuación:

$$(1) \quad S = S_0 e^{-bD},$$

donde b es el factor de peso de la difusión, definido como

$$(2) \quad b = \gamma^2 \delta^2 \left(\Delta - \frac{\delta}{3} \right) |\mathbf{g}|^2,$$

donde γ es la tasa giromagnética del protón, $|\mathbf{g}|$ es la fuerza de los pulsos de gradiente de sensibilización, δ es la duración de los pulsos de gradiente de difusión, y Δ es el tiempo entre pulsos RF de gradiente de difusión. Los valores de difusión D también se llaman coeficientes de difusión aparentes (ADC), para resaltar el hecho de que los valores de difusión generados con este procedimiento dependen de las condiciones experimentales, así como de la dirección y el gradiente de sensibilización, y otros parámetros de secuencia.

2.3.1 Cálculo del tensor de difusión

La ecuación (1) debe reescribirse para el caso anisotrópico introduciendo los vectores de gradiente normalizados $\hat{\mathbf{g}} = \mathbf{g} / |\mathbf{g}|$:

$$(3) \quad S = S_0 e^{-b\hat{\mathbf{g}}^T D \hat{\mathbf{g}}}$$

Al ser simétrico, el tensor de difusión 3×3 tiene seis grados de libertad, y es semidefinito positivo. Para estimar el tensor se necesitan, entonces, al menos seis medidas tomadas desde diferentes direcciones no colineales, además de la imagen tomada como referencia, S_0 . Es decir, es preciso obtener al menos siete imágenes con diferentes direcciones de gradiente. S_0 es la intensidad de la señal en ausencia de un campo de gradiente de sensibilización a la difusión, y da una base a la cual pueden referirse las medidas restantes. Al insertar los gradientes \mathbf{g}_k y las señales $\{S_k\}$ en la ecuación (3) de pérdida de intensidad de la señal, tenemos

$$(4) \quad S_0 = S_0 e^{-b\hat{\mathbf{g}}_k^T D \hat{\mathbf{g}}_k m}$$

resultando en un sistema de seis ecuaciones a partir de las cuales puede ser calculado el tensor:

$$(5) \quad \begin{cases} \ln(S_1) = \ln(S_0) - b\hat{\mathbf{g}}_k^T D \hat{\mathbf{g}}_1 \\ \ln(S_2) = \ln(S_0) - b\hat{\mathbf{g}}_k^T D \hat{\mathbf{g}}_2 \\ \ln(S_3) = \ln(S_0) - b\hat{\mathbf{g}}_k^T D \hat{\mathbf{g}}_3 \\ \ln(S_4) = \ln(S_0) - b\hat{\mathbf{g}}_k^T D \hat{\mathbf{g}}_4 \\ \ln(S_5) = \ln(S_0) - b\hat{\mathbf{g}}_k^T D \hat{\mathbf{g}}_5 \\ \ln(S_6) = \ln(S_0) - b\hat{\mathbf{g}}_k^T D \hat{\mathbf{g}}_6 \end{cases}$$

Resolviendo este sistema de ecuaciones para cada voxel, llegaremos al campo tensorial de difusión final [8].

2.3.2 Anisotropía y medidas macroestructurales

Las técnicas de resonancia magnética obtienen una medida macroscópica de una magnitud microscópica, lo que implica necesariamente hacer una media en cada voxel, por lo que las dimensiones del voxel influyen en el tensor de difusión medido en cualquier punto del cerebro. Existen diversos factores que afectarán a la forma del tensor de difusión representado (típicamente un elipsoide) en la sustancia blanca, como la densidad de las fibras, el grado de mielinización, el diámetro medio de las fibras y la similitud direccional de las fibras en el voxel. La naturaleza geométrica del tensor de difusión medido en un voxel es, así, una medida significativa de la organización de los tractos de fibras.

Se han propuesto en la literatura varias medidas de anisotropía diferentes. Entre ellas, las más populares son dos que se basan en la varianza normalizada de los autovalores: la anisotropía relativa (RA) y la anisotropía fraccional (FA) [9]. Una ventaja es que ambas medidas pueden ser calculadas sin calcular explícitamente los autovalores, ya que pueden expresarse en términos de la norma y la traza del tensor de difusión: la norma se calcula como la raíz cuadrada de la suma de los cuadrados de los elementos del tensor, lo que es igual a la raíz cuadrada de la suma de los cuadrados de los autovalores; y la traza se calcula como la suma de los elementos de la diagonal, lo que es igual a la suma de los autovalores.

$$(6) \quad RA = \frac{1}{\sqrt{2}} \frac{\sqrt{(\lambda_1 - \lambda_2)^2 + (\lambda_2 - \lambda_3)^2 + (\lambda_1 + \lambda_3)^2}}{\lambda_1 + \lambda_2 + \lambda_3} = \frac{\sqrt{3}}{\sqrt{2}} \frac{|\mathbf{D} - \frac{1}{3} \text{Traza}(\mathbf{D}) \mathbf{I}|}{\text{Traza}(\mathbf{D})}$$

$$(7) \quad FA = \frac{1}{\sqrt{2}} \frac{\sqrt{(\lambda_1 - \lambda_2)^2 + (\lambda_2 - \lambda_3)^2 + (\lambda_1 + \lambda_3)^2}}{\sqrt{\lambda_1 + \lambda_2 + \lambda_3}} = \frac{\sqrt{3}}{\sqrt{2}} \frac{|\mathbf{D} - \frac{1}{3} \text{Traza}(\mathbf{D}) \mathbf{I}|}{|\mathbf{D}|},$$

donde \mathbf{I} es el tensor identidad. Las constantes se incluyen para asegurar que las medidas están en el rango entre 0 y 1.

El tensor de difusión puede ser visualizado como un elipsoide cuyos ejes principales corresponden a las direcciones del sistema de autovectores. Usando las propiedades de simetría del elipsoide, el tensor puede descomponerse en dos medidas geométricas básicas. Sean $\lambda_1 \geq \lambda_2 \geq \lambda_3$ los autovalores del tensor de difusión simétrico \mathbf{D} , y sea $\hat{\mathbf{e}}_i$ el autovector normalizado correspondiente a λ_i . El tensor \mathbf{D} puede escribirse entonces como:

$$(8) \quad \mathbf{D} = \lambda_1 \hat{\mathbf{e}}_1 \hat{\mathbf{e}}_1^T + \lambda_2 \hat{\mathbf{e}}_2 \hat{\mathbf{e}}_2^T + \lambda_3 \hat{\mathbf{e}}_3 \hat{\mathbf{e}}_3^T$$

La difusión puede dividirse en tres casos básicos dependiendo del rango del tensor de difusión.

- Caso lineal ($\lambda_1 \gg \lambda_2 \simeq \lambda_3$): la difusión está fundamentalmente en la dirección correspondiente al mayor autovalor:

$$(9) \quad \mathbf{D} \simeq \lambda_1 \mathbf{D}_l = \lambda_1 \hat{\mathbf{e}}_1 \hat{\mathbf{e}}_1^T$$

- Caso planar ($\lambda_1 \simeq \lambda_2 \gg \lambda_3$): la difusión se limita a un plano, determinado por los dos autovectores correspondientes a los dos autovalores mayores,

$$(10) \quad \mathbf{D} \simeq \lambda_1 \mathbf{D}_p = \lambda_1 (\hat{\mathbf{e}}_1 \hat{\mathbf{e}}_1^T + \hat{\mathbf{e}}_2 \hat{\mathbf{e}}_2^T)$$

- Caso esférico ($\lambda_1 \simeq \lambda_2 \simeq \lambda_3$): difusión anisotrópica.

$$(11) \quad \mathbf{D} \simeq \lambda_1 \mathbf{D}_s = \lambda_1 (\hat{\mathbf{e}}_1 \hat{\mathbf{e}}_1^T + \hat{\mathbf{e}}_2 \hat{\mathbf{e}}_2^T + \hat{\mathbf{e}}_3 \hat{\mathbf{e}}_3^T)$$

En general, el tensor de difusión \mathbf{D} será una combinación de estos casos. Al expandir el tensor tomando estos casos como base, se obtiene:

$$(12) \quad \mathbf{D} = \lambda_1 \hat{\mathbf{e}}_1 \hat{\mathbf{e}}_1^T + \lambda_2 \hat{\mathbf{e}}_2 \hat{\mathbf{e}}_2^T + \lambda_3 \hat{\mathbf{e}}_3 \hat{\mathbf{e}}_3^T = (\lambda_1 - \lambda_2) \hat{\mathbf{e}}_1 \hat{\mathbf{e}}_1^T + (\lambda_2 - \lambda_3) (\hat{\mathbf{e}}_1 \hat{\mathbf{e}}_1^T + \hat{\mathbf{e}}_2 \hat{\mathbf{e}}_2^T) + \lambda_3 (\hat{\mathbf{e}}_1 \hat{\mathbf{e}}_1^T + \hat{\mathbf{e}}_2 \hat{\mathbf{e}}_2^T + \hat{\mathbf{e}}_3 \hat{\mathbf{e}}_3^T) = (\lambda_1 - \lambda_2) \mathbf{D}_l + (\lambda_2 - \lambda_3) \mathbf{D}_p + \lambda_3 \mathbf{D}_s,$$

donde $(\lambda_1 - \lambda_2)$, $(\lambda_2 - \lambda_3)$ y λ_3 son las coordenadas de \mathbf{D} en la base tensorial $\{\mathbf{D}_l, \mathbf{D}_p, \mathbf{D}_s\}$.

Las coordenadas del tensor en esta nueva base describen la cercanía del tensor a los casos genéricos de línea, plano y esfera; y por tanto pueden usarse para la clasificación del tensor según su geometría [10]. Como en el caso de la anisotropía fraccional y relativa, existen varias posibilidades de normalización, como la máxima difusividad (λ_1), la traza del tensor ($\lambda_1 + \lambda_2 + \lambda_3$), o la norma del tensor:

$$(13) \quad c_l = \frac{\lambda_1 - \lambda_2}{\lambda_1}$$

$$(14) \quad c_p = \frac{\lambda_2 - \lambda_3}{\lambda_1}$$

$$(15) \quad c_l = \frac{\lambda_3}{\lambda_1}$$

Usando el mayor autovalor del tensor, se obtienen las siguientes medidas cuantitativas de la forma para los casos lineal, planar y esférico:

$$(16) \quad c_a = 1 - c_s = c_l + c_p,$$

donde todas las medidas se encuentran en el rango de 0 a 1, y su suma es 1.

Alternativamente las coordenadas pueden normalizarse con la norma del tensor, dando:

$$(17) \quad c_l = \frac{\lambda_1 - \lambda_2}{\sqrt{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}}$$

$$(18) \quad c_p = \frac{2(\lambda_2 - \lambda_3)}{\sqrt{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}}$$

$$(19) \quad c_s = \frac{3\lambda_3}{\sqrt{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}},$$

donde aparecen los factores de escala 2 y 3 para garantizar que las medidas permanecen entre 0 y 1. Una medida de la anisotropía geométrica

con un comportamiento similar a la anisotropía fraccional (FA) es una medida que describa la desviación respecto al caso esférico:

$$(20) \quad c_a = 1 - c_s = c_l + c_p,$$

que es la suma de las medidas lineal y planar. Si normalizamos por la traza del tensor en lugar de por la norma, la medida será más parecida a la anisotropía relativa (RA).

2.3.3 Interpolación

El campo tensorial obtenido en DT-MRI es una representación discreta de un campo, el de difusión, que es continuo. Por ello son necesarios métodos de interpolación tensorial que permita obtener un tensor de difusión en cada punto del espacio. De esta forma, se puede mejorar la resolución espacial y lograr una visualización más fluida en distintas aplicaciones como, por ejemplo, la tractografía. Existen dos métodos de interpolación ampliamente utilizados en tensores: interpolación trilineal e interpolación log-euclídea.

En la interpolación trilineal los tensores situados en los puntos que rodean al punto de destino son interpolados linealmente. Esta es la aproximación más directa al problema, y representa una extensión de la interpolación lineal unidimensional. En una dimensión, se asigna un peso al tensor situado en cada uno de los dos puntos más cercanos. Estos pesos están relacionados con la distancia entre el punto en que se encuentra el tensor y el punto de interpolación: supongamos que la distancia entre el punto y el tensor \mathbf{T}_0 es t , y $(1-t)$ hasta el tensor \mathbf{T}_1 . Entonces los pesos se asignan a la inversa: $(1-t)$ para \mathbf{T}_0 y t para \mathbf{T}_1 . Los pesos de los tensores deben normalizarse para que sumen 1. Así, la expresión del tensor interpolado es la siguiente:

$$(21) \quad \mathbf{T} = (1 - t)\mathbf{T}_0 + t\mathbf{T}_1$$

En dos dimensiones, los pesos se calculan a partir de la distancia del punto a cada lado del cuadrado formado por cada tensor. El tensor interpolado \mathbf{T} se calcula del siguiente modo:

$$(22) \quad \mathbf{T} = (1 - t_0)(1 - t_1)\mathbf{T}_{00} + t_0(1 - t_1)\mathbf{T}_{01} + (1 - t_0)t_1\mathbf{T}_{10} + t_0t_1\mathbf{T}_{11}$$

El procedimiento para tres dimensiones es similar. En este caso, se toman en cuenta las distancias del punto a las caras del cubo formado por los tensores:

$$(23) \quad \mathbf{T} = (1 - t_0)(1 - t_1)(1 - t_2)\mathbf{T}_{000} + t_0(1 - t_1)(1 - t_2)\mathbf{T}_{001} + \\ (1 - t_0)t_1(1 - t_2)\mathbf{T}_{010} + t_0t_1(1 - t_2)\mathbf{T}_{011} + (1 - t_0)(1 - t_1)t_2\mathbf{T}_{100} + t_0(1 - t_1)t_2\mathbf{T}_{101} + (1 - t_0)t_1t_2\mathbf{T}_{110} + t_0t_1t_2\mathbf{T}_{111}$$

La gran ventaja de este método es su simplicidad, que resulta en una carga computacional reducida.

El segundo método es la interpolación log-euclídea [18], donde se introducen dos nuevas operaciones como son la exponencial y el logaritmo de un tensor. En general, se define la exponencial de una matriz \mathbf{M} como:

$$(24) \quad \exp(M) = \sum_{n=0}^{+\infty} \frac{M^n}{n!}$$

En el caso particular de los tensores, el cálculo se simplifica. Sea un tensor \mathbf{T} , \mathbf{U} la matriz de autovectores de \mathbf{T} y \mathbf{D} la matriz cuyos elementos diagonales son los autovalores de \mathbf{T} . Entonces el tensor tiene la forma $\mathbf{T} = \mathbf{U}\mathbf{D}\mathbf{U}^T$, y su exponencial puede del siguiente modo:

$$(25) \quad \exp(\mathbf{T}) = \mathbf{U}\exp(\mathbf{D})\mathbf{U}^T,$$

donde $\exp(\mathbf{D})$ es la exponencial de \mathbf{D} , calculada de esta forma:

$$(26) \quad \exp(\mathbf{D}) = \begin{pmatrix} \exp(\lambda_1) & 0 & 0 \\ 0 & \exp(\lambda_2) & 0 \\ 0 & 0 & \exp(\lambda_3) \end{pmatrix}$$

Se define de forma similar el logaritmo de un tensor:

$$(27) \quad \log(\mathbf{T}) = \mathbf{U}\log(\mathbf{D})\mathbf{U}^T$$

$$(28) \quad \log(\mathbf{D}) = \begin{pmatrix} \log(\lambda_1) & 0 & 0 \\ 0 & \log(\lambda_2) & 0 \\ 0 & 0 & \log(\lambda_3) \end{pmatrix}$$

Así, este método tiene tres partes. En primer lugar se calcula el logaritmo de los tensores originales. En segundo lugar, se interpolan linealmente estos nuevos tensores, como se explicó con anterioridad, obteniendo un nuevo tensor. Por último, se calcula la exponencial de este tensor para obtener el tensor final interpolado.

Por ejemplo, en el caso de la interpolación en una dimensión, el tensor interpolado \mathbf{T} sería ahora:

$$(29) \quad \mathbf{T} = \exp [(1 - t)\log \mathbf{T}_1 + t \log \mathbf{T}_2]$$

Con la interpolación log-euclídea se obtiene una notable mejora en la calidad de la interpolación, manteniendo una carga computacional baja. El método debe utilizarse si el tensor es semidefinido positivo, para mantener los logaritmos en valores reales.

2.4 Visualización

Las técnicas científicas de visualización conjugan estructura e información a varios niveles, desde los patrones a gran escala abarcando todo el conjunto de los datos, hasta las muestras individuales que lo componen. Los glifos representan múltiples valores codificándolos en la forma, tamaño, orientación y apariencia superficial de una primitiva geométrica base [16]. Idealmente, una composición adecuada de múltiples glifos a lo largo del campo tensorial puede dar pistas sobre las características a mayor escala que pueden ser exploradas posteriormente y extraídas con otras técnicas de visualización, por ejemplo, hyperstreamlines.

Los tensores de difusión pueden representarse como matrices 3x3 simétricas, con tres autovalores reales positivos, y tres autovectores ortogonales de valores reales. Un tensor de difusión \mathbf{T} puede ser factorizado como

$$(30) \quad \mathbf{T} = \mathbf{R}\Lambda\mathbf{R}^{-1},$$

donde Λ es una matriz diagonal de autovalores (ordenados por convenio $\lambda_1 \geq \lambda_2 \geq \lambda_3$), y \mathbf{R} es una matriz de rotación que transforma la base estándar en la matriz de autovectores. En adelante, los términos 'forma del tensor' y 'orientación del tensor' se referirán a los autovalores y autovectores, respectivamente, del tensor.

La visualización tensorial basada en glifos transforma la geometría inicial del glifo G en un glifo tensorial G_T según $G_T = \mathbf{R}\Lambda G$, y trasladando G_T a la localización del tensor. Al no aplicar la rotación \mathbf{R}^{-1} , las características de G alineadas con los ejes (como las aristas de un cubo, o el eje de un cilindro) se convierten en G_T en representaciones de los autovalores y autovectores del tensor.

Los autovectores de un tensor sólo contienen información sobre orientación de las líneas (su dirección no tiene signo), lo que reduce las geometrías de glifos a formas con una simetría rotacional de 180 grados.

La analogía comúnmente aceptada entre tensores 3x3 simétricos y elipsoides convierte a estos en representación natural de la difusión [17]. Los tensores de difusión son representados a menudo como elipsoides cuyo tamaño y forma reflejan el grado de difusión a lo largo de cada eje principal. La dirección de los ejes principales corresponden con los autovectores del tensor (e_1, e_2, e_3) y el tamaño relativo de cada eje queda determinado por los autovalores del tensor ($\lambda_1, \lambda_2, \lambda_3$). La primitiva geométrica para los elipsoides es la esfera. La Figura 2.1.a muestra varios elipsoides en función de los coeficientes geométricos del tensor. En la parte superior aparece el glifo isótropo, representado por una esfera. El caso lineal aparece en la parte inferior izquierda, y el planar en la parte inferior derecha. El resto de figuras muestran casos intermedios.

Esta es la representación más utilizada, sin embargo en algunos casos presenta un problema de ambigüedad de forma, al existir pares de elipsoides que, pese a tener una forma claramente distinta, pueden parecer muy similares debido al punto de vista elegido, diferenciándose solo en la sombra que se proyecta en su superficie.

Otras formas sencillas de glifos son los cilindros y los cuboides. Como se muestra en la Figura 2.1.b, los cuboides describen claramente figuras 'intermedias', alejadas de los casos genéricos de línea, plano y esfera. Sin embargo, en tensores con dos autovalores iguales, puede suceder que los autovectores se elijan arbitrariamente (dentro de un conjunto de posibilidades), lo que daría lugar a que la representación de un mismo tensor podría variar de un caso a otro.

Los glifos cilíndricos, mostrados en la Figura 2.1.c, no presentan este problema, al alinear sus ejes de rotación con el autovector para el cual la precisión numérica es mayor, es decir, el autovector asociado al mayor autovalor para el caso lineal, y el asociado al menor autovalor en el caso de un plano. Pero esto provoca un problema de discontinuidad entre los casos lineal y planar. Así, cambios arbitrariamente pequeños en la forma del tensor pueden provocar cambios discontinuos en la dirección del glifo.

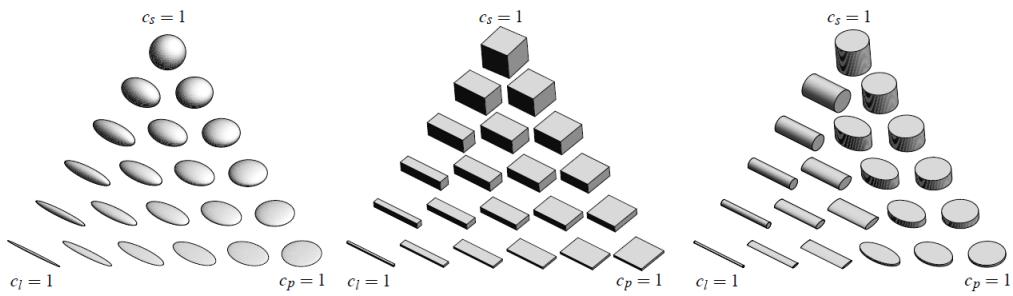


Figura 2.1. Diferentes geometrías de glifo: (a) cuboides, (b) cilindros, (c) elipsoides [16]

Una cuarta geometría, las supercuádricas, más compleja que las anteriores, pretende superar estos problemas. Las supercuádricas pueden parametrizarse explícitamente como:

$$(31) \quad q_z(\theta, \Phi) = \begin{pmatrix} \cos^\alpha \theta \sin^\beta \Phi \\ \sin^\alpha \theta \cos^\beta \Phi \\ \cos^\beta \Phi \end{pmatrix}, \quad 0 \leq \Phi \leq \pi, \quad 0 \leq \theta \leq 2\pi$$

Sin embargo esto genera un abanico demasiado amplio de formas, que debe reducirse para su uso en imágenes de difusión. En este caso, los glifos supercuádricos se definen en términos de las medidas de anisotropía geométrica c_l , c_p , y un parámetro γ controlado por el usuario que determina la prominencia de los bordes del glifo. La Figura 2.2 muestra el efecto del parámetro γ en la forma de las supercuádricas.

$$(32) \quad c_l \geq c_p \Rightarrow \begin{cases} \alpha = (1 - c_p)^\gamma \\ \beta = (1 - c_l)^\gamma \\ \mathbf{q}(\theta, \Phi) = \mathbf{q}_x(\theta, \Phi) \\ q(x, y, z) = q_x(x, y, z) \end{cases}$$

$$(33) \quad c_l < c_p \Rightarrow \begin{cases} \alpha = (1 - c_l)^\gamma \\ \beta = (1 - c_p)^\gamma \\ \mathbf{q}(\theta, \Phi) = \mathbf{q}_z(\theta, \Phi) \\ q(x, y, z) = q_z(x, y, z) \end{cases}$$

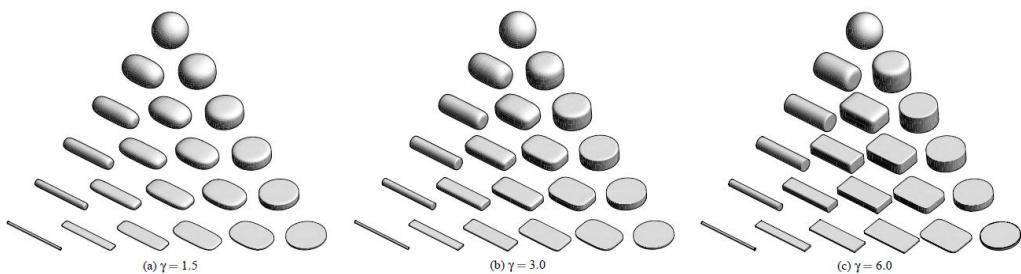


Figura 2.2. Glifos supercuádricos en función del parámetro γ [16]

Estos glifos poseen las necesarias propiedades de simetría de los elipsoides, pero muestran la forma y la orientación más claramente imitando a los cilindros y cuboides cuando es preciso. El parámetro γ controla la rapidez con que se forman los bordes según crecen c_l y c_p . Debería permitirse una elección informada de γ : la visualización de medidas ruidosas recomendarán un valor bajo de γ , más conservador, mientras que la visualización de datos de simulación, con una alta precisión, permitirán valores más altos. Por otra parte, el uso de supercuádricas aumenta la carga computacional, al requerir un número mayor de puntos para su representación.

2.5 Tractografía

La tractografía [20] es un procedimiento que ofrece una representación de las fibras nerviosas y los tractos del cerebro, utilizando técnicas de resonancia magnética (como DT-MRI) y de procesado de imagen. Esta técnica permite observar *in vivo* los tractos internos del cerebro, complicados de estudiar incluso mediante observación directa. Así, la tractografía permite conocer con más detalle la anatomía y estructura del cerebro, y avanzar en el estudio de diversas patologías.

Para realizar la tractografía, se escoge un punto de la sustancia blanca en los datos DT-MRI. Con este primer tensor, se obtiene un segundo punto siguiendo la dirección principal de la difusión, esto es, la dirección del autovector asociado al mayor autovalor del tensor. El proceso se repite para esta segunda posición, obteniendo así los sucesivos puntos que forman el tracto. Para llevar a cabo la tractografía, es necesario un campo de difusión continuo, que puede obtenerse mediante, por ejemplo, interpolación.

La tractografía tiene, sin embargo, algunas dificultades que entorpecen el proceso y pueden empeorar el resultado. La primera es la baja resolución espacial en DT-MRI. El grosor de las fibras es del orden los micrones, mientras que el tamaño del voxel en la imagen es del orden de los milímetros. Pero el problema es menor si se tiene en cuenta que las fibras suelen formar haces, conjuntos de fibras en la misma dirección. La segunda dificultad es ambigüedad en la dirección principal de la difusión. En tensores con una anisotropía planar o esférica, varios autovalores toman valores parecidos, y la dirección principal puede no ser precisa.

Capítulo 3

Imagen por tensor de esfuerzo

Este capítulo contiene una introducción a las técnicas de imagen por tensor de esfuerzo. En este capítulo se explica las distintas técnicas disponibles, los métodos para de visualización del tensor de esfuerzo, y las aplicaciones clínicas de esta modalidad de imagen.

3.1 Introducción

Los protocolos clínicos de evaluación de las enfermedades cardiovasculares hacen cada vez más necesario el uso de técnicas de imagen médica que proporcionen información sobre la anatomía y funcionalidad cardíacas.

Actualmente, si se sospecha que un paciente padece una enfermedad coronaria, se le practican una serie de pruebas diagnósticas [21], cada una más específica pero también más invasiva o costosa. En primer lugar, se obtiene un electrocardiograma sometiendo el corazón del paciente a algún esfuerzo que eleve su ritmo cardíaco.

El siguiente paso es realizar una ecocardiografía (imagen ultrasónica) al paciente, tanto en reposo como sometido a esfuerzo. La ecocardiografía es la técnica de imagen más utilizada para valorar la función cardíaca, debido a su disponibilidad, portabilidad, bajo coste y ausencia de efectos secundarios. Desgraciadamente, aunque la calidad de esta técnica va en aumento, y van apareciendo nuevas modalidades, aún ofrece una calidad de imagen baja y un número de vistas limitado.

Para obtener una imagen del suministro de sangre al tejido cardíaco, se practica un escáner de talio, para el que se inyecta un radioisótopo al paciente para obtener tomogramas, imágenes computadas a partir de múltiples proyecciones y que muestran la respuesta del isótopo dentro del tejido cardíaco. Una región con un flujo cardíaco reducido se mostrará como un 'punto frío' en las imágenes. El escáner de talio es un estándar muy usado en el diagnóstico de isquemia en el miocardio. Sin embargo, esta técnica es costosa, ofrece una resolución espacial pobre, y puede presentar falsos puntos fríos por la superposición de estructuras.

Finalmente, la angiografía coronaria por rayos X es el procedimiento más usado para determinar la localización y la gravedad de la estenosis (estrechamiento) arterial. Se introduce un catéter a través de la aorta hasta los vasos coronarios, guiado por un monitor de rayos X e inyecciones periódicas de un contraste. La estenosis coronaria aparece en el monitor como un estrechamiento o bloqueo del vaso. El procedimiento suele proporcionar imágenes de una excelente calidad, pero es invasivo, no carente de riesgo, y requiere un equipamiento costoso y personal especializado.

La investigación en resonancia magnética cardíaca ha mostrado que un solo examen, realizado enteramente con un escáner de este tipo, podría sustituir la secuencia de pruebas anterior. Aunque un escáner de resonancia magnética es costoso, la capacidad de concentrar las pruebas diagnósticas en una sola sesión reduciría significativamente los costes y sería beneficioso para el paciente [21].

La imagen por resonancia magnética es una modalidad de imagen muy apropiada para la medición del flujo sanguíneo y el movimiento de los tejidos. Las imágenes pueden adquirirse en posiciones y orientaciones libremente definidas, y el contraste entre tejidos blandos es excelente.

3.2 Técnicas para la detección del movimiento en MRI

Las propiedades mecánicas de corazón permiten un diagnóstico temprano y un mejor seguimiento del paciente. El músculo cardíaco tiene propiedades mecánicas anisotrópicas (varían según la dirección), y dependientes del tiempo. Durante el ciclo cardíaco, el miocardio sufre grandes deformaciones elásticas como consecuencia de la contracción y relajación del músculo. Factores fisiológicos como la capacidad de bombeo de los ventrículos, la distribución del flujo coronario y la vulnerabilidad regional a la isquemia y al infarto se ven afectadas por las propiedades mecánicas del miocardio. Una descripción en cuatro dimensiones (tres espaciales y una temporal) del movimiento del miocardio puede ayudar a describir sus propiedades mecánicas.

El método estándar para la detección del movimiento consiste en seguir objetos usando secuencias temporales de datos 2-D y 3-D. A partir de estas imágenes, el contorno y los bordes del tejido son detectados mediante técnicas de segmentación y se usan métodos de registrado para seguir su movimiento. Para ello, se pueden utilizar varias modalidades de imagen en función de la aplicación. Si el órgano en movimiento puede ser observado directamente, como un pie o un brazo, se pueden emplear marcadores especiales o técnicas estéreo de visión por ordenador. La tomografía computerizada puede utilizarse para obtener imágenes del interior del cuerpo, pero la dosis de rayos X se hace crítica cuando son necesarios varios conjuntos de datos 3-D.

La imagen por resonancia magnética (MRI) presenta diversas ventajas, al ofrecer un mejor contraste entre tejidos blandos y una mayor libertad en la posición y orientación de las imágenes. Las técnicas MRI específicas pueden no solo ofrecer imágenes anatómicas detalladas, sino además seguir el movimiento de los tejidos, o dar directamente datos cuantitativos del movimiento [22]. Estas técnicas son: MRI con etiquetado, o TMRI; MRI por contraste de fase, o PCMRI; métodos MRI de campo pulsado basados en gradiente.

3.2.1 TMRI

En MRI pueden crearse sobre los tejidos marcadores magnéticos, o etiquetas. Así, cuando el tejido es observado después de un determinado tiempo, los cambios en la forma y posición de las etiquetas reflejan el movimiento del tejido [23][24]. Comúnmente se utiliza un patrón de franjas paralelas o una combinación de etiquetas en dos planos ortogonales formando una retícula. Las secuencias de etiquetado básicas se integran actualmente en las librerías de secuencias de pulsos de las máquinas de resonancia magnética.

La operación de etiquetado puede verse como una excitación espacial selectiva mediante el uso de gradientes y pulsos de radiofrecuencia. Son excitados múltiples planos de magnetización para un etiquetado por saturación. La magnetización es entonces desfasada mediante el uso de pulsos de gradiente, de forma que no tenga una aportación significativa en las imágenes adquiridas posteriormente.

El efecto de la excitación y el desfase es dejar regiones 'nulas' en la magnetización longitudinal, que aparecerán como nulos en las imágenes en las zonas etiquetadas. Los tejidos se muestran oscuros en las imágenes de resonancia magnética.

El movimiento de los tejidos que ocurre a partir del etiquetado altera el patrón de etiquetas. La detección de las mismas permite reconstruir el movimiento del tejido subyacente y estimar la deformación del miocardio. Un observador podría detectar regiones en las que el miocardio no se contrae o donde la contracción es mucho menor que en el resto del tejido. Para distinguir anomalías menos visibles y para obtener estimaciones cuantitativas de la función cardíaca hacen falta métodos numéricos.

3.2.2 PCMRI

Una aproximación diferente al análisis del movimiento se basa en la sensibilidad de la fase de la señal MR al movimiento [25]. En principio fue pensado para la medición del flujo sanguíneo, pero también se utiliza para obtener medidas del esfuerzo en el miocardio.

Con esta técnica se obtiene un valor de la velocidad del tejido para cada píxel. El fundamento consiste en adquirir datos con dos gradientes de codificación de la velocidad pero dejando invariables el resto de parámetros, y restar las dos imágenes de fase. La imagen resultante será

proporcional al flujo (o movimiento del tejido) si se puede asumir que el tejido subyacente tiene una velocidad constante durante la adquisición.

Los gradientes de codificación de velocidad no afectan a los protones estacionarios pero imponen desplazamientos de fase en los protones en movimiento. Para eliminar los efectos de fase de otras fuentes diferentes que el flujo o el movimiento, se adquiere un escáner de referencia. Dado que la información sobre la velocidad sólo puede obtenerse en una dirección en cada instante, se deben obtener cuatro medidas independientes para llegar a un conjunto de datos 3-D. Para mostrar tanto el flujo como el movimiento del tejido, es necesaria una adecuada planificación de los gradientes para eliminar el aliasing y la eliminación involuntaria de señales [22].

3.2.3 Métodos de campo de gradiente pulsado

La sensibilidad al movimiento inherente a la resonancia magnética fue reconocida poco después del propio fenómeno de resonancia magnética [26]. Es de notar que aparecieron propuestas para usar gradientes de campo magnético para la medición de distribuciones de velocidades incluso antes de la aparición de la MRI. Desde entonces se han desarrollado múltiples técnicas usando un mecanismo común de codificación del movimiento: un par de gradientes de campo pulsado (PFG). Una gran ventaja de estos esquemas es que ofrecen una medida directa sobre el desplazamiento del tejido, simplificando significativamente el postprocesado de la imagen. Combinando métodos PFG y secuencias de imágenes MR, se pueden obtener directamente mapas de desplazamiento: el desplazamiento es proporcional a la fase en cada píxel.

3.3 Estimación del tensor de esfuerzo

La contracción o relajación del tejido muscular produce un cambio en la forma, una deformación material que puede cuantificarse midiendo el esfuerzo en pequeñas regiones. La estimación del esfuerzo en un punto del espacio en un instante determinado viene dado por un tensor de esfuerzo, una matriz 3x3 cuyos elementos indican deformaciones del material como estiramientos (o compresiones, estiramientos negativos) y cortes.

El esfuerzo mide cambios locales en la forma y por tanto no se ve afectado por traslaciones globales (desplazamientos constantes en el espacio). Así, el desplazamiento del corazón en el espacio durante el ciclo cardíaco no afecta a los valores del esfuerzo.

Durante las distintas fases del ciclo cardiaco, la pared muscular se hace más gruesa (se estira) o más delgada (se comprime). Los autovectores del tensor de esfuerzo se sitúan en las direcciones principales del estiramiento o la compresión. La fase de estiramiento se traduce en autovalores positivos, y la compresión en autovalores negativos (estiramiento negativo).

Para calcular la tasa de esfuerzo a partir de un campo de velocidades, se calcula el campo de gradientes de velocidad 3x3 \mathbf{L}_{ij} según:

$$(34) \quad \mathbf{L}_{ij} = \frac{\partial u_i}{\partial x_j},$$

donde u_i , $i=1,2,3$ son las tres componentes de la velocidad en la dirección x_j , $j=1,2,3$. La tasa de esfuerzo se representa por el tensor \mathbf{D}_{ij} , que es la parte simétrica de \mathbf{L}_{ij} :

$$(35) \quad \mathbf{D}_{ij} = \frac{1}{2} \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right)$$

Para el caso de tensores de esfuerzo bidimensionales, el cálculo se realiza de manera idéntica, con $i, j = 1, 2$.

Los autovalores y autovectores de la matriz de tasa de esfuerzo son los valores y direcciones principales de la tasa de esfuerzo en el miocardio. El signo del autovalor distingue entre estiramiento positivo y negativo del material en la dirección del autovector correspondiente.

3.3.1 Interpolación

La interpolación del tensor de esfuerzo permite obtener un campo tensorial continuo, donde el esfuerzo puede calcularse para cualquier punto de la imagen. Se puede utilizar para tensor de esfuerzo cualquiera de las dos técnicas explicadas en la sección 2.3.3 : interpolación lineal e interpolación log-euclídea.

Sin embargo, cabe destacar un método más, específico para tensores bidimensionales como el tensor de esfuerzo [36]. Dado un tensor 2x2 simétrico \mathbf{T} , se extraen sus autovalores (λ_1 y λ_2) y autovectores (e_1 y e_2), unitarios y ortogonales entre sí. A continuación se calcula el ángulo θ , formado por el primer autovector e_1 y el eje de abscisas. Los valores de λ_1 , λ_2 y θ determinan el tensor \mathbf{T} de forma única. Así, se pueden interpolar estos tres valores para

obtener los autovalores y el ángulo θ del nuevo tensor y, con ellos, los autovalores y los coeficientes del tensor interpolado.

Con esto se obtiene una interpolación muy cercana a la visualización, ya que se interpolan los valores y la dirección principal de la difusión, es decir, las características que se representan gráficamente.

3.4 Visualización

El tensor de esfuerzo tridimensional puede visualizarse mediante un elipsoide en cada punto [35]. Las direcciones de los tres ejes del elipsoide se alinean con los tres autovectores del tensor, y la longitud del elipsoide en la dirección de cada autovector viene dada por la magnitud del autovalor correspondiente. Para facilitar la visualización y evitar grandes diferencias de tamaño entre los distintos elipsoides, los autovalores son normalizados, de manera que la magnitud del mayor autovalor sea uno. Esto permite visualizar todos los elipsoides, manteniendo la forma de cada uno de ellos. Para no perder información, puede representarse la magnitud original del mayor autovalor mediante el color del elipsoide.

En dos dimensiones, la forma más sencilla de representar el tensor de esfuerzo 2x2 sería trasladar el caso 3-D al 2-D, y utilizar elipses. Así, el tensor vendría representado por una elipse cuyos ejes estarían alineados con los dos autovectores del tensor, y la longitud de dichos ejes vendría dada por los autovalores de la matriz.

En otro método de visualización, el tensor se representa interpretando los elementos del mismo. Los elementos diagonales E_{ii} pueden verse como la elongación o compresión en la dirección x_i , mientras que los dos elementos restantes corresponden a los esfuerzos de torsión [30]. Así, en un cuadrado infinitesimal deformado sin cambio de área pueden aproximarse por la variación del ángulo entre los ejes.

El tensor de esfuerzo se representa, como se muestra en la Figura 3.1 por un rectángulo infinitesimal en cada voxel, cuyas diagonales se orientan según la dirección de los autovectores, y la longitud de cada diagonal viene dada por:

$$(36) \quad SD_i = \sqrt{2} \left(\frac{L}{4} + \frac{L}{4} R_1^i + \frac{L}{2} R_2 \right),$$

donde L es el lado del cuadrado, $i = 1,2$, R_1 es la tasa de normalización dada por $R_1^i = \frac{1}{2}(\lambda_i + 1)$ y R_2 es la norma euclídea del campo de deformación

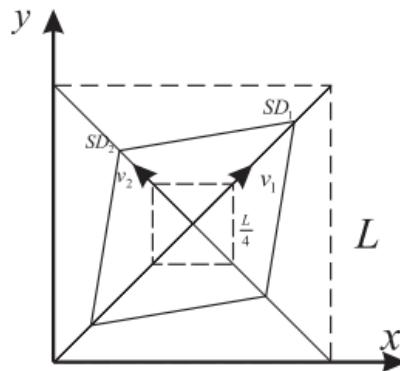


Figura 3.1. Representación de glifos 2D [29]

normalizado en cada instante. El cuadrado tendrá un lado mínimo de $L/4$, y máximo de L .

De este modo se muestran más claramente la compresión y la dilatación. Estas son las variables que se quieren representar de forma más intuitiva, y el uso de una forma poligonal en lugar de una curva como la elipse permite hacerlo de forma más directa. Como contrapartida, en el caso de tener dos autovalores de igual magnitud, la orientación de los autovectores y del cuadrado será arbitraria, y la figura no aportará ninguna información. Sin embargo, este efecto pierde importancia si tenemos en cuenta que, aunque la orientación no esté clara en un píxel concreto, la orientación de varios tensores en conjunto no se ve afectada, y pueden ser vistos como un todo.

Una tercera modalidad simula el esfuerzo mediante un modelo matemático [35]. Se muestran dos cilindros concéntricos que representan los bordes de la pared ventricular. Los radios de estos cilindros varían con el tiempo según una función sinusoidal. La altura también varía con el tiempo, de forma que el volumen total permanece constante. En este modelo, se representa el esfuerzo en el grosor de la pared ventricular, es decir, en el espacio entre ambos cilindros, y se suele representar mediante elipsoides.

Capítulo 4

Librerías empleadas: ITK, VTK, FLTK

En este capítulo se realiza un repaso por el origen y las principales características de las herramientas que se van a emplear para el desarrollo del proyecto. ITK se emplea en Saturn para la representación de datos y el procesado de imagen. VTK se utiliza en visualización, y FLTK para la creación de interfaces.

4.1 ITK

4.1.1 Introducción

ITK (*Insight ToolKit*) es un software para el procesado, segmentación y registrado de imágenes [40][41]. Aunque se puede usar para cualquier propósito, ITK está orientado principalmente a la imagen médica, donde son especialmente importantes las funcionalidades que ofrece ITK. La segmentación, por una parte, es el proceso de identificación y clasificación de los datos disponibles en una representación digital, típicamente una imagen médica. El registrado, en cambio, es la tarea de alinear varias imágenes, o de establecer correspondencias entre sus puntos. Por ejemplo, para combinar una imagen de resonancia magnética con una tomografía.

ITK es código abierto y multiplataforma, ya que utiliza la herramienta CMake. Es orientado a objetos, y está programado en C++, aunque se pueden generar, de manera automática, interfaces para trabajar con Tcl, Java y Python. Además emplea un modelo de programación conocido como programación genérica.

ITK surge en 1999, cuando la Biblioteca Nacional de Medicina, del Instituto Nacional de Salud de Estados Unidos, obtuvo un contrato para desarrollar la herramienta.

4.1.2 Características generales

A continuación se enumeran las principales características de ITK, algunas de las cuales se explican con más detalle en diferentes apartados:

- Código abierto, bajo licencia BSD.
- Orientado a objetos: ITK está escrito en C++.
- Multiplataforma, gracias al uso de CMake.
- Programación genérica, que le dota de eficiencia y flexibilidad.
- Manejo de memoria mediante punteros inteligentes y conteo de referencias.

- Manejo de eventos mediante el modelo observador (*subject-observer*, en inglés).
- Soporte multihilo.
- Representación numérica: emplea la librería numérica VNL (*Vision Numerics Library*), que incluye representación y operaciones para matrices, polinomios, funciones, etc.
- Representación de datos: conjuntos de datos estructurados y no estructurados, con flexibilidad en el tipo de píxel.
- Registrado: soporta cuatro tipos de registrado: registrado de imagen, multirresolución, registrado basado en ecuaciones por derivadas parciales (PDE) y por el método de elementos finitos (FEM).
- Envoltorios: ITK está escrito en C++, pero se pueden generar correspondencias con Python, Tcl y Java. Esta característica está aún en desarrollo.

4.1.3 Programación genérica

La programación genérica es un modelo de desarrollo que organiza las librerías en componentes software genéricos. El objetivo es que estos componentes puedan “conectarse” entre sí de un modo eficiente y flexible. La idea es generalizar las funciones utilizadas para que puedan ser reutilizadas. Esto se consigue parametrizando al máximo el algoritmo, y utilizando tipos de datos genéricos.

En la programación genérica, se emplean tres tipos de objetos: *containers* para almacenar la información, iteradores para acceder a los datos, y algoritmos genéricos, que utilizan los *containers* y los iteradores para crear algoritmos básicos.

La programación genérica es posible en C++ mediante el mecanismo de programación con *templates* o plantillas. Esta técnica permite al desarrollador escribir código utilizando tipos de dato indeterminados. El usuario es quien debe concretar el tipo de dato al utilizar el código, y el compilador se encarga de comprobar la compatibilidad. Este tipo de dato puede ser un tipo nativo (como *float* o *double*) o una clase.

ITK utiliza la programación genérica en su desarrollo. La ventaja de este hecho es que sus clases soportan casi cualquier tipo de dato. Por ejemplo, una imagen en ITK puede tener una gran variedad de tipos de píxel, desde tipos nativos, a clases de ITK como vectores o matrices o cualquier otra clase debidamente definida por el usuario.

4.1.4 Gestión de la memoria

ITK emplea un sistema de gestión de memoria que lleva la cuenta del número de referencias a un objeto. Cuando un objeto referencia a otro, se hace una llamada al método *Register()*, que incrementa el contador para el objeto referenciado. Cuando una referencia al objeto desaparece, se invoca el método *Delete()*, que decrementa el contador. Si el número de referencias llega a cero, el objeto es destruido y se libera su espacio de memoria.

La gestión de las referencias se realiza en ITK con la clase *SmartPointer*. Esta clase actúa como un puntero tradicional, pero además se encarga de llamar a los métodos *Register()* y *UnRegister()* (similar a *Delete()*) cuando empieza a apuntar a un objeto y cuando deja de hacerlo. Esto evita que el usuario tenga que hacerlo manualmente, y mejora el aprovechamiento de la memoria.

4.1.5 Representación de datos

En ITK, la información está representada por los llamados objetos de datos, que circulan por el sistema y atraviesan los filtros y objetos de procesado del pipeline. Existen dos tipos de objetos de datos principales en ITK, la imagen y la malla.

La imagen (*itk::Image*) representa un conjunto de datos n-dimensional con una estructura regular. La imagen está compuesta por píxeles de datos, separados una distancia fija en cada dirección. Las direcciones de muestreo son paralelas a los ejes de coordenadas.

El tipo de píxel para la imagen es arbitrario, siempre que soporte ciertos operadores, y se especifica como parámetro de plantilla al instanciar la imagen. Los tipos de píxel más comunes son los tipos simples (*int*, *float*, etc.) o las clases de ITK (*Vector*, *Matrix*, etc.), aunque puede ser cualquier otro tipo definido por el usuario.

El otro tipo de objeto de datos es la malla (*itk::Mesh*). Las mallas representan conjunto de datos sin estructura, formados por celdas, puntos y una lista que representa las conexiones entre ellos.

4.1.6 El pipeline de datos

El pipeline está formado por los objetos de procesado, que toman uno o varios objetos de datos a su entrada y generan otros a la salida. Existen tres tipos de objetos de procesado, fuentes, filtros y *mappers*, aunque a menudo se utiliza el término filtro para denominarlos a todos ellos. Las fuentes (por ejemplo un lector de ficheros) generan los datos. Los filtros toman un conjunto de datos, lo procesan, y generan datos nuevos. Los *mappers* toman un conjunto de datos y generan una salida a otros sistemas, por ejemplo un fichero o un sistema de visualización. Los diferentes elementos del pipeline se conectan entre sí mediante los métodos *SetInput()* y *GetOutput()*.

4.1.7 Filtrado

Los filtros son parte fundamental del pipeline de ITK, ya que son los encargados del procesado de la imagen. Los filtros toman un conjunto de datos a su entrada, y generan una nueva información a la salida. ITK implementa los filtros más utilizados en procesado de imagen, como el filtro de umbral, la detección de bordes, el mapeado de intensidades, los filtros de gradiente, filtros de suavizado, etc.

4.1.8 Lectura y escritura en ficheros

El acceso, lectura y modificación de ficheros de datos es posible en ITK gracias a la clase *itk:ImageIOBase*, una clase abstracta que encapsula la interacción con los ficheros. *ImageIOBase* tiene una variedad de subclases para los distintos formatos de fichero. ITK soporta, entre otros formatos, *BMP*, *DICOM*, *JPEG*, *Meta*, *NRRD*, *PNG*, *TIFF* o *VTK*. Sólo algunos de ellos soportan tipos de pixel no escalares (vectores, matrices, etc.)

Para la lectura y escritura de ficheros se utilizan las clases *ImageFileReader* e *ImageFileWriter*, respectivamente. ITK también soporta la escritura de una imagen en varios ficheros (por ejemplo una imagen 3D en una serie de ficheros de imagen 2D). Se utilizan las clases *ImageSeriesReader* e *ImageSeriesWriter*.

4.2 VTK

4.2.1 Introducción

VTK (*Visualization ToolKit*) es un software para el procesado, representación y visualización por ordenador de imágenes 3D [43][44]. La visualización es el proceso de convertir información, típicamente numérica, en imágenes. La visualización resulta útil en campos como la imagen médica, donde técnicas como la resonancia magnética generan una gran cantidad de datos, muy complicados de interpretar directamente. Otras áreas de aplicación son la simulación de procesos físicos o la industria del entretenimiento.

VTK es código abierto y multiplataforma, gracias al empleo de CMake. Es orientado a objetos, y está programado en C++, aunque se puede trabajar también con Tcl, Java y Python. VTK no ofrece una interfaz gráfica de usuario, pero es compatible con sistemas como *Tk* o *X/Motif*.

VTK surgió en 1993 como complemento al libro de texto *The Visualization Toolkit An Object Oriented Approach to 3D Graphics* [42]. El objetivo del libro era colaborar con otros investigadores para desarrollar un sistema abierto para crear aplicaciones gráficas y de visualización. Una vez escrito el núcleo del código, se desarrolló gracias a las aportaciones de programadores y organizaciones de todo el mundo. Actualmente se emplea en aplicaciones comerciales y de I+D, y en diversas herramientas de visualización como *Paraview*, *VisIt*, *VisTrails*, *Slicer*, *MedINRIA*, *MayaVi*, *OsiriX* o *Saturn*.

4.2.2 Características generales

Estas son algunas de las características generales de la librería VTK:

- Código abierto, bajo licencia BSD.
- Orientado a objetos: VTK está escrito en C++, lo que le dota de velocidad y eficiencia.
- Multiplataforma, gracias a CMake.
- Representación de datos: soporta conjuntos de puntos sin estructura, datos poligonales, imágenes, volúmenes y datos en rejilla.

- Filtros: soporta una multitud de algoritmos de procesado.
- Control implícito de la ejecución.
- Otros lenguajes: VTK ofrece interfaces para Python, Java y Tcl.

4.2.3 El modelo de gráficos

El modelo de gráficos es uno de los dos grandes subsistemas de VTK, junto con el pipeline de visualización. Se trata de una capa abstracta que asegura la portabilidad entre distintos lenguajes gráficos (como *OpenGL*). Los elementos principales del modelo de gráficos son los siguientes: ventana de renderización, renderizador o *renderer*, actor y *mapper*, así como la cámara y las luces.

El *mapper* es el primer elemento de la cadena, que toma un objeto de datos a su entrada e interpreta cómo visualizarlo. Cada *mapper* se asocia con un actor. Un actor contiene información sobre la visibilidad, la orientación, el tamaño y la posición del objeto, y puede ser 2D o 3D. Los actores, por su parte, se añaden a un *renderer*, que se encarga de dibujar la imagen. En último lugar se encuentran las ventanas, que contienen uno o varios *renderers* y muestran su contenido.

El modelo de gráficos pretende emular el proceso de visión de la vida real. De este modo, en VTK lo que se visualiza es una escena formada por luces, cámaras y actores. Las fuentes de luz virtuales se consideran puntuales, y emiten en todas direcciones desde una distancia infinita, de forma que todos los rayos de luz llegan en direcciones paralelas. Se pueden variar las propiedades de la luz, y utilizar varias fuentes. Siguiendo este modelo, los actores que componen la escena reciben la luz y la reflejan. El tercer elemento es la cámara. La cámara recoge esta luz, y representa el punto de vista del usuario. Las cámaras pueden acercarse y alejarse de la escena, moverse o rotar.

Por último, cabe mencionar que el punto de vista puede controlarse mediante código, o bien haciendo uso de la clase *vtkRenderWindowInteractor*, que permite manejar la escena con el ratón y el teclado.

4.2.4 El pipeline de visualización

El pipeline de visualización es el segundo gran subsistema de VTK, y su tarea es convertir los datos de entrada en formas que puedan ser mostradas por

el sistema de gráficos. Se compone de objetos de datos y de procesado o filtros, que se interconectan de forma que la salida del último filtro será la entrada para el sistema de gráficos.

Los objetos de datos representan y dan acceso a los datos, y no son creados de forma explícita sino que se generan a la salida de los filtros. Los objetos de datos se pueden clasificar en función del tipo de celda, del tipo de dato, y de la estructura del *dataset*. Entre los tipos de celda aparecen celdas lineales como líneas, triángulos o píxeles, y no lineales. Los tipos de dato pueden ser escalares, vectores, normales, etc. En cuanto a la estructura del *dataset*, aparecen tipos como la imagen (un conjunto de píxeles con una estructura ordenada y un espaciado regular), los datos poligonales o el conjunto de puntos sin estructura.

Por otra parte, los objetos de procesado o filtros tienen la tarea de operar sobre los datos de entrada para generar unos datos de salida. Los filtros pueden clasificarse en fuentes, que generan los datos iniciales, objetos de filtrado, o *mappers*, que suponen el final del pipeline. Un filtro puede realizar tareas sencillas, como rotar un objeto, o más complicadas como detectar los bordes de una imagen.

4.2.5 Gestión de memoria

La visualización tiene unos requerimientos elevados en términos tanto de carga computacional como de consumo de memoria. Muchos algoritmos de visualización son costosos, debido al tamaño de los datos y a la complejidad del propio algoritmo. La primera posibilidad es almacenar constantemente los objetos de datos intermedios. De este modo sólo se procesan una vez y el gasto computacional es reducido, pero requiere de un espacio de memoria elevado. La segunda opción es almacenar los resultados intermedios hasta que son utilizados, minimizando así el espacio ocupado en memoria.

4.2.6 Control implícito de la ejecución

VTK implementa un control implícito de la ejecución del pipeline de visualización, de modo que la ejecución sólo ocurre cuando se solicita la salida de un filtro. El control se basa en los métodos *Update()* y *Execute()* (actualizar y ejecutar, respectivamente). El método *Update()* es llamado normalmente cuando el usuario solicita la renderización de la escena. El *mapper* invoca el método *Update()* en sus entradas y el proceso se extiende hacia atrás hasta llegar a las

fuentes. Al recibir la orden, cada objeto comprueba si ha sido modificado desde que fue ejecutado por última vez, y en tal caso invoca el método *Execute()* sobre sí mismo. Si no ha sido modificado, la salida está actualizada y no es necesario procesarla de nuevo.

4.3 FLTK

4.3.1 Introducción

FLTK (*Fast Light ToolKit*) es un herramienta de creación de interfaces gráficas de usuario para Windows, Linux y MacOSX [45][46]. FLTK proporciona las funcionalidades de una interfaz moderna sin sobrecargar el código, y soporta gráficos 3D a través de *OpenGL*.

FLTK es código abierto y multiplataforma, gracias al uso de CMake. Está escrito en C++, y es compatible a bajo nivel con las distintas plataformas, lo que mejora su velocidad y eficiencia. Ofrece además una aplicación gráfica para crear las interfaces, FLUID.

FLTK nace en los años 80 cuando Bill Spitzak, ingeniero de Sun Microsystems, se decidió a reescribir la herramienta *Forms* para que fuera compatible con *OpenGL*, y comenzó a introducir sus propias mejoras. Actualmente es mantenida por un pequeño grupo de desarrolladores con un repositorio central en los EE.UU.

4.3.2 Características generales

Estas son algunas de las características de FLTK:

- Código abierto, bajo licencia GNU GPL.
- Orientado a objetos: FLTK está escrito en C++, lo que le dota de velocidad y eficiencia.
- Multiplataforma, gracias a CMake.
- Soporte para gráficos 3D mediante OpenGL.

- Escrito directamente sobre el núcleo de las librerías gráficas, para optimizar la velocidad, el rendimiento y el tamaño del código.
- Compatibilidad a bajo nivel entre las versiones X11, Win32 y MacOS.
- Código fuente sencillo, entendible y modificable directamente por el desarrollador.

4.3.3 Tratamiento de los eventos

Las aplicaciones FLTK se basan en un modelo simple de procesado de eventos. Las acciones del usuario como el movimiento del ratón, los clics, o las pulsaciones del teclado generan eventos que se envían a la aplicación.

FLTK soporta también pseudo-eventos de temporización, de ficheros y de estado ocioso. Los pseudo-eventos son llamadas a funciones realizadas sin una intervención directa del usuario. Las funciones de temporización son llamadas cuando expira un tiempo determinado, por ejemplo en acciones que se repiten cada cierto tiempo. Los eventos de ficheros ocurren cuando un archivo está listo para ser leído o escrito, y son comunes en comunicaciones a través de la red. Los eventos de estado ocioso, por último, aparecen cuando la interfaz no está realizando ninguna acción, y puede ser por ejemplo refrescar un el estado de la pantalla.

4.3.4 FLUID

FLUID (*Fast Light User Interface Designer*) es un editor gráfico empleado para generar código fuente FLTK. El programa almacena su estado en ficheros con la extensión *.fl*, que se convierten después en ficheros *.cxx* y *.h* de C++. Los ficheros *.fl* son de texto, y el desarrollador puede editarlos directamente.

FLUID permite situar de manera gráfica los diferentes elementos de interfaz de FLTK, como botones, cuadros de texto, barras de menú, etc. También permite configurar las llamadas (*callbacks*) a los distintos elementos, o definir métodos auxiliares a la interfaz.

4.4 CMake

CMake es un sistema para configurar la compilación independientemente del sistema operativo y del compilador [47]. El sistema crea ficheros de configuración en cada directorio fuente, que son usados para crear los ficheros de compilación típicos (como el Makefile en Unix), que pueden emplearse normalmente.

CMake puede compilar código fuente, crear librerías, generar envoltorios y producir ejecutables. Soporta compilaciones en lugares arbitrarios, y varias combinaciones para un mismo directorio. Soporta también compilaciones estáticas y dinámicas de librerías, y genera un fichero de caché que puede ser usado con un editor gráfico.

CMake es código abierto, bajo licencia de la compañía Kitware. Está diseñado para soportar jerarquías de directorios complejas y aplicaciones dependientes de múltiples librerías. El proceso está controlado por un fichero llamado CMakeLists.txt en cada directorio, que contiene una serie de comandos. El conjunto de estos ficheros conforma un proyecto, que es procesado por CMake para generar los ficheros de compilación.

CMake nace por la necesidad de ITK de un entorno de compilación multiplataforma potente. Está influenciado por el sistema *pcmake* (asociado a VTK), y en la herramienta *configure* de Unix. CMake comenzó en el año 2000, y su desarrollo se ha visto impulsado por las aportaciones de los desarrolladores que empezaron a utilizarlo.

Capítulo 5

Saturn y otras interfaces

En este capítulo se explica detalladamente el propósito, la estructura y el funcionamiento de la herramienta Saturn, especialmente de aquellas partes relevantes para este proyecto. Al final del capítulo se hace un repaso de las interfaces de imagen médica existentes, más allá de Saturn.

5.1 Introducción a Saturn

Saturn es una herramienta de procesado de imágenes de ultrasonido, y forma parte del proyecto de investigación "Desarrollo de Sistemas Avanzados de Ultrasonografía Diagnóstica e Intervencionista (USIMAG)" [50]. El objetivo de este proyecto es el desarrollo de nuevas técnicas de procesado de imagen en ultrasonografía, y su aplicación clínica en neurología y ecografía torácica y abdominal. El proyecto Usimag está financiado por la Comisión Interministerial de Ciencia y Tecnología, y en él participa el Laboratorio de Procesado de Imagen (LPI) de la Universidad de Valladolid, así como el Grupo de Imagen, Tecnología Médica y Televisión (GIMET) de la Universidad de Las Palmas de Gran Canaria, y el MedicLab de la Universidad Politécnica de Valencia.

Dentro de este contexto surge UsimagTool, renombrado más tarde como Saturn. Desde el inicio de su desarrollo, Saturn tiene una serie de objetivos claros, necesarios para una herramienta de imagen médica, y algunos de los cuales no cumplen el resto de programas disponibles [48][49]:

- Código abierto, lo que permite a otras personas estudiar, modificar y reutilizar el código.
- Eficiente, robusto y rápido, mediante el uso de un lenguaje orientado a objetos como C++.
- Modularidad y flexibilidad en el desarrollo, que mejora la eficiencia y permite añadir y modificar funcionalidades de forma ágil y sencilla.
- Multiplataforma: al funcionar sobre distintos sistemas operativos, puede llegar a más gente.
- Usabilidad: una interfaz de usuario sencilla que facilita al personal médico la interacción con la herramienta.
- Documentación: un sitio web bien documentado, así como manuales y tutoriales para usuarios y desarrolladores.

Se podrían detallar las carencias de otras herramientas en cuanto a los puntos anteriores. Se pueden tomar como ejemplo dos de las más importantes: MedInria, que no es código abierto y Slicer, que emplea una arquitectura más compleja que Saturn, lo que dificulta la labor de los desarrolladores. Saturn cumple los objetivos anteriores, a la vez que cuenta con diversos algoritmos para el procesado de imágenes de ultrasonido, lo que le diferencia de otras herramientas.

El código de Saturn se basa fundamentalmente en tres librerías: VTK, ITK y FLTK. Estas librerías son de código abierto, orientadas objetos (C++), lo que las hace más eficientes, y cuentan con el apoyo de la comunidad científica, por lo que son idóneas para esta aplicación.

ITK es actualmente la librería más potente para el procesado de imagen médica. Incluye los algoritmos de procesado de imagen más importantes, y está en constante desarrollo. FLTK, con su herramienta Fluid, permite crear interfaces gráficas de un modo sencillo y rápido, y ofreciendo un gran número de posibilidades al desarrollador. FLTK se utiliza junto con VTK para la visualización.

5.2 Interfaz de usuario

La Figura 5.1 muestra la interfaz de Saturn. Se pueden distinguir tres partes: un área de datos, una de configuración y otra de visualización. El área de datos se encuentra en la parte superior izquierda. Permite elegir el tipo de dato que se quiere visualizar (Scalar, Tensor, Model, DWI), muestra una lista con los volúmenes de ese tipo que se encuentran cargados en el programa, y permite realizar diversas operaciones sobre ellos, como estimar los tensores a partir de los datos de difusión, o visualizar diferentes magnitudes escalares en los planos 2D. Además contiene accesos directos a varios paneles de configuración.

El área de configuración contiene uno entre los diversos paneles de configuración que implementa Saturn como se muestra en la Figura 5.1 y la Figura 5.2. En la Figura 5.1 aparece el panel Preferencias, que ofrece diversos controles de visualización, como acercar y alejar la imagen, moverla o darle la vuelta, o mostrar en la visualización diversas propiedades de los datos (corte visualizado, dimensiones del volumen, tamaño del voxel, etc.).

El panel de Scalar Magnitudes (Figura 5.2.a) permite escoger qué magnitud se visualiza en los cortes 2D. Aparecen coeficientes de anisotropía como FA y RA, coeficientes geométricos, elementos del tensor o autovalores.

El panel Tractography (Figura 5.2.b) se utiliza para realizar la tractografía a partir de los puntos pulsados por el usuario en las imágenes. Se puede elegir el color de los tractos mostrados (un color específico para todo el tracto o colorear cada tensor según un determinado parámetro), el radio de visualización, la longitud de cada tramo, y los umbrales de anisotropía fraccional y curvatura, así como varios parámetros relacionados con la región de interés (ROI), y diversos tipos de tractografía. Saturn implementa tractografía por fuerza bruta y el método de Runge-Kutta.

El panel Tractography Auto (Figura 5.2.c) permite realizar automáticamente la tractografía, eligiendo en la lista los tractos de fibras que se desean computar y visualizar.

El cuarto panel, Fibers Edit o Model Prop (Figura 5.2.d), permite editar la visualización de los tractos. Permite, entre otras cosas, cambiar el tamaño de los tractos, colorearlos según un parámetro diferente, o cambiar las propiedades de la luz que inciden sobre ellos.

Por último, el panel Measures (Figura 5.2.e) ofrece algunas propiedades estadísticas sobre la región de interés o las fibras activas. Estos parámetros son: anisotropía fraccional y relativa, desviación media, coeficientes geométricos del tensor, elementos del tensor y autovalores. Estas magnitudes se calculan en promedio, y los resultados pueden almacenarse en un fichero.

La tercera zona de la interfaz de Saturn es la de visualización. La herramienta ofrece tres modos de visualización: cuatro visores 2D funcionando en paralelo (4x2D), tres visores 2D con uno 3D (3+1), o un visor 3D de mayor tamaño, que ocupa el espacio de los cuatro visores anteriores. Los visores ofrecen las opciones habituales, como acercar y alejar la imagen, rotarla, moverla, etc., seleccionar el plano que se desea visualizar, mostrar las propiedades de la imagen, o agrandar el visor.

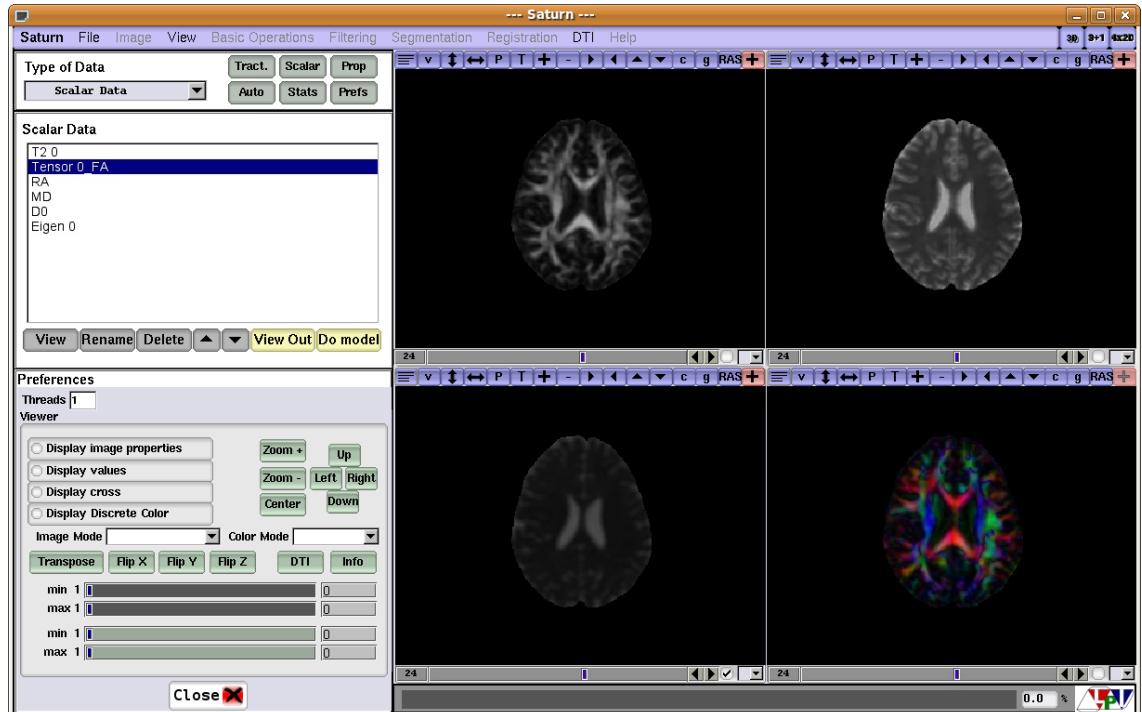


Figura 5.1. Interfaz de usuario de Saturn

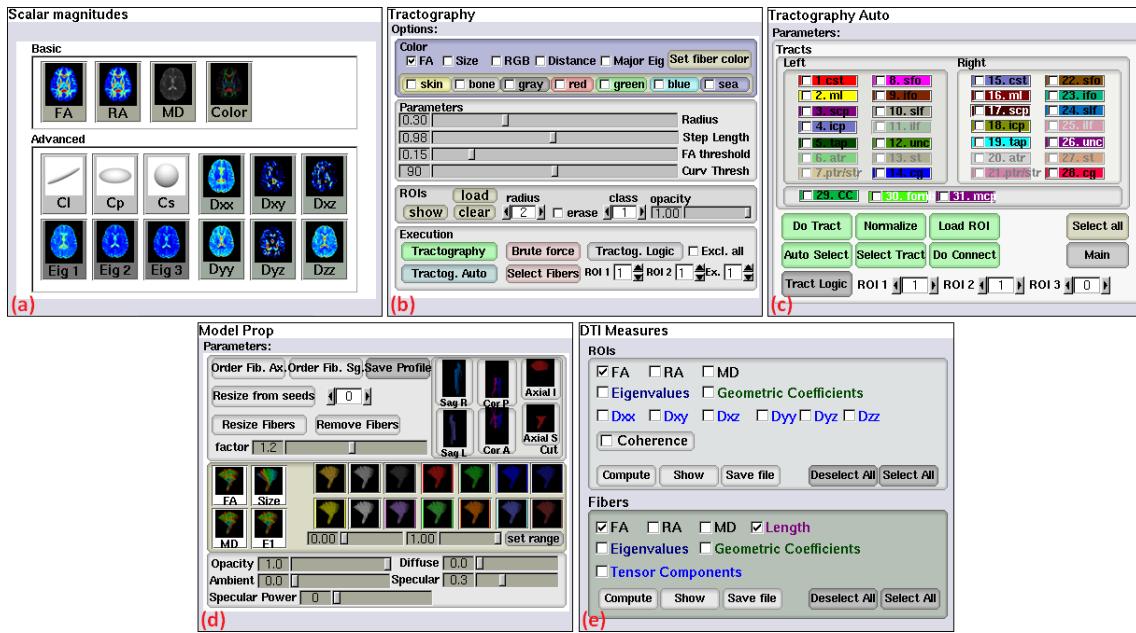


Figura 5.2. Paneles de configuración de Saturn: (a) Magnitudes escalares, (b) Tractografía, (c) Tractografía Auto, (d) Editar Fibras, (e) Medidas

El visualizador 3D se muestra en la Figura 5.3. Con el visualizador 3D se pueden ver los cortes de la imagen DT-MRI proyectados sobre tres planos perpendiculares, lo que permite estudiar con más detalle las imágenes. Se pueden elegir el número de plano en cada dirección, o no mostrar alguno de los tres planos. El visor 3D se va a utilizar para otras funciones, como la visualización de tractos de fibra obtenidos por tractograffía (Figura 5.4) o la visualización de glifos, que se implementa con este trabajo.

La interfaz presenta además diversos menús en su parte superior que permiten al usuario cargar o guardar datos, cambiar el modo de vista, o acceder a los paneles de configuración. Dado que Saturn está en desarrollo, algunos de los menús no están disponibles, a la espera de la implementación de sus funciones.

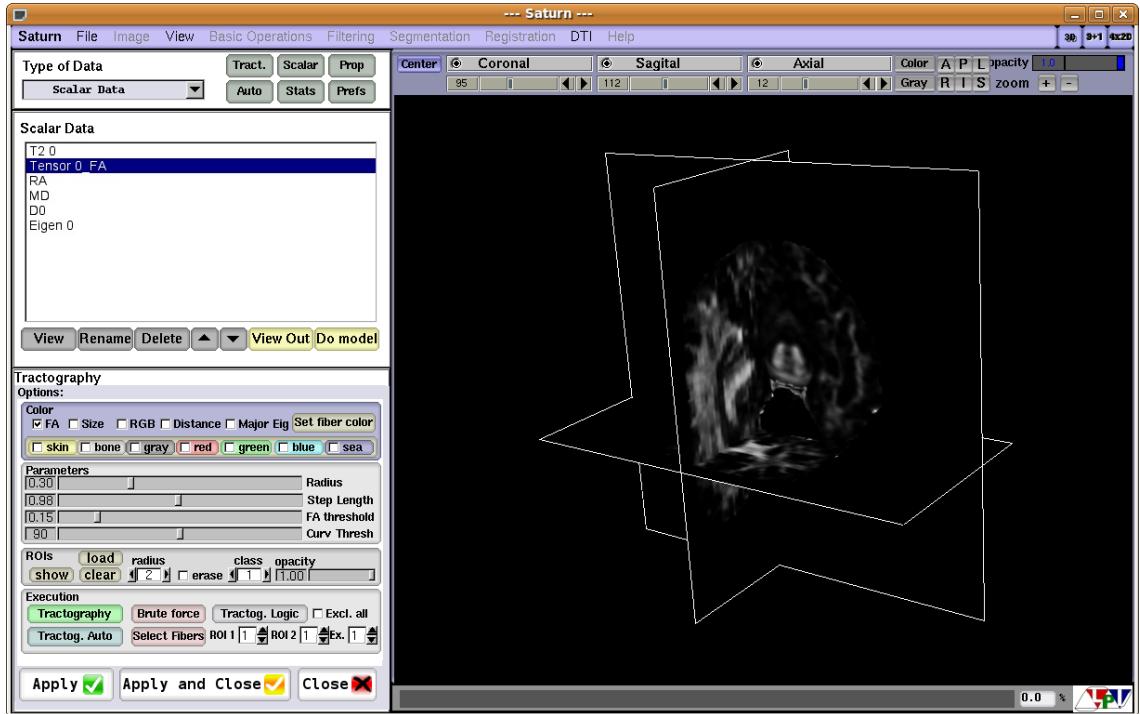


Figura 5.3. Visualización 3D en Saturn

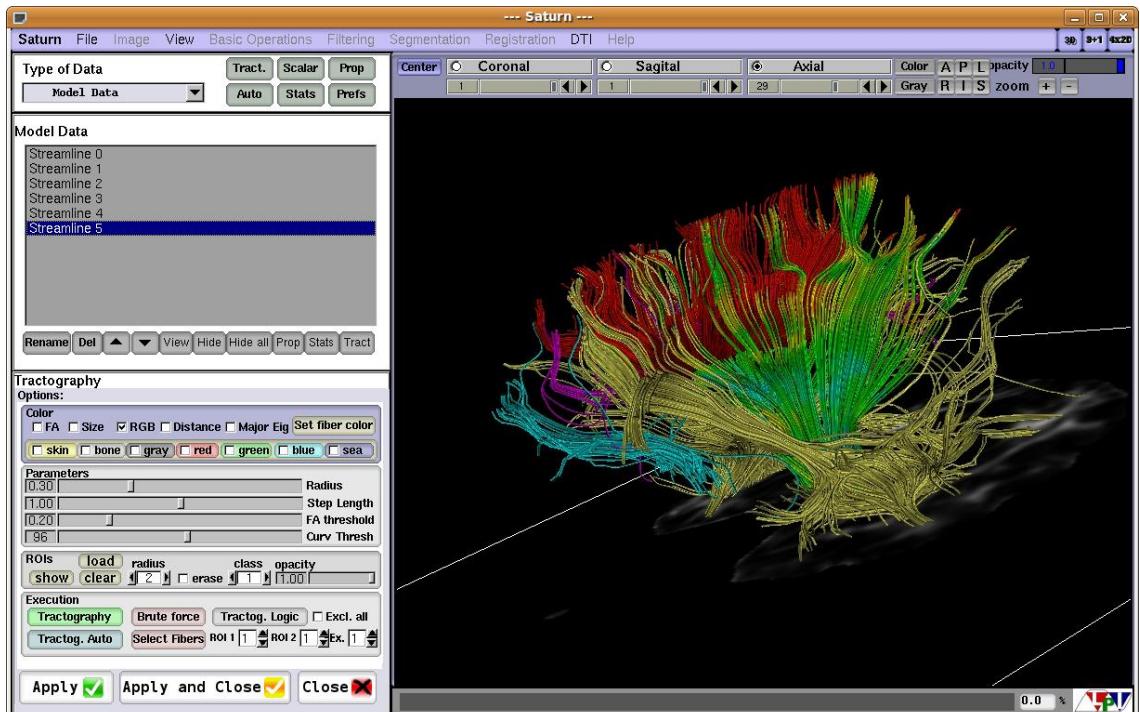


Figura 5.4. Tractografía en Saturn

5.3 Código de Saturn

5.3.1 Clase UsimagToolBase

La superclase UsimagToolBase es la clase básica de la aplicación Saturn, y se utiliza como referencia en el resto del código. En su fichero de cabecera se definen los tipos de datos que se van a emplear, se declaran algunas variables comunes y se declaran varios métodos básicos.

La definición de tipos utiliza plantillas (o templates) de ITK para definir los tipos de datos que se van a utilizar con más frecuencia. Entre ellos se puede encontrar el tipo de información que define cada píxel de un conjunto de datos o el tipo de imagen que se utiliza (tipo de píxel y dimensión). También aparecen clases para la entrada y salida de datos (readers y writers), visores de imágenes (viewers), filtros, iteradores ITK o conectores ITK/VTK. Para este proyecto interesan especialmente tres definiciones, correspondientes a los datos de tensor de difusión:

```
typedef itk::DTITensor<float> TensorPixelType;
typedef itk::Image<TensorPixelType, Dimension> TensorImageType;
typedef VolumesContainer<DataTensorElementType> VectorOfTensorDataType;
```

La primera de ellas (TensorPixelType) define el tipo de datos que se va a emplear. DTITensor es una clase creada específicamente para Saturn, que almacena el tensor y permite realizar diversas operaciones sobre él. El tensor va a contener datos no enteros (tipo float). La segunda línea define el tipo de imagen (TensorImageType), que está formada por los píxeles anteriores, y tiene 3 dimensiones (anteriormente se especifica Dimension=3). La tercera línea especifica un tipo de dato que contiene la imagen tensorial y algunos métodos y datos adicionales. Las clases VolumesContainer y DataTensorElementType también son específicas de Saturn y se explicarán más adelante.

En segundo lugar, UsimagToolBase declara una serie de variables que se van a utilizar con frecuencia y por parte de un gran número de clases a lo largo del código de la aplicación. Entre ellas, las más importantes son los vectores de datos, las imágenes y los visores. Aparece aquí la declaración del vector que va a contener los datos de difusión. El código de este proyecto utiliza el siguiente vector para almacenar imágenes tensoriales:

```
VectorOfTensorDataType m_VectorTensorData;
```

Por último encontramos varias definiciones de métodos virtuales, y un constructor que inicializa las variables.

5.3.2 Clase UsimagToolGUI

La clase UsimagToolGUI hereda de UsimagToolBase, y define gran parte de la interfaz de usuario de la aplicación. La Figura 5.5 muestra la interfaz definida en UsimagToolGUI, tal y como la muestra con la herramienta Fluid.

El panel situado en la parte superior izquierda de la interfaz muestra una lista con los conjuntos de datos abiertos por la aplicación, así como accesos directos a otros paneles de la aplicación. En la parte inferior izquierda aparece un espacio, que ocupan algunos de los paneles específicos accesibles desde el menú. De estos paneles, sólo el panel Preferencias se encuentra implementado en UsimagToolGUI. Los demás paneles se definen en TensorGUI, o en otras clases.

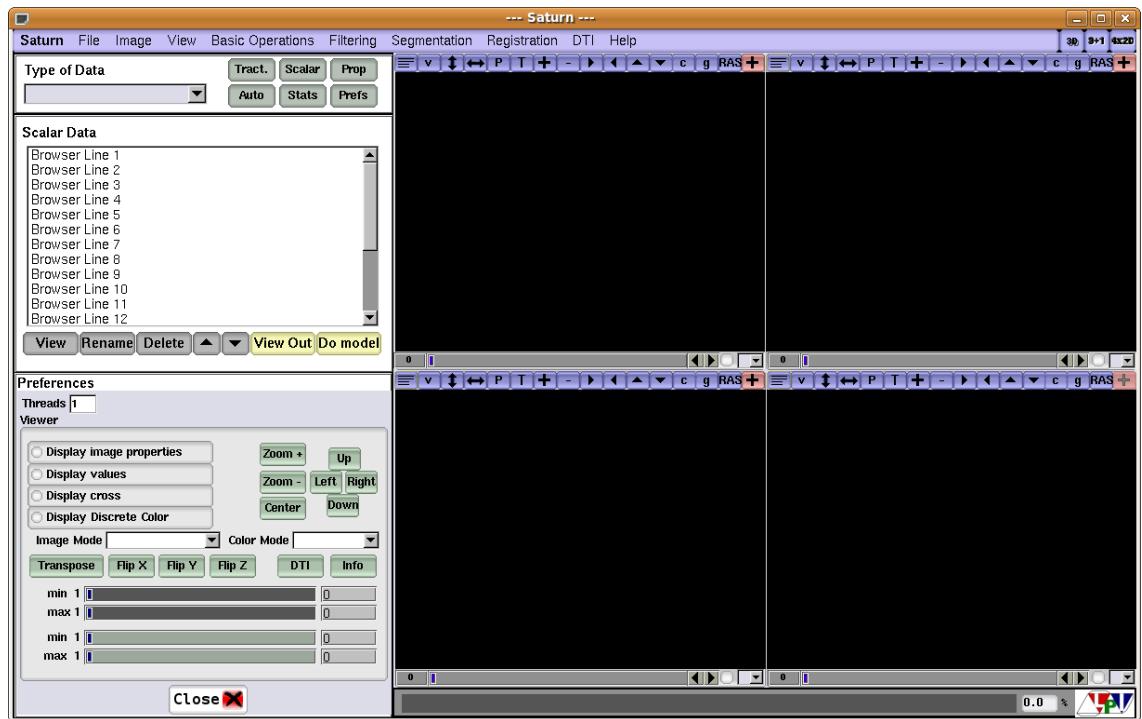


Figura 5.5. Visualización con Fluid de la interfaz definida en UsimagToolGUI

5.3.3 Clase UsimagToolConsole

La clase UsimagToolConsole hereda de UsimagToolGUI, e implementa la lógica asociada a dicha interfaz. Así, la clase UsimagToolGUI define los elementos que componen la interfaz de usuario, así como su colocación y comportamiento, mientras que UsimagToolConsole implementa los callbacks de los elementos

activos de la interfaz, como botones, barras de desplazamiento, etc. Estos métodos realizan las tareas de cargar o guardar los datos, dibujar o eliminar imágenes, o modificar distintas propiedades de las imágenes. La aplicación soporta una gran variedad de formatos de entrada, como ficheros VTK o NRRD, imágenes JPEG, PNG o TIFF, imágenes DICOM, etc.

5.3.4 Clase TensorGUI

La clase TensorGUI define la interfaz de usuario de los diferentes paneles que se sitúan en la parte inferior izquierda de la aplicación. Existen actualmente seis paneles, que ya han sido explicados en una sección anterior: Preferences, Scalars, Tractography, Tractography Auto, Fibers Edit y Measures, aunque el panel Preferences forma parte de UsimagToolGUI. Este proyecto incluye varios paneles para la visualización de glifos. La definición de estos nuevos paneles, que se explicarán más adelante, se encuentran también en TensorGUI.

5.3.5 Clase TensorConsole

La clase TensorConsole implementa la lógica asociada la interfaz TensorGUI, con métodos para las distintas funciones que ofrece la interfaz. Esta clase es importante, porque va a incluir varios métodos del código de este proyecto. TensorConsole hereda de TensorGUI.

En primer lugar, el fichero de cabeceras de TensorConsole define de nuevo los tipos de dato y las variables que ya aparecían en UsimagToolBase. Esto es necesario debido a que TensorConsole no hereda, directa ni indirectamente, de UsimagToolBase. Sin embargo, al iniciar la aplicación, estas variables son inicializadas para que tomen los mismos valores en las dos clases. Además se incluyen otras variables de clase necesarias para el correcto funcionamiento de los métodos.

Entre los métodos, una gran parte de ellos están asociados a las técnicas de tractografía. Aparecen así métodos para calcular la trayectoria de las fibras, para crear las streamlines con las que se representan, o diversos métodos para realizar cálculos sobre ellas. Interesa especialmente para este proyecto el método RungeKuttaTractography(), que calcula los puntos que recorre el tracto. Estos puntos se utilizan en este proyecto para representar en ellos los glifos correspondientes, y enriquecer así la visualización de la tractografía.

5.3.6 Clase DTITensor

La clase DTITensor es el tipo de dato fundamental que se va a utilizar en las imágenes tensoriales de difusión. Se trata de un array de seis elementos que almacena cada uno de los seis elementos independientes del tensor (el tensor de difusión es simétrico). De este modo, si a, b, c, d, e y f son los seis elementos almacenados en el array, el tensor tiene la siguiente forma:

$$\begin{pmatrix} a & b & c \\ b & d & e \\ c & e & f \end{pmatrix}$$

En la clase se define una variedad de métodos para trabajar con los tensores. En primer lugar, varios métodos para obtener los autovalores y autovectores del tensor. Los autovalores y autovectores se devuelven con los tipos de dato EigenValuesArrayType y EigenVectorsMatrixType, una array de tres elementos y una matriz 3x3 respectivamente. En segundo lugar aparecen diversos métodos para obtener características del tensor. Por ejemplo, el determinante del tensor, la anisotropía fraccional o relativa, o los coeficientes geométricos lineal, planar y esférico. Por último, la clase ofrece también métodos para obtener la imagen del tensor en el espacio log-euclídeo y viceversa. Estos métodos se usan para la interpolación.

5.3.7 Clase DataTensorElementType

La clase DataTensorElementType es un contenedor simple para las imágenes tensoriales con las que trabaja Saturn, con sólo tres atributos y tres métodos. Sus atributos son el nombre asociado a la imagen, un identificador y la propia imagen. El nombre asociado a la imagen es por defecto el nombre del fichero del que se han obtenido los datos, pero puede ser modificado por el usuario. El identificador es un entero asignado automáticamente, y la imagen es un dato del tipo TensorImageType, del que ya se ha hablado anteriormente. La clase contiene además tres métodos para crear, copiar y eliminar la imagen.

La clase DataTensorElementType está definida en el fichero VolumesContainer.h, donde también aparecen otras clases para diferentes tipos de imágenes pero con una estructura similar, como DataElementType, DataDWIElementType o DataModelElementType.

5.3.8 Clase VolumesContainer

VolumesContainer es la clase que se utiliza para almacenar todas las imágenes de un mismo tipo que se encuentran cargadas a la vez en Saturn. Se trata de una vector que contiene elementos del tipo Data*****ElementType (DataTensorElementType en el caso de imágenes tensoriales). La interfaz ofrece los métodos habituales para trabajar con un vector, como añadir y eliminar elementos, o copiar los datos del vector. Además, VolumesContainer permite registrar dos tipos de elementos de lista de la interfaz: Fl_Browser (lista de elementos con desplazamiento) y Fl_Coice (lista desplegable). De este modo, las listas se actualizan automáticamente cuando un nuevo elemento se añade al vector, es decir, cuando Saturn carga una nueva imagen.

5.3.9 Clase Viewer3D

La clase Viewer3D implementa el visor 3D de Saturn. En su interfaz tiene especial importancia la visualización tridimensional de planos 2D y la tractografía. La visualización de planos bidimensionales en el espacio permite mostrar al mismo tiempo tres cortes perpendiculares, y observar el resultado desde diferentes ángulos y posiciones, lo que mejora el estudio de la imagen. En cuanto a la tractografía, Viewer3D recibe la geometría y los escalares de los tractos, para dibujar y colorear los tractos de forma transparente para el usuario, que no necesita preocuparse del último tramo del pipeline de VTK (mapper, actor, luces, etc.).

La clase Viewer3D implementa otros métodos para modificar la luz, la opacidad o el color de los distintos actores, y un método sencillo para mostrar imágenes del tipo vtkPolyData.

5.3.10 Resto de código

En esta sección se han explicado las clases más relevantes para este proyecto. Sin embargo, el código de Saturn consta de varias decenas de clases y ficheros, contenidos en un directorio principal y una serie de carpetas específicas. El directorio principal incluye las ya mencionadas UsimagToolBase, UsimagToolGUI, UsimagToolConsole, TensorGUI, TensorConsole y VolumesContainer. Aparecen además dos clases para la interconexión entre ITK y VTK (ImageToVTKImageFilter y vtkITKUtility), las clases

GenericImageToImageFilter y geodesicPath3D, y varias interfaces de usuario más (BasicOpGUI, FilteringGUI, ImageViewerGUI, SegmentationGUI).

La clase DTITensor, explicada anteriormente se encuentra en la carpeta *tensor*, que contiene una serie de utilidades para imágenes tensoriales, como filtros ITK, transformaciones, lectores de ficheros, etc. La clase Viewer3D se encuentra en la carpeta 3DViewer, que incluye otras clases de interconexión entre VTK y FLTK. Existe un total de 12 directorios de código más, además de uno de imágenes: ASR, Demons3D, DPAD, FltkImageViewer, Images, knn-1canal, Kretz, MyFltkImageViewer, registrado_tristan, SRAD, VtkFltk, vtkMarcacionElipse y wiener. El código de Saturn está aún en desarrollo, por lo que es posible que en un futuro algunos de estos directorios desaparezcan, cambien, o se creen otros nuevos.

5.4 Otras interfaces de visualización

Como ya se ha mencionado, existen otras interfaces gráficas para el manejo de campos tensoriales de imagen médica [51]. Entre ellas se encuentran 3D Slicer, MedInria, BioTensor, DtStudio y otras. En esta sección se va a hacer hincapié en las dos primeras, por ser las más avanzadas.

3D Slicer [52] es un software orientado a la imagen médica, en particular a la planificación preoperatoria, la cirugía asistida por imagen y la visualización diagnóstica. Ofrece diferentes modalidades de visualización, así como diferentes técnicas de procesado de imagen, como filtrado, segmentación y registrado. Sin embargo, 3D Slicer no ofrece garantías de precisión clínica o fiabilidad para la investigación.

La interfaz de la aplicación consta de dos partes: en la zona izquierda aparece un panel con los controles de visualización, información sobre el volumen, etc., mientras que a la derecha se encuentra el área de visualización. Además, en la parte superior existe una barra de herramientas que permite elegir entre una variedad de módulos, a cada uno de los cuales va asociado un panel en la parte izquierda de la interfaz.

Se ofrecen tres métodos de visualización para volúmenes DT-MRI. En primer lugar, volúmenes de índices de anisotropía o invariantes tensoriales, representados en cortes 2D ortogonales. En segundo lugar, un método de visualización de tractos de fibras, y en tercer lugar, visualización con glifos.

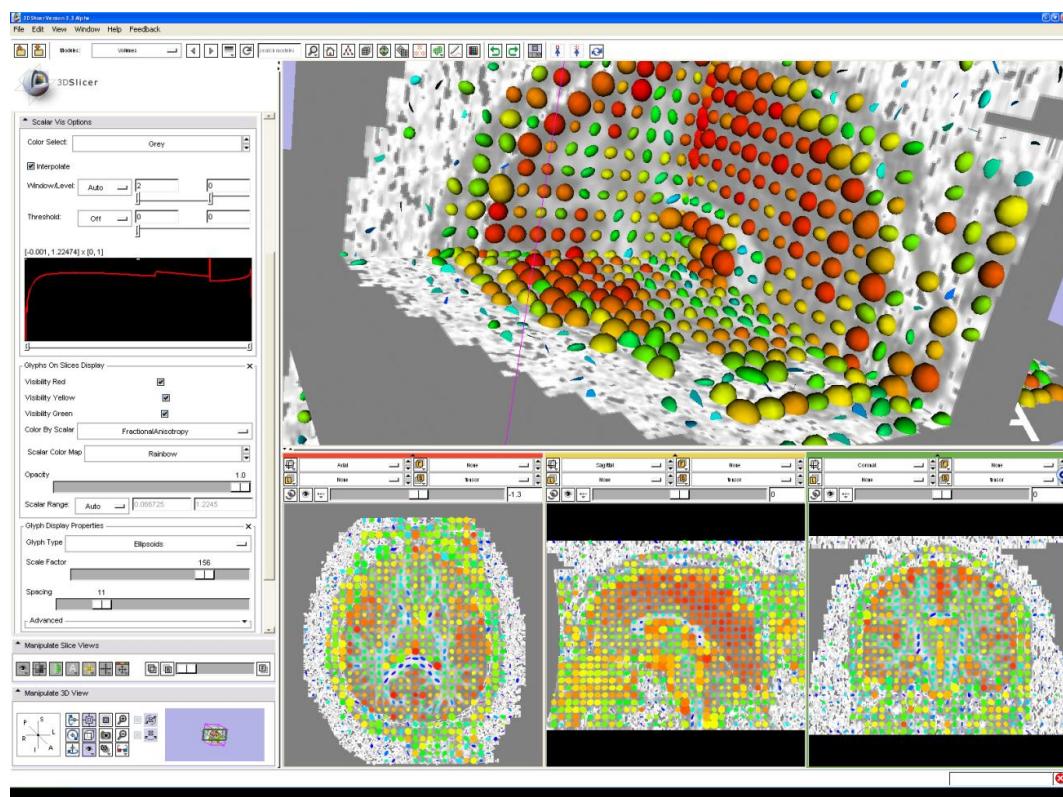


Figura 5.6. Visualización de glifos en 3D Slicer [52]

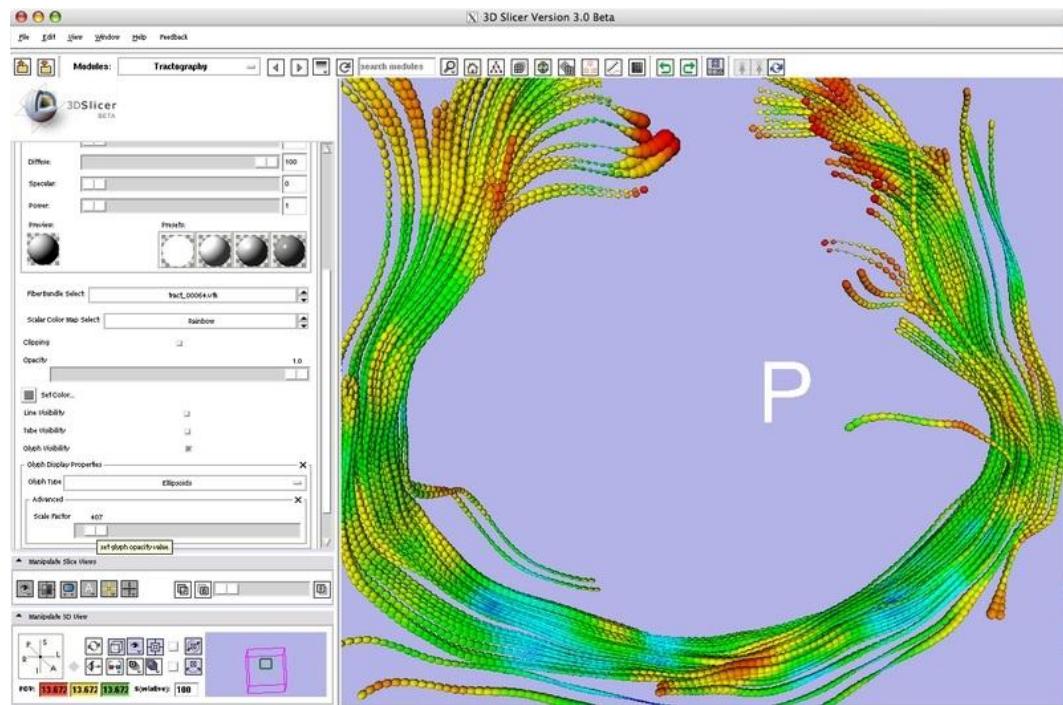


Figura 5.7. Visualización de tractos mediante glifos en 3D Slicer [52]

La interfaz para la visualización de glifos en 3D Slicer puede verse en la Figura 5.6. Esta opción está disponible para visualización de planos (Figura 5.6) y tractografía (Figura 5.7). En ambos casos, el usuario puede indicar el tipo de glifo, el escalar utilizado para colorear los glifos, el factor de escala, la opacidad o el número de glifos que se muestran.

3D Slicer utiliza Tcl/Tk en la interfaz gráfica, VTK para la visualización e ITK para el procesado, y tiene una arquitectura modular. El software es de código abierto, y fue creado por el Laboratorio de Inteligencia Artificial del MIT (MIT AI Laboratory) y por el Brigham and Women's Hospital de Harvard. Su última versión estable es la 3.6, y la versión 4 ya está en desarrollo.

MedINRIA [53][54], por su parte, es un software desarrollado originalmente para el procesado y visualización de datos DT-MRI, y en la actualidad integra además módulos de otros tipos. Tiene varias semejanzas con Saturn y 3D Slicer, como la arquitectura modular y el uso de ITK y VTK. Una de las diferencias más importantes es que, a pesar de que se uso es gratuito, su redistribución no lo es, y su código no ha sido publicado.

El software cuenta con dos módulos específicos para DT-MRI, DTI-Track y Tensor Viewer, y utiliza la métrica log-euclídea, desarrollada por el mismo equipo de investigación. La Figura 5.8 muestra la interfaz de MedINRIA para la visualización de glifos. Esta posibilidad sólo está disponible para cortes 2D, y no para tractografía. El usuario puede escoger el tipo de glifo, la tasa de muestreo (número de glifos que se muestran en relación con el número de tensores), la resolución y el tamaño de los glifos, y los planos sagital, coronal y axial que se muestran.

MedINRIA fue creado por Pierre Fillard y Nicolas Toussaint, como parte del proyecto Asclepios en el centro del INRIA (Institut National de Recherche en Informatique et en Automatique) en Sophia Antipolis, Francia.

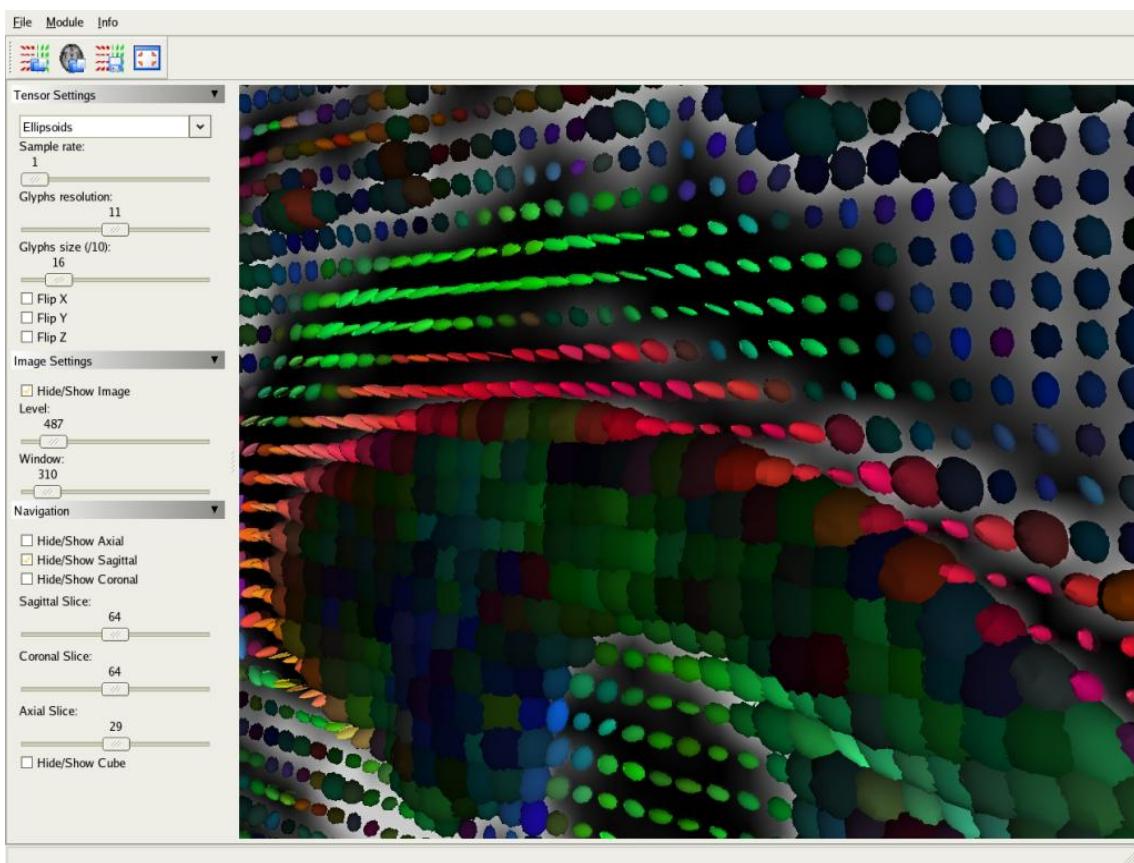


Figura 5.8. Interfaz de visualización de glifos en MedINRIA [54]

Capítulo 6

Desarrollo de una interfaz de visualización para DTI

En este capítulo se explica extensamente la interfaz de visualización desarrollada en este proyecto. Se explican los objetivos, la estructura, el funcionamiento, y las aplicaciones de la nueva interfaz, y se muestran algunos ejemplos del resultado conseguido.

6.1 Introducción

En este capítulo se presenta la interfaz desarrollada en este proyecto para la visualización de glifos a partir de imágenes de tensor de difusión. La interfaz tiene el objetivo de añadir esta nueva característica a Saturn con un módulo robusto, flexible y de uso sencillo.

El capítulo se estructura como sigue: en primer lugar, se presenta la clase *vtkTensorGlyphDTI*, encargada de generar los glifos para su visualización. Esta clase es el elemento central de la interfaz, y por ello se estudia y explica de forma minuciosa. A continuación, se explican las distintas modificaciones que se han realizado sobre el código ya existente en Saturn, esto es, los nuevos métodos y variables que se han introducido para integrar este módulo. En la siguiente sección se muestra el aspecto visual de la interfaz, así como sus partes y su forma de uso. Por último, se muestran algunas pruebas de rendimiento del nuevo módulo y diversos ejemplos de cada una de las características implementadas.

6.2 Clase *vtkTensorGlyphDTI*

La clase *vtkTensorGlyphDTI* es la encargada de convertir un volumen ITK cargado en Saturn en un conjunto de glifos del tipo *vtkPolyData* que pueda ser visualizado. La clase se ha desarrollado con el objetivo de ser parte de Saturn, y ofrecer una visualización de glifos flexible, configurable e integrada.

De este modo, los objetivos que se persiguen con el desarrollo de esta clase son los siguientes:

- Creación de glifos a partir de un campo tensorial que recibe como entrada. Los glifos se generarán como se explica en la literatura, esto es, los autovalores determinan el tamaño del glifo en la dirección de cada eje, los autovectores determinan su orientación, y la posición del glifo viene determinada por la posición que ocupa el tensor dentro de la imagen.
- La visualización debe integrarse con la de Saturn, por lo que se debe poder mostrar un solo plano (un solo corte) de todo el volumen.
- Uso de las diversas geometrías de glifo que aparecen en la literatura.
- Posibilidad de colorear los glifos según diversos parámetros.

- Posibilidad de discriminar los glifos que se muestran, en función de diferentes coeficientes del tensor. De este modo se pueden eliminar de la visualización áreas irrelevantes y mejorar la eficiencia.
- Posibilidad de visualizar sólo una zona determinada de la imagen. Esto reduce el coste computacional y el consumo de memoria asociado a la renderización de un número elevado de glifos. Los recursos que se ahorran permiten mejorar la eficiencia o dotar de un mayor nivel de detalle a los glifos de la zona de interés.
- Flexibilidad en la visualización, de forma que se puedan visualizar glifos en puntos arbitrarios. Esto puede permitir la visualización de glifos en tractografía, la visualización en planos oblicuos, el empaquetado de glifos, etc.
- Posibilidad de variar la resolución de los glifos. Esto permite adaptar la calidad de la visualización a las prestaciones del equipo en que se ejecuta Saturn.
- Flexibilidad: debe ser posible añadir nuevas opciones y funciones de una forma rápida y sencilla.

La generación de glifos a partir de un campo tensorial es una tarea que ya realiza la clase *vtkTensorGlyph*, que forma parte del paquete VTK. Sin embargo, el uso de *vtkTensorGlyph* obliga a una conversión entre los datos utilizados en Saturn, del tipo *DTITensor* y de *Image* de ITK, a los tipos de VTK utilizados por *vtkTensorGlyph*. Esto supone una pérdida de eficiencia en una tarea, la de la generación de glifos, que conlleva de por sí una carga computacional importante. La creación de la nueva clase *vtkTensorGlyphDTI* permite solucionar este problema, trabajando directamente sobre los datos de Saturn.

Más allá de los tipos de dato, la creación de una nueva clase ofrece un control total sobre el código, que permite adaptarlo a las necesidades de Saturn. Así, operaciones como la selección de plano, el coloreado de los glifos o la normalización de los autovalores se integran en el código de la clase de forma transparente para el desarrollador. La clase *vtkTensorGlyph*, sin embargo, no soporta estas operaciones, y el desarrollador debe realizarlas de forma manual, lo que aumenta la complejidad del código.

6.2.1 Tipos de dato

El punto de partida es el propio código de Saturn, que marca el tipo de datos de entrada, así como algunos de los tipos de datos que van a utilizarse dentro del código. Los tipos de dato definidos en *vtkTensorGlyphDTI* son:

- *RealType*: especifica el tipo de elementos que contienen los tensores (float). Este tipo de dato es necesario al operar con tensores del tipo *DTITensor*.

```
typedef itk::NumericTraits<float>::RealType
RealType;
```

NumericTraits es una plantilla para definir tipos de dato numéricos en ITK.

- *TensorPixelType*: tipo de píxel de la imagen. Es del tipo *DTITensor*, lo que simplifica parte del proceso, ya que implementa métodos para calcular los autovalores y autovectores del tensor, los coeficientes geométricos, los coeficientes de anisotropía y las transiciones al espacio log-euclídeo.

```
typedef itk::DTITensor<float>      TensorPixelType;
```

- *TensorImageType*: tipo de imagen tensorial, con tres dimensiones (en el código se define *Dimension = 3*) y *DTITensor* como tipo de píxel.

```
typedef itk::Image<TensorPixelType, Dimension> TensorImageType;
```

Dentro de la clase *Image* de ITK, varios métodos se utilizan en el código de *vtkTensorGlyphDTI*. El método *GetPixel()* permite obtener un píxel de la imagen, en este caso un tensor, a partir de un índice que especifique la posición del píxel en la imagen. Otros métodos permiten obtener la posición en el espacio del origen de coordenadas de la imagen, el espaciado entre píxeles, o la correspondencia entre un píxel de la imagen y su posición en el espacio.

- *EigenValuesArrayType* y *EigenVectorsMatrixType*: se trata de un vector de tres elementos y de un array 3x3, respectivamente. Se pasan como parámetro a los métodos para el cálculo de autovalores y autovectores del tensor de *DTITensor*. Sus elementos pueden ser accedidos como en un array común, sin métodos especiales.

```
typedef itk::FixedArray<RealType,3>      EigenValuesArrayType;
typedef itk::Matrix<RealType,3,3>  EigenVectorsMatrixType;
```

6.2.2 Variables de clase

Estas son las variables definidas a nivel de clase:

- *input*: imagen tensorial de entrada. De ella se extraen los datos.

```
TensorImageType::Pointer input;
```

- *source*: variable interna de la clase. Contiene un ejemplar del glifo primitivo utilizado (esfera, cubo, supercuádrica).

```
vtkPolyData *source;
```

- *inputPoints* (por defecto, NULL): contiene los puntos donde se quiere que se dibujen glifos. Si un punto no pertenece a la imagen, el tensor se interpola en ese punto. Se muestran todos los glifos asociados a estos planos, sin tener en cuenta el filtro de discriminación de glifos ni el crop (recorte) de planos.

```
vtkPoints *inputPoints;
```

- *GlyphType* (por defecto, *ELLIPSOID*): indica el tipo de glifo a mostrar. Las posibilidades son elipsoide, cuboide y supercuádrica.

```
int GlyphType;
```

```
enum {
    ELLIPSOID,
    CUBOID,
    SUPERQUADRIC
};
```

- *Scaling*, *ScaleFactor* (por defecto, 1 y 1.0): si *Scaling* es distinto de cero, aplica a cada glifo el factor de escala indicado por *ScaleFactor*. Si *Scaling* es cero, no hace nada.

```
int      Scaling;
double  ScaleFactor;
```

- *ColorMode* (por defecto COLOR_BY_FA): indica el parámetro que se debe utilizar para colorear los glifos. Puede ser la anisotropía fraccional, la relativa o el coeficiente geométrico lineal.

```
int  ColorMode;
```

```
enum {
    COLOR_BY_FA,
    COLOR_BY_RA,
    COLOR_BY_CL
};
```

- *Bounds* (por defecto, [0,0,0,0,0,0]): contiene los índices de recorte de planos. Se trata de un array de seis elementos, con los índices mínimo y máximo en que se mostrarán glifos para cada plano. Estos seis elementos son, por orden: índice X mínimo, índice X máximo, Y mínimo, Y máximo, Z mínimo y Z máximo. No se visualiza ningún tensor con un índice menor que alguno de los mínimos o mayor que uno de los máximos. Los elementos con índice en el límite sí que se muestran, así que para mostrar un sólo plano se debe introducir el mismo valor en los índices mínimo y máximo para dicho plano (por ejemplo, para el mostrar el corte número 100 en la dirección Z, Z mínimo y Z máximo deben valer 100).

```
int Bounds[6];
```

- *PhiResolution*, *ThetaResolution* (por defecto, 8 y 8): resolución de los glifos en las dos direcciones angulares.

```
int PhiResolution;
int ThetaResolution;
```

- *Gamma* (por defecto, 3.0): parámetro gamma de los glifos supercuádricos. Gamma determina la forma, más o menos redondeada, de la supercuádrica.

```
double Gamma;
```

- *FilterMode* (por defecto, FILTER_BY_FA): indica el parámetro que se tendrá en cuenta para la discriminación de glifos. Puede ser la anisotropía fraccional, o los coeficientes geométricos lineal o esférico.

```
int FilterMode;
```

```
enum {
    FILTER_BY_FA,
    FILTER_BY_CL,
    FILTER_BY_CS
};
```

- *FilterThreshold* (por defecto, 0.0): indica el valor que se toma como límite para la discriminación de glifos. Si se filtra por el coeficiente esférico, este parámetro indica el valor máximo de este coeficiente que debe tener un tensor para ser visualizado. En los otros dos casos indica el valor mínimo.

```
double FilterThreshold;
```

6.2.3 Métodos

La lógica de la clase cuenta con tres métodos, además de los *getters* y *setters* para los atributos. El método principal es *GetOutput()*. Esta función toma los datos de entrada y los parámetros, y devuelve los glifos resultantes, en forma de un objeto del tipo *vtkPolyData*. Los métodos *interpolacionLineal()* e *interpolacionLogEuclidea()* se utilizan para interpolar los ocho glifos adyacentes (los dos más cercanos en cada dimensión del espacio), mediante uno u otro método.

Método GetOutput()

El método *GetOutput()* se encarga de generar la representación de los tensores. Esta función recibe a su entrada un puntero a un objeto de tipo *vtkPolyData*, *output*, que debe ser creado con anterioridad. Este objeto va a contener, después de la ejecución, los glifos creados.

A continuación se detallan los pasos que se llevan a cabo en *GetOutput()* para generar los glifos:

1. Inicialización de las variables que se van a utilizar: matrices, celdas, puntos, etc.
2. Elección y generación de la fuente. Se crea la primitiva geométrica a partir de la cual se dibujan los glifos. La geometría de la fuente se va a copiar y modificar en cada punto para producir el glifo. Esta primitiva se crea en el origen de coordenadas, y tiene un tamaño unitario. Se utilizan las fuentes disponibles en VTK: *vtkSphereSource*, *vtkCubeSource*, *vtkSuperquadricSource*.
3. Se utiliza un bucle para recorrer los píxeles de la imagen como una cuadrícula. El bucle sólo pasa por los píxeles que se encuentran dentro de los límites de recorte (crop) de planos, especificados por el atributo *Bounds*. El bucle da los siguientes pasos:

- a. Obtener el índice del píxel actual.
- b. Extraer de la imagen el tensor asociado a dicho índice.
- c. Calcular los autovalores y autovectores del tensor. Para ello se hace uso del método *ComputeEigenSystem()* contenido en la clase *DTITensor*. Los autovalores obtenidos están ordenados, con el mayor autovalor primero. Los autovectores están normalizados para tener norma uno.
- d. Si el primer autovalor es cero, se continúa a la siguiente iteración del bucle. Esto evita realizar el proceso para tensores nulos, que de cualquier modo no se iban a mostrar. Así se reduce la carga computacional y el consumo de memoria.
- e. Se calculan los coeficientes geométricos del tensor. Estos coeficientes se van a utilizar para discriminar los tensores, y para determinar los parámetros de las supercuádricas, en caso de que se utilice este tipo de glifo.
- f. Discriminación de tensores: se comprueba el valor de la anisotropía fraccional o de los coeficientes geométricos, así como el valor límite recibido como parámetro. Si el tensor no se encuentra dentro de los límites, se continúa a la siguiente iteración del bucle y el glifo no se genera.
- g. Traslación del glifo. La posición del glifo se determina a partir del índice del píxel dentro de la imagen, del espaciado de la imagen y de la posición del origen de coordenadas. El glifo se traslada a dicho punto mediante el método *Translate()* de *vtkTransform*. La clase *vtkTransform* se va a utilizar para las transformaciones geométricas del glifo: traslación, rotación y escalado.
- h. Rotación del glifo. Para ello se utiliza una matriz de rotación 3x3 con los autovectores del tensor. Al tercer autovector se le aplica un factor signo, que previene una orientación inadecuada de las normales del glifo (ver nota).
- i. Se calcula el factor de normalización. Para visualizar adecuadamente los glifos, los autovalores se normalizan para que el mayor de ellos valga uno. Así, el factor de normalización es el inverso del mayor autovalor.

- j. Se calcula el factor de escala global: si el escalado está activado (variable *Scaling*), se calcula como el producto del factor de normalización por el factor de escala. Si no, el factor de escala global es simplemente el factor de normalización.
 - k. Escalado del glifo. El factor de escala en cada dirección se calcula como el producto del factor de escala global por el autovalor asociado a cada dirección. De esta forma, los autovalores determinan la forma del glifo y el factor de escala su tamaño.
 - l. Si la primitiva del glifo es una supercuádrica, hay que calcular los parámetros α y β de la misma. Estos parámetros se calculan a partir de los coeficientes geométricos lineal y planar y el parámetro gamma. Una vez calculados, se introducen en la fuente original.
 - m. Se aplica la transformación a los puntos, obteniendo los puntos del nuevo glifo. Se utiliza el método *TransformPoints()*, que añade estos nuevos puntos a *newPts*, el conjunto de puntos de todos los glifos del volumen. Lo mismo se hace con las normales.
 - n. Se calcula el escalar que se va a utilizar para colorear los glifos, y se añade a un array llamado *newScalars*.
 - o. Un contador lleva la cuenta del número de glifos dibujados.
4. Se comprueba si hay puntos de entrada donde mostrar puntos (variable *inputPoints*), es decir, si el usuario de la clase desea mostrar glifos en puntos arbitrarios. Si es así, un bucle recorre todos los puntos de entrada. El bucle es parecido al anterior, con dos diferencias (pasos a y b, y paso h). La primera es la obtención del tensor. Se calcula la posición del punto de entrada y se interpolan los tensores de su alrededor, con el método *interpolacionLineal()* o *interpolacionLogEuclidea()*. La segunda diferencia es la traslación del glifo. En este caso no es necesario calcular la posición, sino que es simplemente el punto de entrada para cada iteración del bucle. El contador de glifos no se pone a cero en este bucle, sino que continúa la cuenta donde terminó en el primer bucle.
 5. Se reserva el espacio de memoria para las celdas de todos los glifos. El número de celdas de cada tipo (vértice, línea, polígono, franjas) es el producto del número de celdas de la fuente por el número de glifos.

6. Se insertan en el volumen de salida (*output*) las celdas de todos los glifos. Las celdas se pueden insertar al final, ya que no varían de un glifo a otro, sólo dependen de la forma y la resolución de la fuente. De hecho, en principio también se podrían insertar al principio de la función, pero en la práctica esto no es posible ya que en el principio no se conoce el número final de glifos.
7. Se añaden al volumen de salida los nuevos puntos, escalares y normales creados.
8. Se eliminan de la memoria los elementos creados en la ejecución

Nota

El factor signo aparece cuando al detectar anomalías en la iluminación de algunos glifos. En la Figura 6.1.a se muestra muestra dicho problema: deberían aparecer glifos en la mayoría de las zonas oscuras. Si se acerca aún más la imagen, se observa que los glifos están de hecho dibujados, pero su superficie está oscura. El problema se debe a una inadecuada orientación de las normales: su dirección es correcta pero su sentido no lo es.

Se ha detectado que, en los casos en que la visualización es correcta, el producto vectorial del primer autovector por el segundo es igual al tercero. Y cuando la visualización no es correcta, este producto vectorial tiene el signo cambiado. Así, para calcular el factor signo se detecta cuál de las dos posibilidades se da en cada caso y el factor toma el valor +1 o -1 en consecuencia. El cambio de signo del vector no afecta a la dirección del vector, y por tanto tampoco a la rotación del glifo.

En la Figura 6.1.b se muestra la misma zona de la imagen después de aplicar este factor.

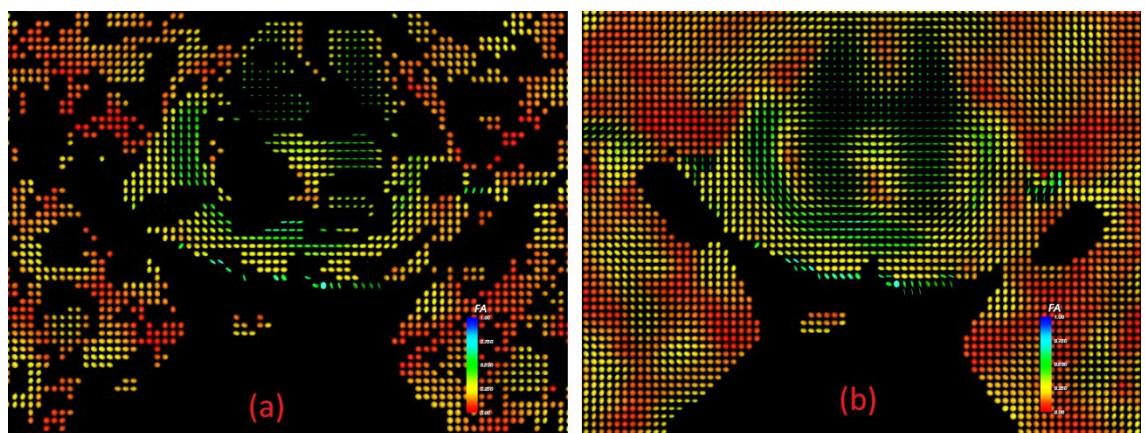


Figura 6.1. Justificación del factor signo: (a) no se aplica el factor signo, (b) sí se aplica

6.2.4 Cumplimiento de los objetivos

En un punto anterior se presentaban algunos objetivos que se esperaban cumplir con esta clase. A continuación se explica cómo se consigue cada uno de ellos:

- Creación de glifos: se utilizan métodos de *DTITensor* para extraer los autovalores y autovectores del tensor, y *vtkTransform* para modificar la geometría del glifo.
- Integración en Saturn: la variable *Bounds* permite la visualización de un solo corte.
- Formas de glifo: la clase soporta elipsoides, cuboides y supercuádricas.
- Coloreado de glifos: soporta anisotropía fraccional (FA), anisotropía relativa (RA) y el coeficiente geométrico lineal (c_l).
- Visualización de una zona de la imagen: lo permite el parámetro *Bounds*, que marca los límites.
- Discriminación de glifos: se puede usar como umbral la anisotropía fraccional o los coeficientes geométricos lineal y esférico.
- Glifos en puntos arbitrarios: la clase admite puntos de entrada, interpola los tensores en todos ellos y muestra los glifos.
- Resolución de los glifos: se indica en la creación de las fuentes.
- Flexibilidad: el código de la clase es claro, y la adición de nuevas opciones (por ejemplo nuevas posibilidades de filtrado) es sencilla.

6.2.5 La clase *vtkTensorGlyphDTI* frente a *vtkTensorGlyph*

Las clases *vtkTensorGlyphDTI* y *vtkTensorGlyph* tienen un fundamento común, la representación gráfica de un campo tensorial mediante glifos. De hecho, la clase que aquí se presenta empezó a desarrollarse tomando como referencia la clase *vtkTensorGlyph*, y fue desarrollándose según las necesidades específicas hasta presentar el código actual, totalmente renovado y reformado.

Por ello, cabe destacar las diferencias respecto a *vtkTensorGlyph*, y la justificación de estos cambios:

- Formato de entrada: fue la causa primera para la creación de una nueva clase. El uso de *vtkTensorGlyph* supone una conversión de tipos, entre datos VTK e ITK, que no es necesaria con la nueva clase.
- Extracción de autovalores y autovectores: derivada de la anterior. La clase *DTITensor* permite extraerlos de forma directa, mientras que *vtkTensorGlyph* copia cada tensor en una matriz para extraerlos con la función *Jacobi()*.
- Generación de la fuente: en *vtkTensorGlyph* se recibe como parámetro de entrada. En la nueva clase, por el contrario, sólo se recibe la elección de la geometría del glifo, y la clase genera la fuente. Esto mejora la transparencia, al evitar al usuario de la clase la tarea de generar dicha fuente.
- Desaparecen los parámetros *ThreeGlyphs* y *Symmetric*, que permiten mostrar varios glifos en cada posición. En DTI se muestra un glifo en cada punto.
- Desaparece la variable *ExtractEigenValues*. Este parámetro no tiene sentido en DTI, donde la extracción de autovalores es imprescindible para la generación de glifos.
- Desaparecen los parámetros *MaximumScaleFactor* y *ClampScaling*. Al normalizar los autovalores del tensor, el factor de escala coincide con el máximo factor de escala, y por tanto estos parámetros pierden sentido.
- Desaparece el parámetro *Length*, que permite dar al glifo un tamaño fijo. Con la normalización de autovalores, esto se consigue simplemente con el parámetro *ScaleFactor*.
- Cambian los valores que puede tomar *ColorMode*. La clase *vtkTensorGlyph*, las dos opciones son colorear por autovalor y colorear por escalares. El parámetro es ahora más específico, e indica cuál de los coeficientes del tensor debe utilizarse.
- Desaparece la variable *ColorGlyphs*. Los glifos se colorean siempre.

Obviando las diferencias anteriores, la generación de glifos tiene en líneas generales los mismos pasos en ambas clases: extracción del tensor, obtención de autovalores y autovectores, aplicación de la transformación a la fuente, generación de celdas, inclusión de los elementos en el volumen de salida. Sin embargo, cabe mencionar que en *vtkTensorGlyphDTI* la creación de las celdas es el último paso, y no uno de los primeros como sucede en *vtkTensorGlyph*. La justificación es que, debido a la discriminación de tensores, al comienzo del proceso no se conoce el número total de glifos. Se utiliza un contador para conocer el número final y generar las celdas.

6.2.6 Uso de la clase

La clase *vtkTensorGlyphDTI* se sitúa en los primeros pasos del pipeline de VTK, al hacer la transición entre un conjunto de datos y un volumen preparado para visualizar. Así, si se dispone de una imagen tensorial como la usada en Saturn, el uso de la clase es simple.

Después de crear una instancia de la clase, se debe usar el método *SetInput()* para pasarle la imagen de entrada, de la que se obtienen los datos. Además, según el tipo de visualización que se desee, se especificarán los límites de visualización (*Bounds*) para la visualización de planos, o un conjunto de puntos VTK (*vtkPoints*) para la visualización en posiciones específicas. Es necesario al menos uno de estos pasos para que realmente se dibuje algún glifo. También es posible utilizar los dos modos al mismo tiempo.

A continuación se deben indicar a la clase el resto de parámetros deseados (geometría del glifo, tipo de coloreado, etc.). Estos parámetros pueden provenir de una interfaz gráfica o de otro tipo, ser elegidos por el desarrollador, etc. Todos estos parámetros toman valores por defecto, por lo que este paso es opcional.

Una vez definidas las características de los glifos, se debe realizar una llamada al método *GetOutput()*. Este método devuelve un puntero a un objeto de tipo *vtkPolyData*, con el que se puede recorrer el resto del pipeline (*mapper*, *actor*, etc.) para obtener gráficamente los glifos.

6.3 Código en *TensorConsole*

La interfaz para glifos DTI cuenta con varios métodos y variables incluidos en la clase *TensorConsole*. Estos métodos sirven de puente entre la interfaz de

gráfica y la clase *vtkTensorGlyphDTI*, y entre esta clase y la visualización final de los glifos. Las variables son las siguientes:

- *m_activeActorX, m_activeActorY, m_activeActorZ, m_tractActor*: objetos del tipo *vtkActor*. Son los actores que actualmente se representan en el visor 3D.

```
vtkActor* m_activeActorX;
vtkActor* m_activeActorY;
vtkActor* m_activeActorZ;
vtkActor* m_tractActor;
```

- *m_scalarBar*: actor que representa una barra de escalares, relacionando los colores de los glifos, con los valores correspondientes.

```
vtkScalarBarActor* m_scalarBar;
```

- *m_puntosTract*: puntos de la tractografía. Este objeto toma los puntos de la última tractografía computada, y si el usuario así lo desea, se representarán glifos en estos puntos.

```
vtkPoints* m_puntosTract;
```

- *m_planoActivoX, m_planoActivoY, m_planoActivoZ*: indican si el plano visualizado está siendo visualizado actualmente en Saturn. Estas variables sirven para saber, al activar la visualización de glifos, qué glifos deben dibujarse.

```
bool m_planoActivoX;
bool m_planoActivoY;
bool m_planoActivoZ;
```

- *m_tractActiva*: como en el caso anterior, esta variable indica si se está visualizando una tractografía.

```
bool m_tractActiva;
```

Por otra parte, se añaden seis métodos nuevos, cuyos prototipos se muestran a continuación:

```
void verGlifos(int);
void verGlifosTract();
void borrarGlifos(int);
void cambiarOpacidad(float);
void glifosActivos();
void imagenActiva(int,bool);
```

El primer método es *verGlifos()* encargado de la visualización de glifos en planos. Esta función recibe el parámetro *numImagen*, que indica cuál de los tres planos debe mostrarse. Los valores de *numImagen* son 0 para el plano sagital (X), 1 para el plano coronal (Y), 2 para el plano axial (Z).

El método *verGlifos()* crea un objeto del tipo *vtkTensorGlyphDTI*, y le introduce la imagen tensorial y los diferentes parámetros. Estos parámetros se obtienen de los elementos activos del panel de configuración Glifos DTI, creado para este proyecto. Los controles del visor 3D de Saturn indican los índices de recorte (*Bounds* en *vtkTensorGlyphDTI*) del plano visualizado, mientras que el panel de configuración determina el recorte de los otros dos. Es decir, si en el visor 3D se visualiza el plano axial 20, los índices mínimo y máximo para el plano Z serán ambos 20, mientras que los índices de recorte para el plano X y Y se obtienen del panel de configuración Glifos DTI.

Los glifos se obtienen con el método *GetOutput()*, y el volumen se renderiza con el método *ConnectMapper()* del visor 3D. Se utiliza un actor y un volumen *vtkPolyData* para cada plano X, Y, Z, para que se puedan mostrar los tres al mismo tiempo. También se dibuja una barra de escalares, para mostrar la correspondencia numérica entre los colores y el valor que representan.

El método *verGlifosTract()* se encarga de la visualización de glifos en tractografía. Su funcionamiento es similar que el del método *verGlifos()*, con la diferencia de que en este caso los valores de *Bounds* son puestos a cero, para no visualizar glifos en los cortes, y se pasa a la clase *vtkTensorGlyphDTI* el conjunto de puntos *m_puntosTract*, que contiene los puntos computados en la tractografía.

En tercer lugar se encuentra la función *glifosActivos()*. Este método es llamado cuando la visualización de glifos pasa a estar activa. Comprueba dónde se deben visualizar glifos (variables *m_planoActivoX*, etc. y *m_tractActiva*), y llama a los métodos correspondientes.

El método *imagenActiva()* se encarga de modificar las variables booleanas que indican las visualizaciones activas (*m_planoActivoX*, *m_planoActivoY*, etc.). Recibe dos parámetros, un entero que indica qué variable se desea modificar (0 para el plano X, 1 para el Y, 2 para el Z, 3 para la tractografía) y un booleano con el nuevo valor de la variable (true para visualizar, false para no visualizar).

El método *borrarGlifos()* elimina todos los glifos de uno de los tres planos del espacio. El parámetro de entrada *numImagen* indica cuál de estos planos debe eliminarse: 0 para el plano X, 1 para el plano Y, 2 para el plano Z. Este método elimina el objeto *vtkPolyData* y el actor correspondientes al plano seleccionado.

El método *cambiarOpacidad()* modifica de forma dinámica la opacidad de los glifos visualizados actualmente.

6.4 Interfaz de usuario

La interfaz de usuario para la visualización de glifos está contenida en un nuevo panel de configuración de Saturn, llamado Glifos DTI. La interfaz ha sido diseñada de forma que ofrezca todas las opciones implementadas, que su uso sea lo más sencillo posible, y que su diseño siga la misma línea que otros paneles de Saturn.

La Figura 6.2 muestra el aspecto de la interfaz diseñada. Se distinguen cinco zonas en la misma: Color, Ejecución, Glifos, Recorte de planos y Filtrado, cada uno con los parámetros que ya se han mencionado con anterioridad.

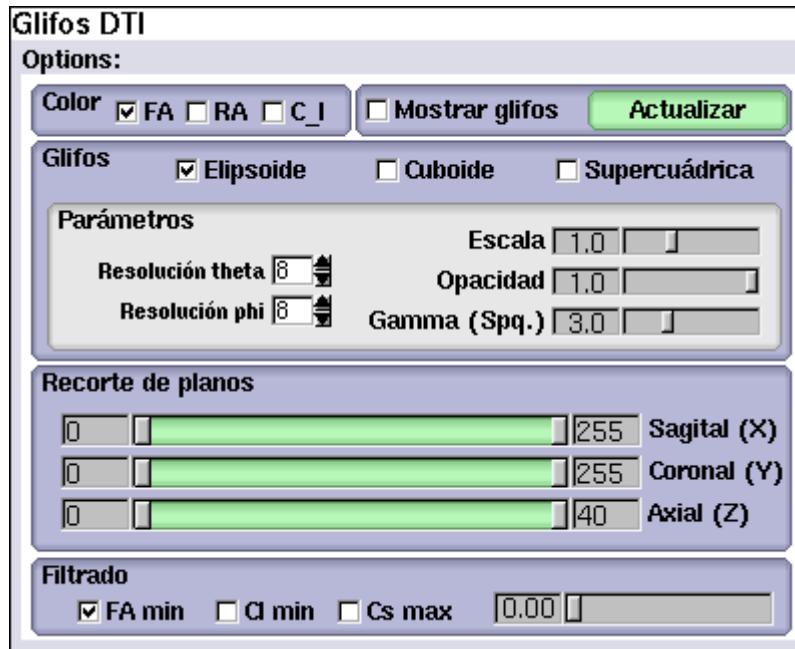


Figura 6.2. Interfaz de usuario para glifos en DTI

En el cuadro Ejecución aparece la opción de mostrar o no los glifos. La visualización de glifos es una operación con una carga computacional considerable, y si no es necesaria es recomendable desactivarla. El cuadro también contiene el botón Actualizar, que sirve para refrescar la visualización, y se utiliza cuando se ha modificado alguno de los parámetros de los glifos. Todos los parámetros y opciones que se explican a continuación requieren dibujar de

nuevo los glifos para tener efecto, a excepción de la opacidad de los glifos, que cambia de forma dinámica.

En el cuadro Color, las tres opciones que se ofrecen para colorear los glifos son anisotropía fraccional (FA), anisotropía relativa (RA) y coeficiente geométrico lineal (c_l).

En el cuadro de Glifos se ofrecen las tres posibles geometrías de glifo: elipsoide, cuboide y supercuádrica. Además aparece otro grupo de parámetros: la resolución angular de la fuente en phi y theta (longitud y latitud respectivamente), que no afecta a los cuboides; el parámetro gamma característico de los glifos supercuádricos; y la opacidad de los glifos.

En el cuadro Recorte de planos se especifican los límites de visualización de glifos para cada plano del espacio. En cada barra de desplazamiento aparecen dos elementos móviles, que marcan los índices mínimo y máximo para cada plano. Como se observa en la Figura 6.3, la interfaz resalta el espacio entre los dos límites para cada plano.

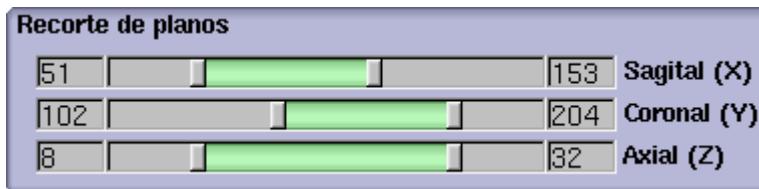


Figura 6.3. Aspecto al variar los índices de recorte de planos

En el cuadro Filtrado, por último, aparece las tres posibilidades de discriminación de glifos: anisotropía fraccional mínima, coeficiente lineal mínimo y coeficiente esférico máximo. También aparece un deslizador para escoger el valor del umbral.

La interfaz de usuario ha sido creada en Fluid, y está implementada en la clase *TensorGUI*. El valor por defecto de cada parámetro es el que se muestra en la Figura 6.2.

6.4.1 Forma de uso

Para visualizar en planos

La visualización en planos permite estudiar la difusión con mayor profundidad, al ofrecer las direcciones de difusión en cada punto. Los pasos a seguir son los siguientes:

1. Abrir un volumen tensorial. Para ello, acudir al menú *File >> Open tensor...* En la nueva ventana, hacer clic en *Explore* para que aparezca el navegador de directorios. Buscar y seleccionar el fichero con los datos de tensores, y hacer clic en *OK* en las dos ventanas abiertas, para que se carguen los tensores.
2. Cambiar a la vista 3D, en el menú *View*.
3. Activar la orientación u orientaciones que se desea visualizar (Sagital, Coronal, Axial).
4. Seleccionar el número de plano que se desea visualizar en cada caso. Se pueden usar las flechas laterales, o hacer clic directamente sobre el deslizador.
5. Abrir el panel de configuración para glifos: *DTI >> Glifos DTI*.
6. Configurar los parámetros preferidos para la visualización.
7. Marcar la opción *Mostrar Glifos*, en la parte superior derecha del panel.

Para visualizar glifos en tractografía

La visualización de glifos en tractografía permite conocer con más detalle las características de la difusión a lo largo de los tractos, ya que muestra las direcciones de la difusión, y la magnitud de la misma para cada dirección. Los pasos a seguir son los siguientes:

1. Abrir un volumen tensorial. Para ello, acudir al menú *File >> Open tensor...* En la nueva ventana, hacer clic en *Explore* para que aparezca el navegador de directorios. Buscar y seleccionar el fichero con los datos de tensores, y hacer clic en *OK* en las dos ventanas abiertas, para que se carguen los tensores.

2. Realizar la tractografía. Hacer clic en uno o más puntos de la sustancia blanca en los visores 2D, configurar los parámetros preferidos en el panel *Tractography* y hacer clic en el botón *Tractography*.
3. Cambiar a la vista 3D, en el menú *View*.
4. Abrir el panel de configuración para glifos: *DTI >> Glifos DTI*.
5. Configurar los parámetros preferidos para la visualización. El recorte de planos y el filtrado no tienen efecto en este caso.
6. Marcar la opción *Mostrar Glifos*, en la parte superior derecha del panel.

6.5 Pruebas

Para evaluar el funcionamiento de la interfaz, se han realizado una serie de mediciones sobre el tiempo de carga, esto es, el tiempo que tarda el programa en dibujar los glifos. Este tiempo depende en esencia de tres factores: el número de glifos, la geometría del glifo y la resolución del mismo. Por ello, son éstas las variables que se han considerado a la hora de medir y comparar los tiempos.

En la Figura 6.4, se muestra una tabla con los tiempos de carga para distintos números de glifos. Los números corresponden al número de glifos en un corte típico en el plano coronal (2740 glifos), sagital (3850) y axial (9460) respectivamente. Se han empleado glifos elipsoidales con resolución 8 tanto en phi como en theta.

En la Figura 6.5 se muestra una tabla con los tiempos de carga para diferentes geometrías de glifos. En la tabla se comparan elipsoides, cuboides y supercuádricas. La resolución para elipsoides y supercuádricas es la misma, 8 en phi y theta, y la medición se ha hecho para el plano sagital (3.850 glifos).

En la Figura 6.6 aparece una tabla que muestra el efecto de la variación de la resolución en el tiempo de carga. Para ello, se han empleado glifos elipsoidales y cuatro resoluciones diferentes (aunque siempre la misma en phi y theta). Las mediciones se han realizado, de nuevo, para el plano sagital (3850 glifos).

Número de glifos	2740	3850	9460
Latencia	0.9 s	1.26 s	2.6 s

Figura 6.4. Comparación del tiempo de carga en función del número de glifos

Geometría de glifo	Elipsoide	Cuboide	Supercuádrica
Latencia	1.26 s	0.35 s	3.99 s

Figura 6.5. Comparación de la latencia en función de la geometría del glifo

Resolución	4	8	12	16
Latencia	0.34 s	1.26 s	2.52 s	4.61 s

Figura 6.6. Latencia en función de la resolución de los glifos

En estas tablas se observa la lógica influencia del número de glifos en el tiempo de carga, pero también es notable el efecto de la geometría del glifo y de su resolución.

Por un lado, las diferentes geometrías de glifos tienen también una variada complejidad, que se refleja en la latencia. El cuboide es un glifo muy sencillo, con muy pocas líneas, puntos y polígonos y por ello tiene un tiempo de carga muy rápido. La supercuádrica, sin embargo, necesita una gran complejidad para representar los detalles geométricos que la caracterizan, y esto ralentiza su representación. El elipsoide se encuentra en un punto intermedio, al tener una latencia más alta que el cuboide pero mucho menor que la supercuádrica.

Por otro lado, también se observa la influencia de la resolución en el tiempo de carga. Hay que notar que la latencia no aumenta proporcionalmente a la resolución sino de forma exponencial, por lo que la resolución debe ser escogida cuidadosamente en cada caso para no empeorar el rendimiento más de lo debido.

En cuanto a los detalles técnicos, las pruebas han sido realizadas sobre un equipo con un procesador Intel de doble núcleo a 1.8 GHz y 1 GB de memoria. Los valores de la tabla están promediados de 10 ejecuciones cada uno.

6.6 Ejemplos

En esta sección se muestran algunos ejemplos de la visualización de glifos con la nueva interfaz. La Figura 6.7 muestra la visualización en el plano axial (plano Z) que resulta de seguir los pasos indicados anteriormente. En esta imagen, los glifos están superpuestos con la visualización plana, por lo que al acercar la imagen, se comprobaría que el plano corta a los glifos. Por ello, para una visualización óptima, se recomienda reducir la opacidad del plano (parte superior derecha del visor 3D), como se muestra en la Figura 6.8.

La Figura 6.9 muestra los glifos desde una distancia más cercana. Se puede controlar el visor 3D (zoom, rotación, etc.) mediante la interfaz de Saturn, o con el ratón directamente sobre la imagen.

La visualización de glifos en los planos sagital (plano X) y coronal (plano Y) se muestra en la Figura 6.10 y en la Figura 6.11, respectivamente. La Figura 6.12 muestra la visualización en tres planos.

La Figura 6.13 muestra las distintas posibilidades para colorear los glifos: anisotropía fraccional (Figura 6.13.a), relativa (Figura 6.13.b) y coeficiente geométrico lineal (Figura 6.13.c). La Figura 6.14 muestra las diversas geometrías disponibles para los glifos: elipsoide (Figura 6.14.a), cuboide (Figura 6.14.b) y supercuádrica (Figura 6.14.c).

La Figura 6.15 muestra el efecto del factor de escala en la visualización de elipsoides: al aumentar el tamaño de los glifos (Figura 6.15.b), la visualización es más densa y los glifos con una linealidad elevada se ven más claramente que con su tamaño original (Figura 6.15.a). La Figura 6.16 muestra una posible aplicación del factor de escala a la visualización con cuboides. Cuando no se aplica el factor de escala (Figura 6.16.a), los cuboides tienen un tamaño excesivo, y se superponen entre sí. Con un factor de escala de 0.7 (Figura 6.16.b) los glifos más isótropos se distinguen correctamente.

La Figura 6.17 muestra un ejemplo de crop o recorte de planos. Al delimitar el área de interés, la visualización es más ágil y se puede mejorar su calidad. La Figura 6.18 muestra un ejemplo de discriminación de glifos. Se eliminan todos los glifos con una anisotropía fraccional menor de 0.2, lo que deja sólo aquellos glifos más anisótropos, que suelen ser más relevantes.

La Figura 6.19 muestra el efecto del parámetro gamma (γ) en la forma de las supercuádricas. Con $\gamma=1.0$ (Figura 6.19.a), las supercuádricas tienen una forma muy similar a la de los elipsoides. Con $\gamma=5.0$ (Figura 6.19.b), los bordes son más pronunciados y la orientación de los glifos se observa con más claridad.

La Figura 6.20, por último, muestra la visualización de glifos en tractografía.

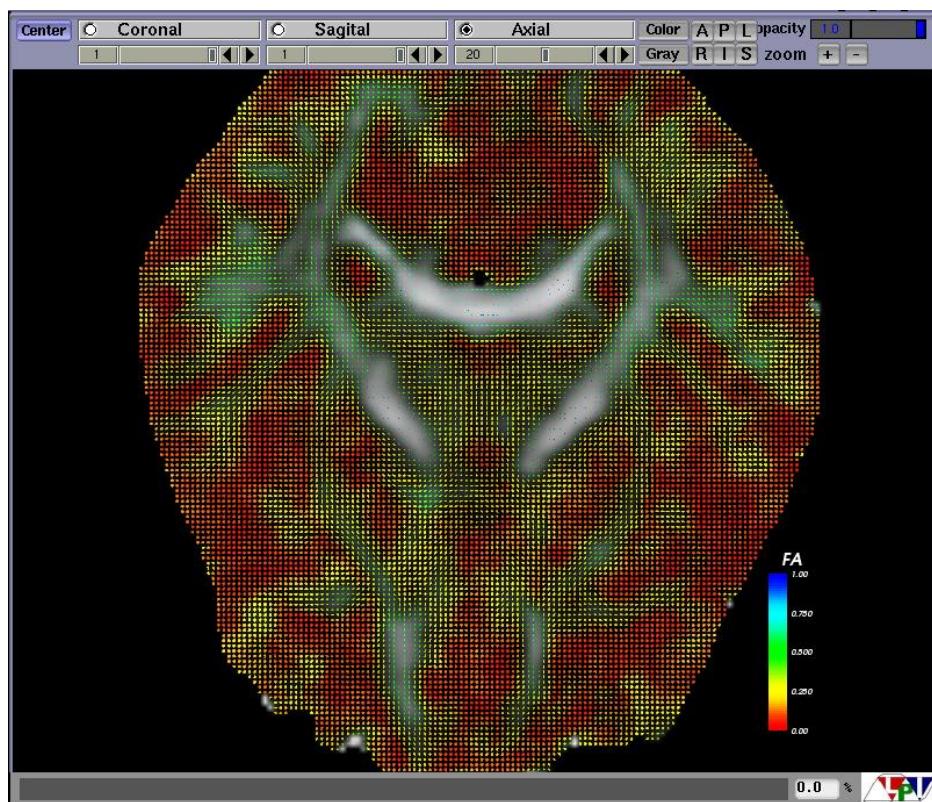


Figura 6.7. Visualización de glifos en el plano axial

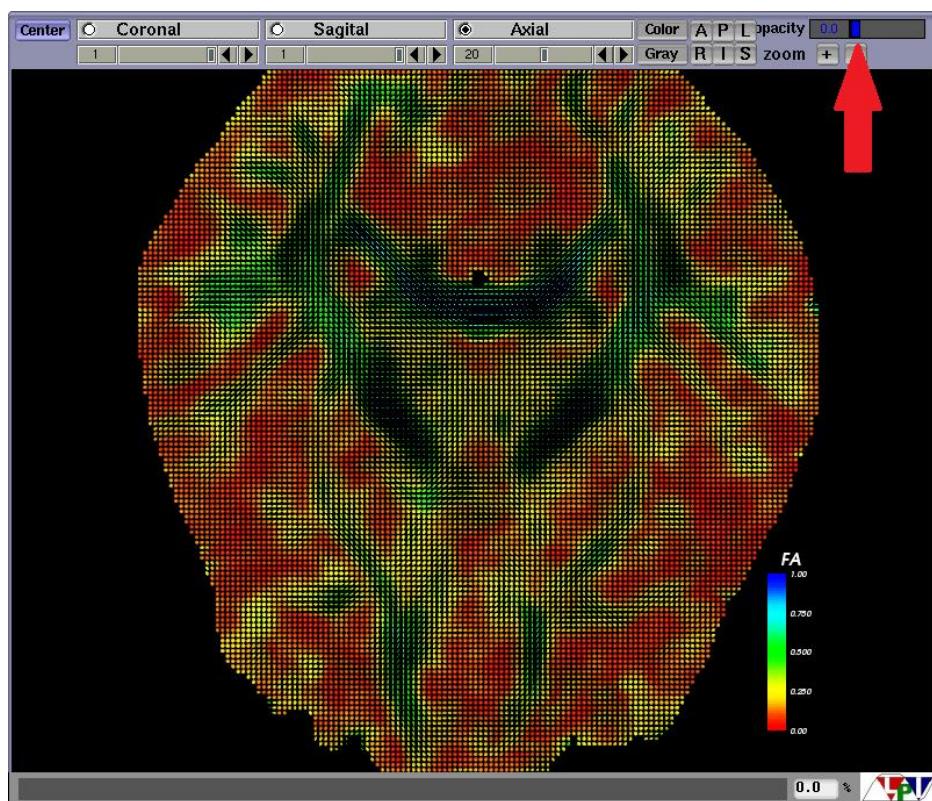


Figura 6.8. Vista con la opacidad reducida hasta cero

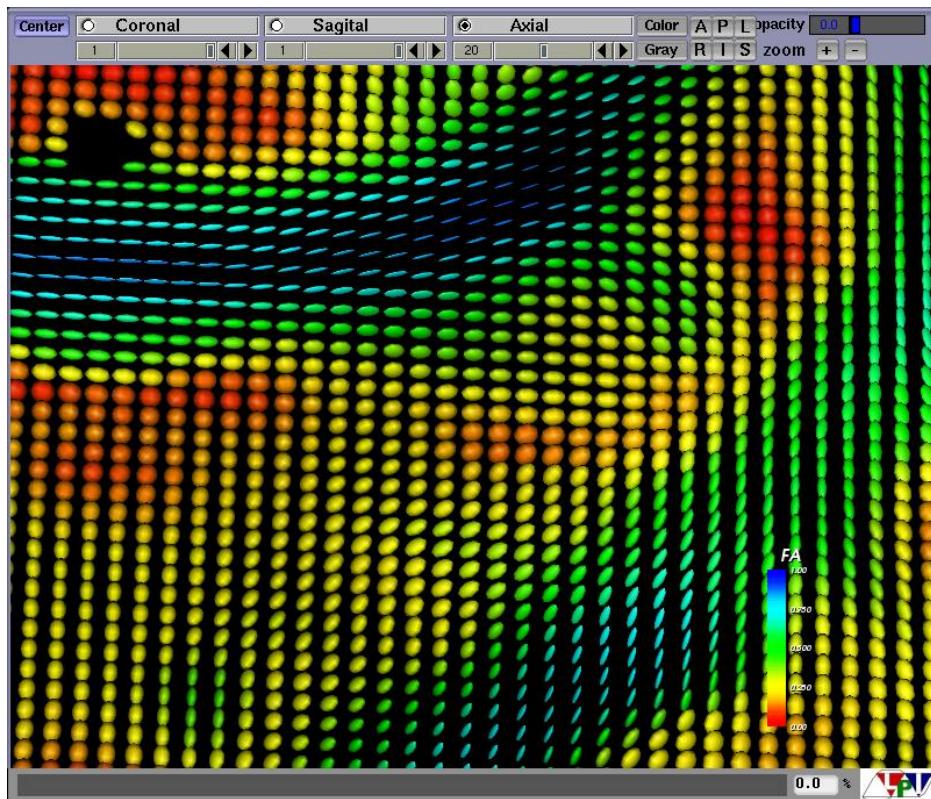


Figura 6.9. Visualización cercana de los glifos

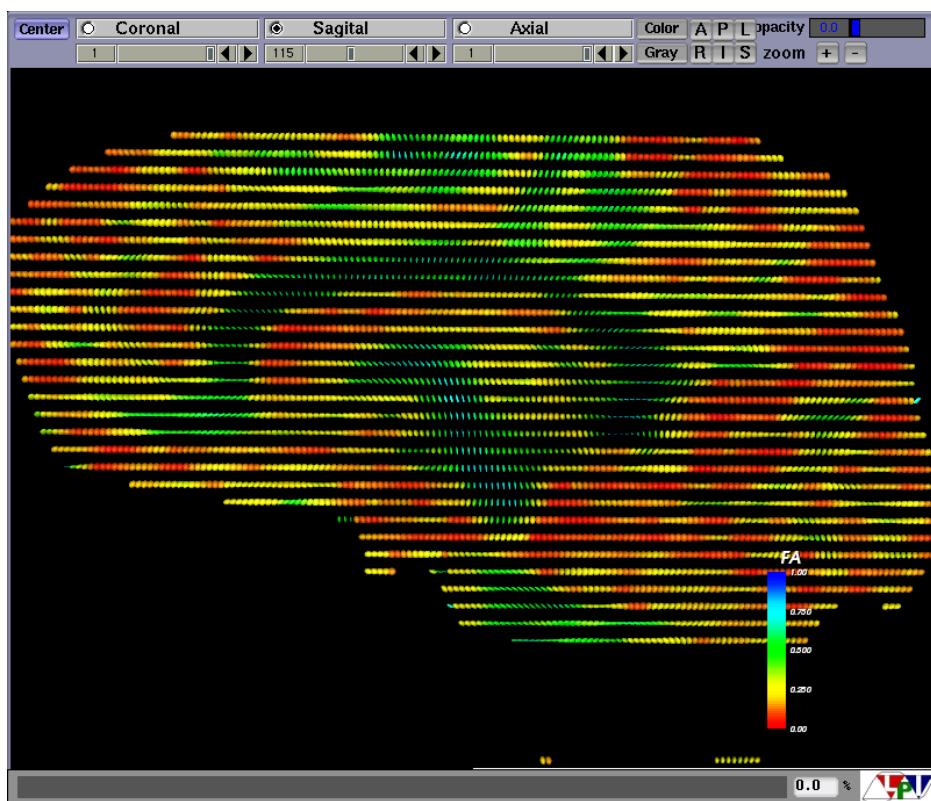


Figura 6.10. Glifos en el plano sagital (X)

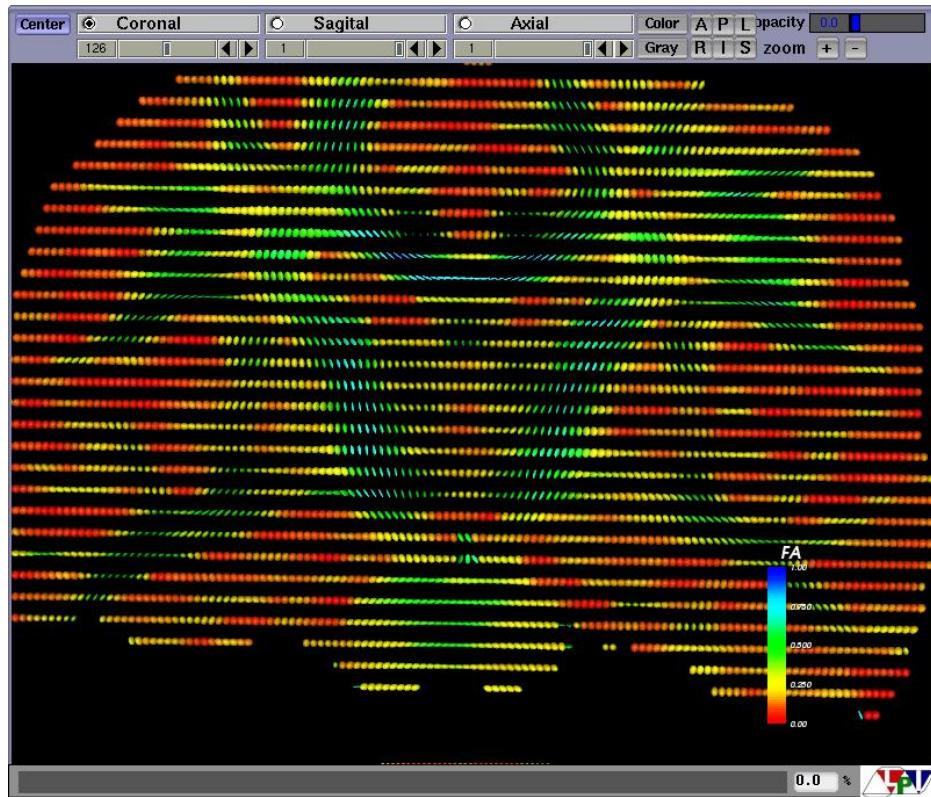


Figura 6.11. Glifos en el plano coronal (Y)

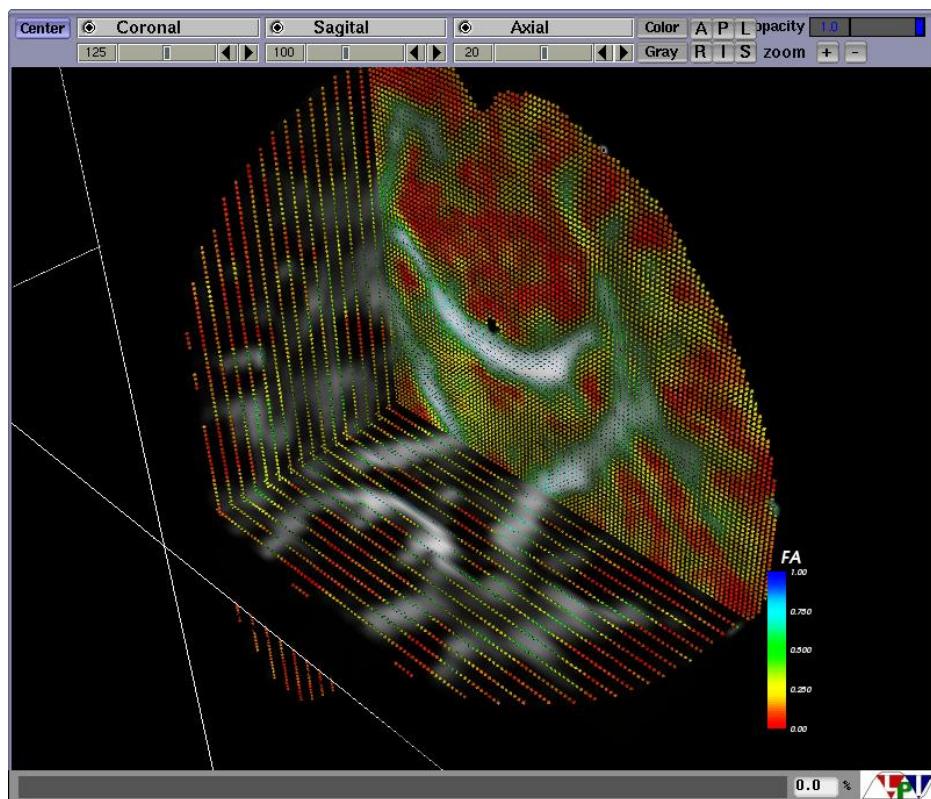


Figura 6.12. Glifos en tres planos

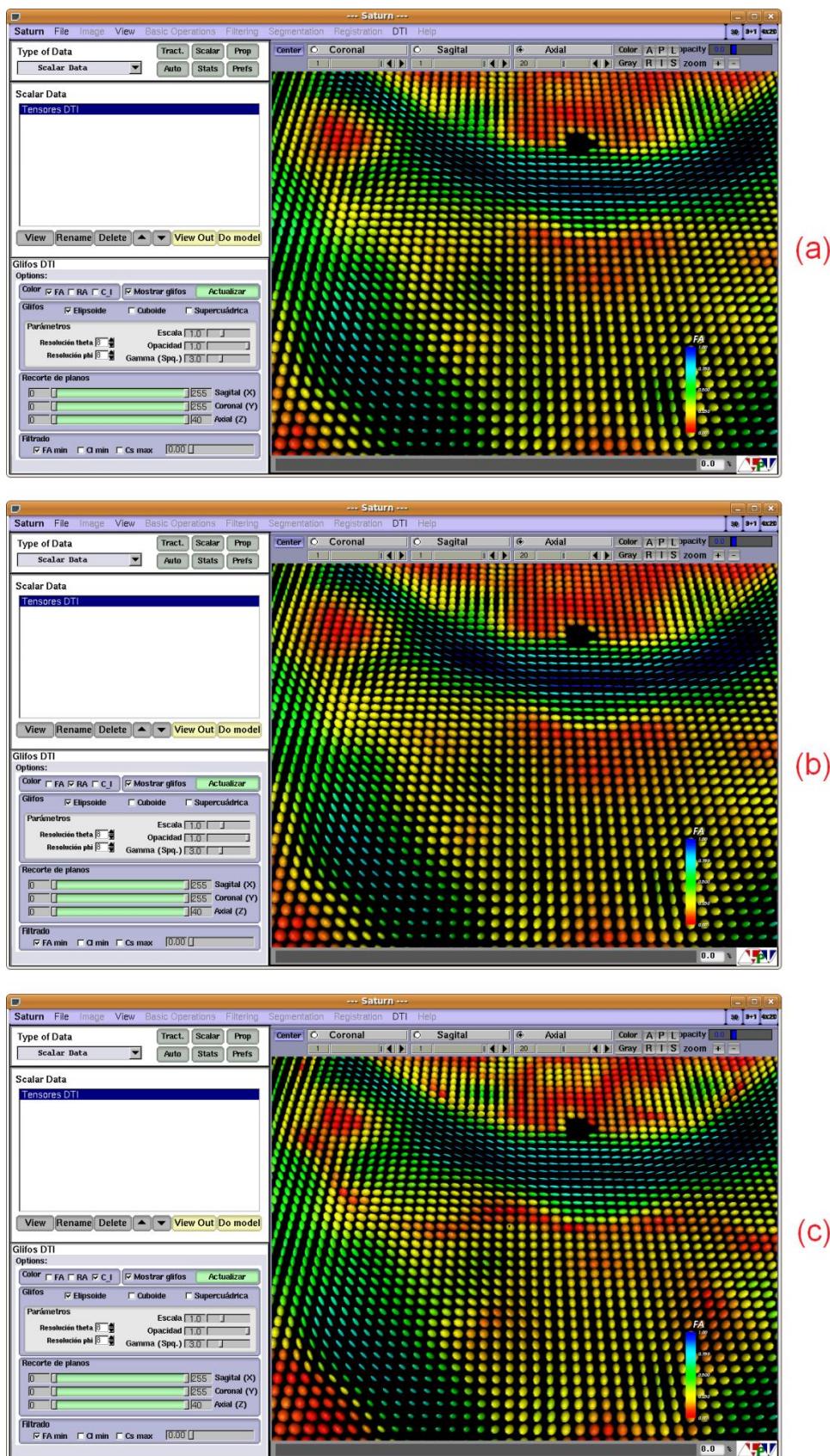


Figura 6.13. Distintos tipos de coloreado: (a) anisotropía fraccional, (b) anisotropía relativa, (c) coeficiente lineal

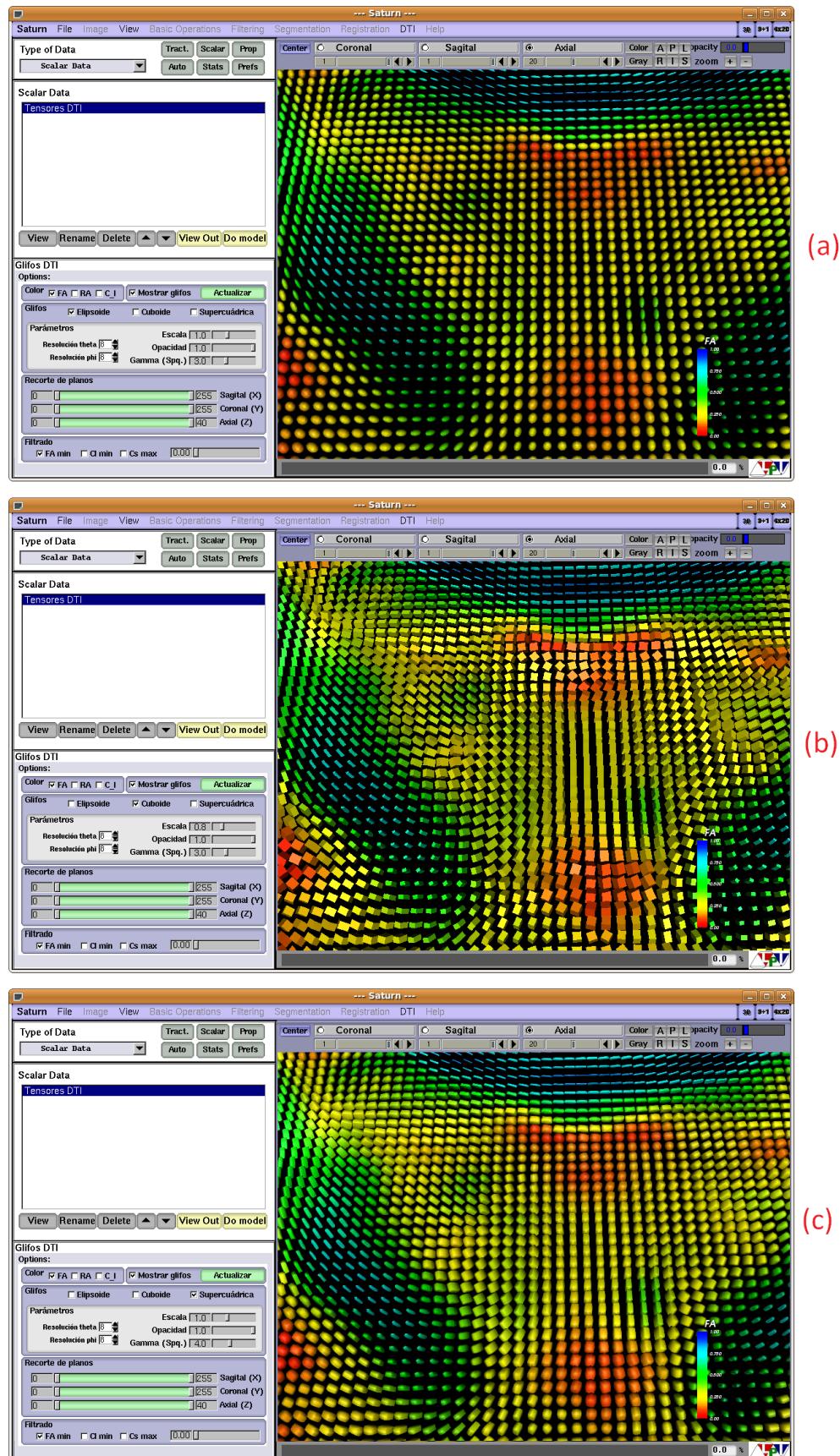


Figura 6.14. Diferentes geometrías de glifo: (a) elipsoides, (b) cuboides, (c) supercuádricas

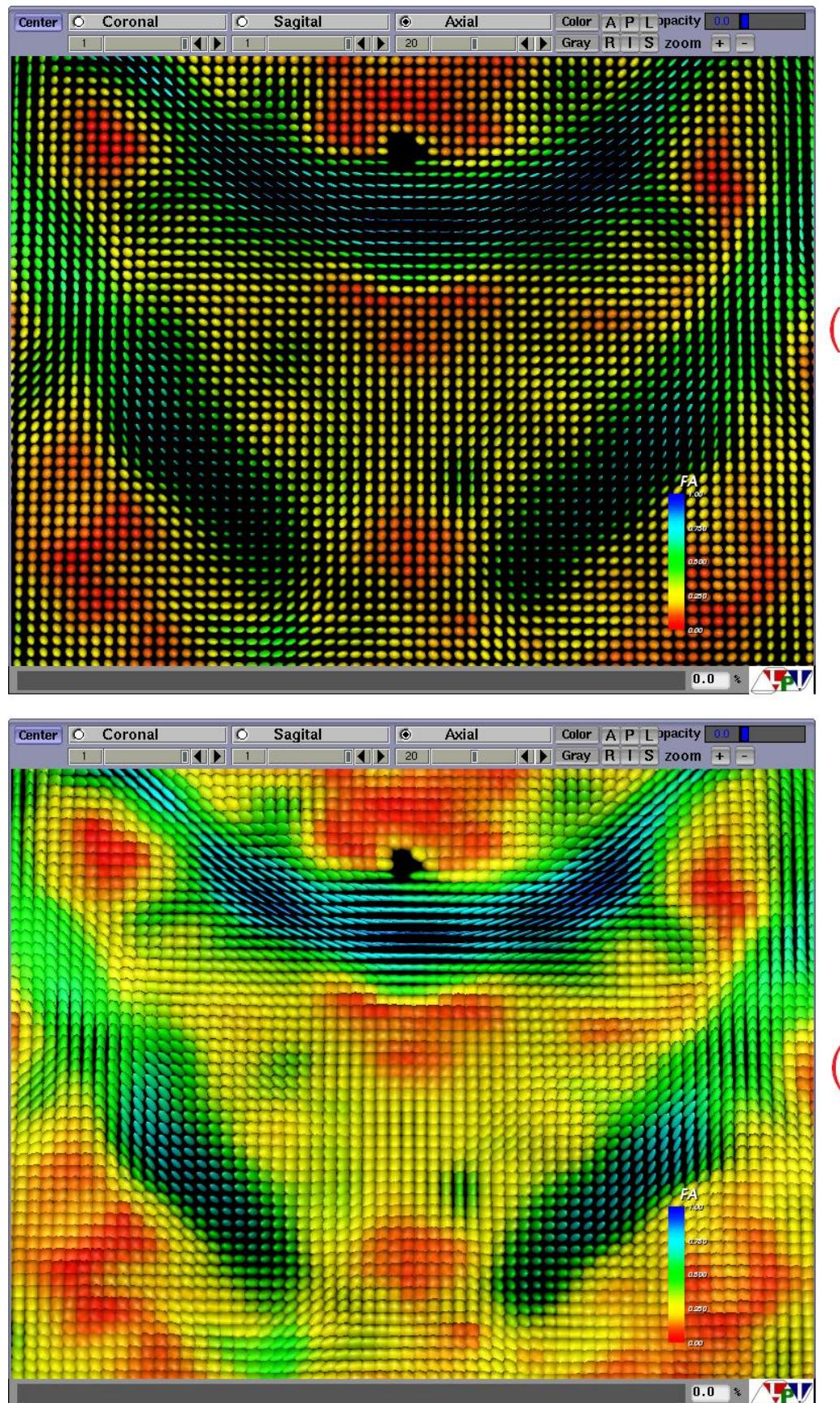


Figura 6.15. Efecto del factor de escala: (a) factor de escala 1.0, (b) factor de escala 2.0

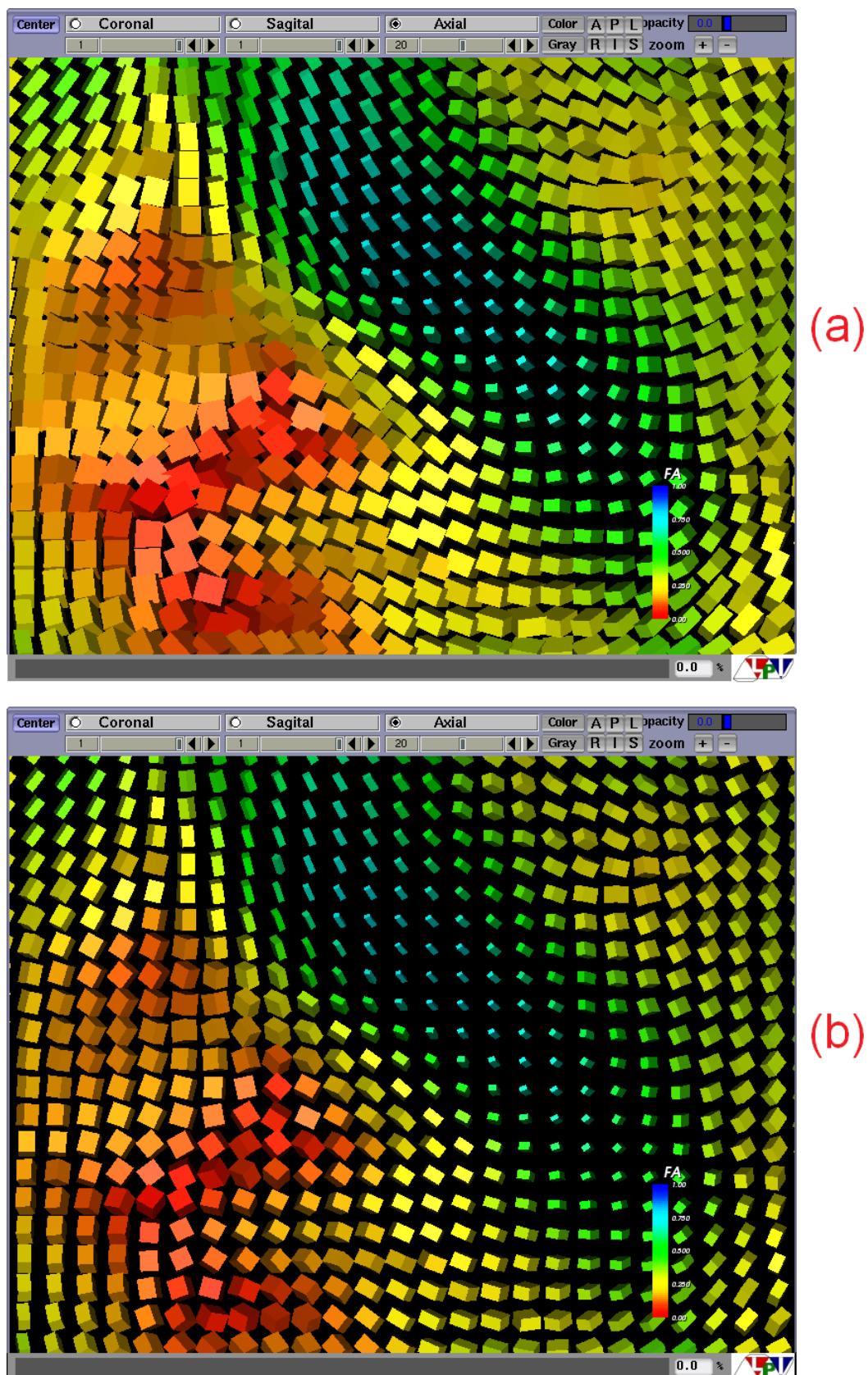


Figura 6.16. Aplicación del factor de escala a la visualización con cuboides: (a) factor de escala 1.0, (b) factor de escala 0.7

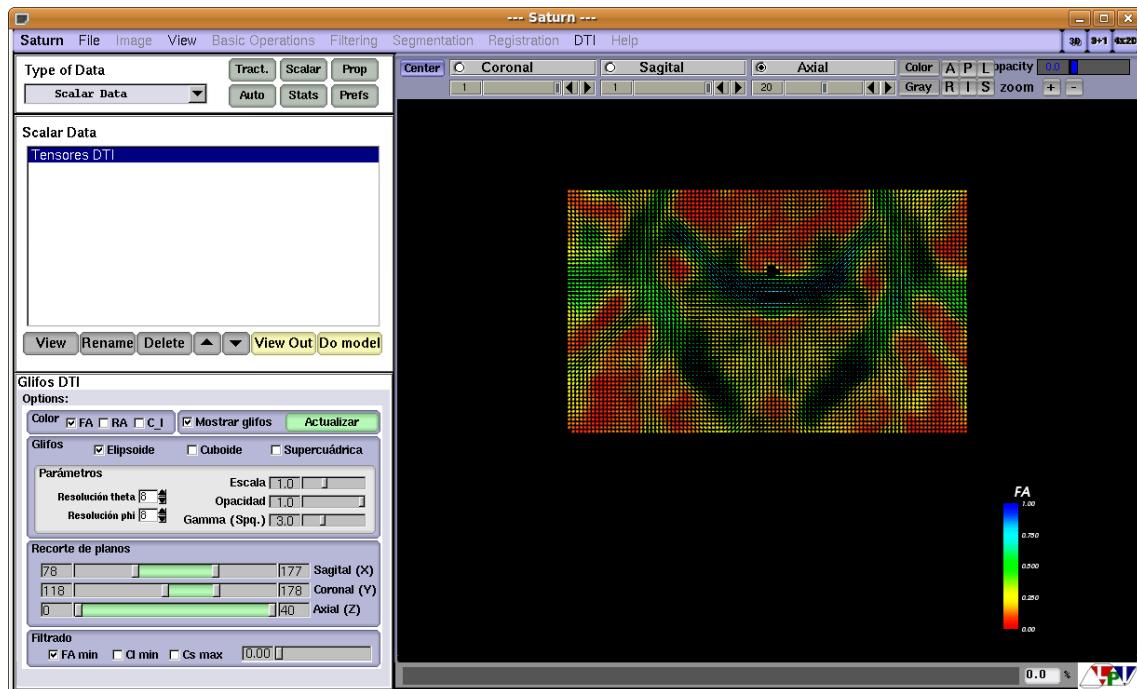


Figura 6.17. Crop de planos (recorte de planos)

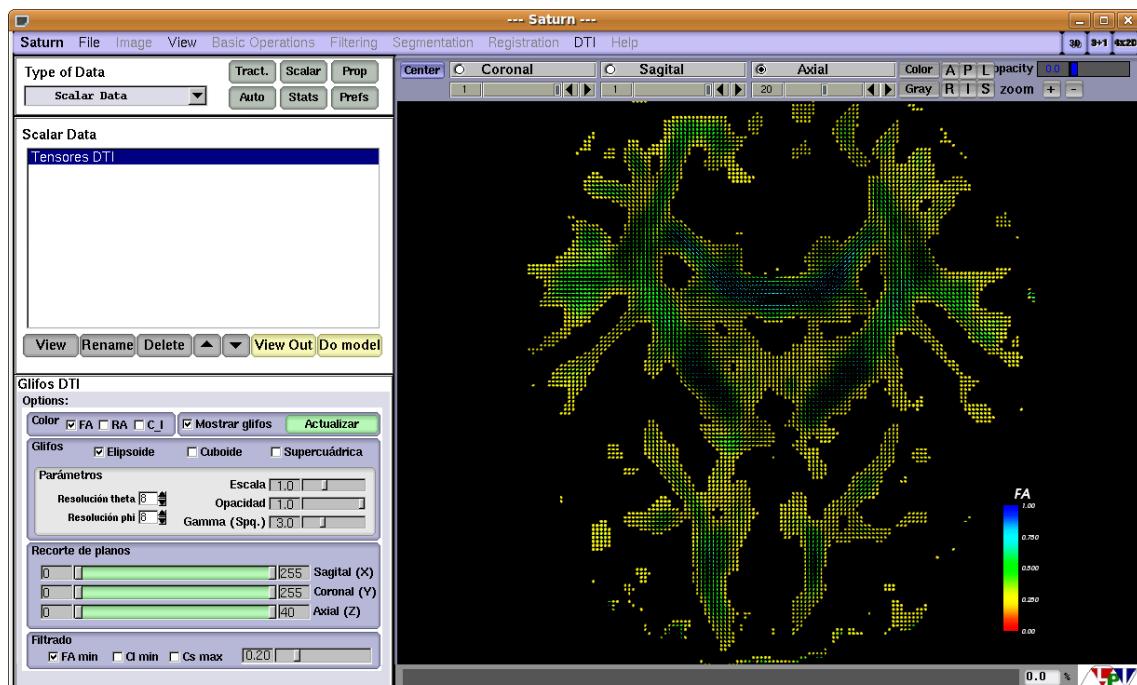


Figura 6.18. Discriminación de glifos (FA > 20)

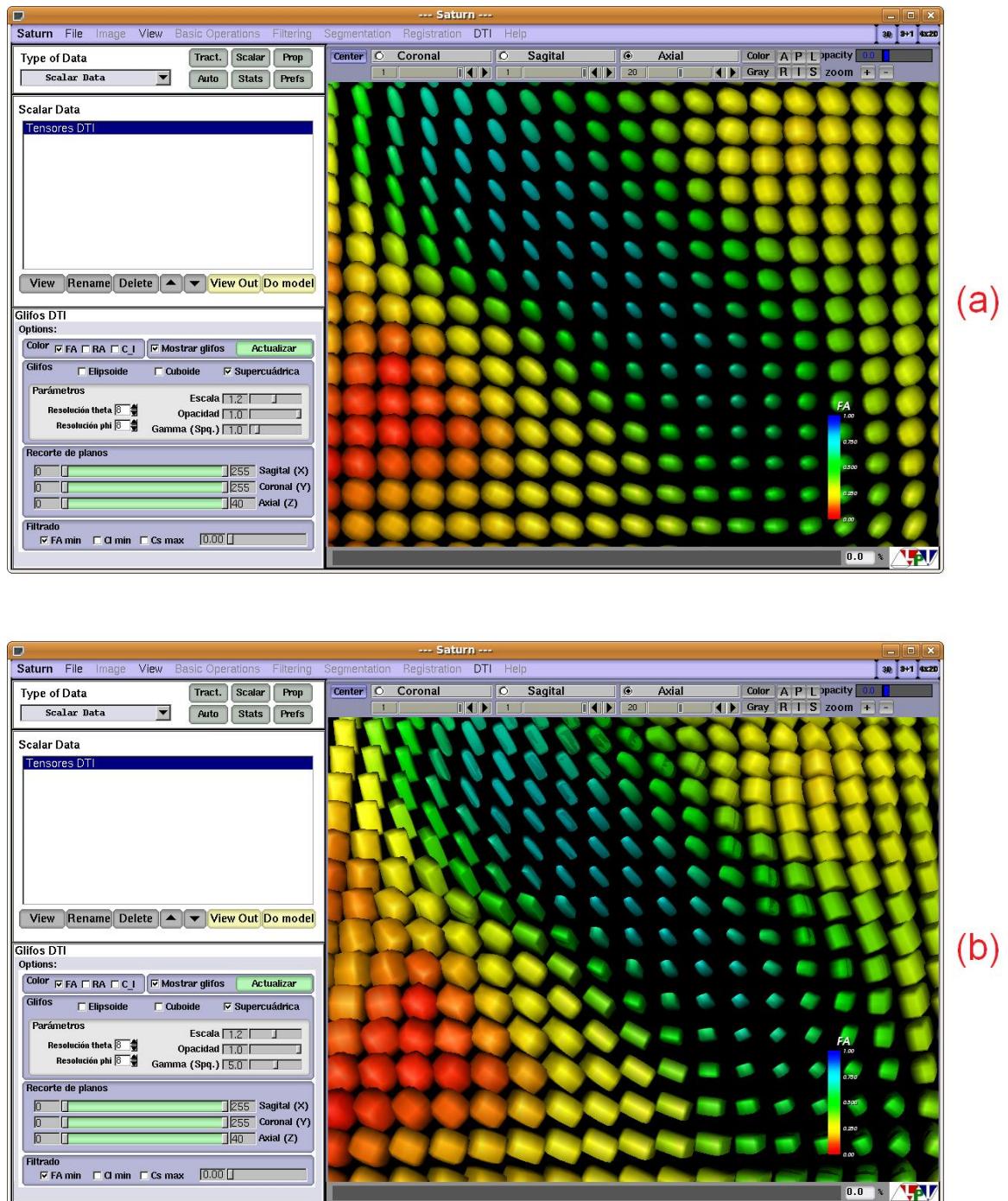


Figura 6.19. Efecto del parámetro gamma: (a) gamma 1.0, (b) gamma 5.0

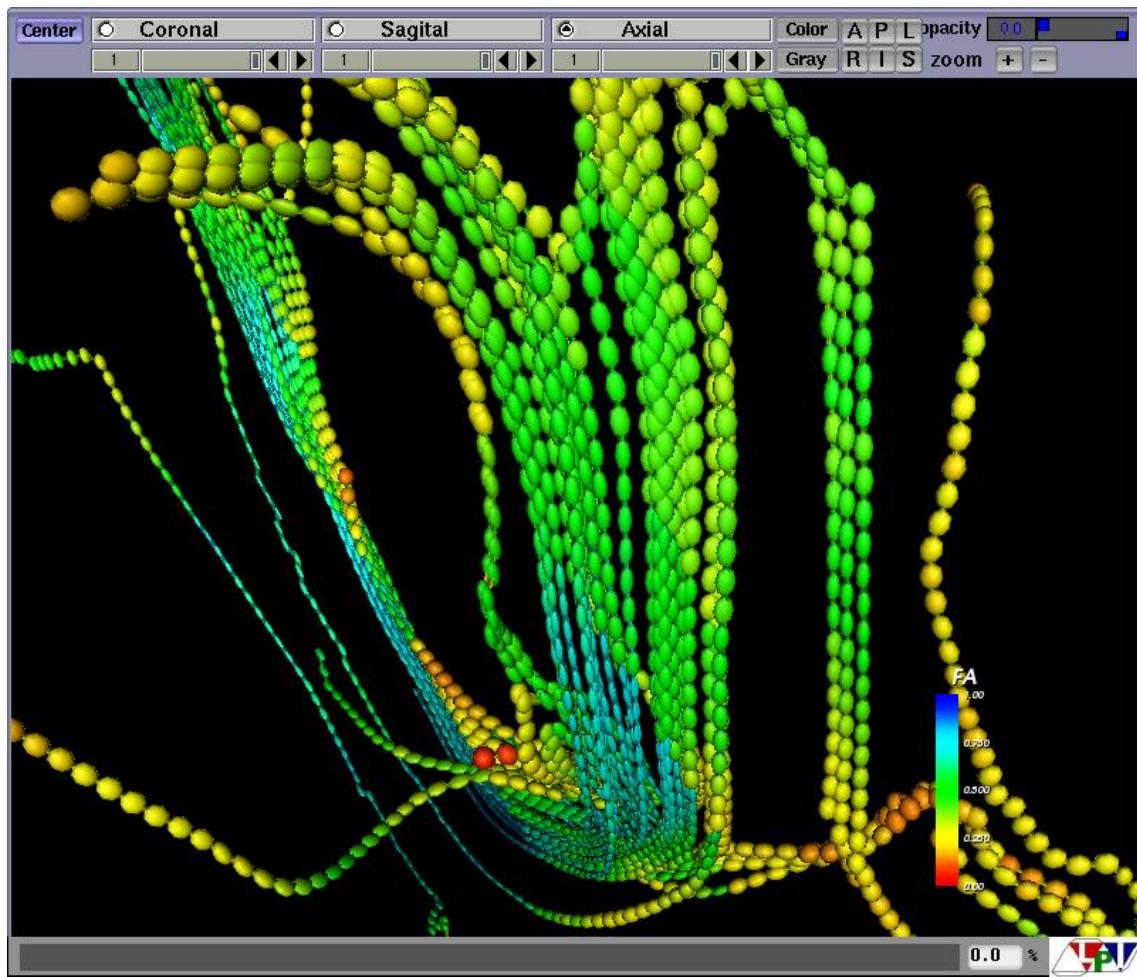


Figura 6.20. Glifos en tractografía

Capítulo 7

Desarrollo de una interfaz para tensor de esfuerzo

En este capítulo se explica la interfaz desarrollada para la visualización del tensor de esfuerzo. Se explican los nuevos elementos de Saturn a nivel de código y de interfaz, así como las características del nuevo módulo, su funcionamiento y su forma de uso.

7.1 Introducción

En este capítulo se presenta la implementación de la segunda parte del proyecto, correspondiente a tensor de esfuerzo. En este caso, el desarrollo de este aspecto en Saturn es nulo, por lo que el objetivo es la integración en Saturn de una primera interfaz. Este desarrollo conlleva, por lo tanto, la definición de nuevos tipos de datos, de imágenes, y de otros elementos que sirvan para representar el tensor de esfuerzo y trabajar con él, además de los métodos e interfaces para visualizarlo.

El capítulo se estructura como sigue: en primer lugar se presenta la clase *StrainTensor*, con la que se representan los datos. A continuación, se utiliza esta nueva clase para definir toda una serie de tipos de dato necesarios para representar, por ejemplo, imágenes completas de tensor de esfuerzo. El siguiente punto explica los métodos añadidos a la clase *TensorConsole*. Estos métodos gestionan la visualización del tensor de esfuerzo, tanto de imágenes escalares como de glifos. A continuación se explican otros elementos utilizados para el procesado de los datos o su visualización, como las clases *ComputeStrainScalars* o *vtkTensorGlyphStrain*, y la lectura y escritura de los datos en ficheros. En cuarto lugar, se presenta la interfaz desarrollada, tanto sus elementos como su forma de uso.

7.2 Clase StrainTensor

La clase *StrainTensor* es el tipo de dato con el que se representa el tensor de esfuerzo en este proyecto. La clase sigue el modelo de *DTITensor*, explicada en la sección 5.3.6 , y está definida en los ficheros *itkStrainTensor.h* y *.cxx*, bajo el nuevo directorio *strain*. *StrainTensor* se trata de un array de tres elementos que contiene los tres elementos independientes del tensor de esfuerzo 2x2 (el tensor es simétrico).

En la clase se definen una variedad de métodos para trabajar con los tensores. Estos métodos permiten calcular los autovalores y autovectores del tensor, calcular el logaritmo y la exponencial del tensor, u obtener diversos escalares asociados, como el determinante, la traza o el invariante del tensor, una magnitud tomada de [35]109 (y llevada al caso de tensores 2x2), y que representa la tasa de esfuerzo total.

7.3 Tipos de dato

Estos son los nuevos tipos de dato utilizados para imágenes de tensor de esfuerzo, y definidos en *UsimagToolBase.h*:

- *STPixelType*: tipo de píxel de las imágenes de tensor de esfuerzo. El tipo utilizado es la clase *StrainTensor*, que representa el tensor y define una serie de operaciones sobre él.

```
typedef itk::StrainTensor<float> STPixelType;
```

- *STImageType*: tipo de imagen empleado. Tiene cuatro dimensiones, tres espaciales y una temporal.

```
typedef itk::Image<STPixelType, 4> STImageType;
```

- *VectorOfSTDDataType*: define un contenedor para todas las imágenes de tensor de esfuerzo cargadas en la aplicación. *DataSTEElementType* es una nueva clase, similar a *DataTensorElementType* (ver sección 5.3.7) pero aplicada al caso de tensor de esfuerzo.

```
typedef VolumesContainer<DataSTELEMENTTYPE>
VectorOfSTDDataType;
```

- *STReaderType* y *STWriterType*: tipos utilizados para leer y escribir la imagen en ficheros.

```
typedef itk::ImageSeriesReader <VectorImage4DType>
STReaderType;
```

```
typedef itk::ImageSeriesWriter <VectorImage4DType,
VectorImage3DType >      STWriterType;
```

Los tipos *VectorImage3DType* y *VectorImage4DType* se definen porque el *Reader* y el *Writer* de ITK no pueden trabajar directamente con un píxel del tipo *StrainTensor*.

```
typedef itk::Image<itk::CovariantVector<float, 3>, 3>
VectorImage3DType;
typedef itk::Image<itk::CovariantVector<float, 3>, 4>
VectorImage4DType;
```

- *STImageType3D*: tipo de imagen de tensor de esfuerzo con tres dimensiones. Este tipo es necesario porque la lectura y escritura en ficheros sólo es soportada para imágenes de tres dimensiones.

```
typedef itk::Image< STPixelType, 3 >
STImageType3D;
```

- *ExtractFilterType*: Filtro para extraer una imagen tridimensional a partir de una imagen 4D. Se emplea porque muchas operaciones no son soportadas con imágenes 4D.

```
typedef itk::ExtractImageFilter< STImageType, STImageType3D >
ExtractFilterType;
```

- *ComputeStrainScalarsType*: filtro para obtener una imagen de escalares a partir de una imagen tensorial.

```
typedef itk::ComputeStrainScalars
<STImageType3D,InputImageType> ComputeStrainScalarsType;
```

```
typedef ComputeStrainScalarsType::Pointer
ComputeStrainScalarsPointer;
```

- *VTKstrainExportType*: tipo empleado para exportar una imagen de ITK a VTK. Se usa para conectar ambos pipelines.

```
typedef itk::VTKImageExport< InputImageType>
VTKstrainExportType;
```

Además de estos tipos de dato, se declara en *UsimagToolBase.h* una nueva variable, un contenedor que almacena todas las imágenes de tensor de esfuerzo cargadas en la aplicación:

```
VectorOfSTDDataType m_VectorSTData;
```

7.4 Clase ComputeStrainScalars

La clase *ComputeStrainScalars* cumple la tarea de convertir una imagen tensorial en una imagen de escalares. Esto permite mostrar un mapa de escalares describiendo alguna de las características de los tensores. Esta clase se encuentra definida en los ficheros *itkComputeStrainScalars.h* y *.cxx* bajo el directorio *strain*, y los escalares que soporta son los elementos del tensor, sus autovalores y el invariante del mismo.

7.5 Lectura y escritura de ficheros

La lectura y escritura de ficheros es una parte importante de la interfaz, ya que permite la entrada de datos de tensor de esfuerzo al programa. Esto se realiza con los métodos *LoadStrainTensor()* y *SaveStrainTensor()*, definidos en *UsimagToolConsole*.

Los métodos de ITK para la lectura y escritura de ficheros no soportan imágenes de cuatro dimensiones, por lo que la alternativa es usar almacenar la imagen de cuatro dimensiones en una serie de ficheros, cada uno de los cuales contiene una imagen de tres dimensiones.

Para la escritura en ficheros se muestra un explorador al usuario en el que éste elige el directorio de destino y el nombre base de los ficheros. A este nombre se le agrega un número de dos cifras que indica el número de imagen y una extensión (.vtk). La clase *NumericSeriesFileNames* se encarga de generar automáticamente estos nombres, y la clase *ImageSeriesWriter* escribe en el fichero.

En el caso de la lectura de datos, el usuario debe elegir en el explorador el primer fichero de la serie, con el número 00. La clase *ImageSeriesReader* se encarga de obtener la imagen 4D, que es alineada con el resto de imágenes cargadas (si las hay), y agregada al contenedor de imágenes de tensor de esfuerzo.

Los formatos soportados son VTK, NRRD, DICOM y Meta.

7.6 Clase vtkTensorGlyphStrain

La clase *vtkTensorGlyphStrain* se encarga de convertir los datos de tensor de esfuerzo en glifos que puedan ser visualizados. La utilización de glifos muestra de forma directa la orientación del esfuerzo en cada punto, ofreciendo así una descripción completa del tensor de esfuerzo.

La estructura y el código de *vtkTensorGlyphStrain* siguen el modelo de la clase *vtkTensorGlyphDTI*, desarrollada también para este proyecto y explicada en la sección 6.1 . La nueva clase es más sencilla que la anterior, pero conserva varias de sus funcionalidades:

- Integración con Saturn: la clase maneja tensores de esfuerzo 2x2, y utiliza el mismo tipo de dato que el resto de la aplicación, por lo que no es necesaria ninguna conversión de tipos.
- Generación de glifos: éstos tienen forma de romboide, siguiendo lo explicado en [31].
- Coloreado de glifos: según diferentes magnitudes escalares, como los elementos del tensor, los autovalores o el invariante del mismo.
- Glifos en posiciones diversas: por defecto, se dibujan glifos en los puntos donde existe el tensor de esfuerzo, pero también permite elegir un conjunto de puntos en los que interpolar y dibujar el tensor.
- Escalado de glifos: según un factor determinado por el usuario.

7.6.1 Tipos de dato

La imagen de entrada debe ser una imagen ITK de cuatro dimensiones con tipo de píxel *StrainTensor*. Asimismo, si se desean introducir puntos de entrada en los que mostrar glifos, estos puntos deben ser un objeto de tipo *vtkPoints*. Por último, la salida del método *GetOutput()*, el encargado de generar los glifos, es un objeto del tipo *vtkPolyData*. El resto de tipos utilizados internamente por la clase son los mismos utilizados por *StrainTensor*.

7.6.2 Variables de clase

Estas son las variables definidas a nivel de clase:

- *input*: imagen tensorial de entrada.

```
StrainImageType::Pointer input;
```

- *inputPoints* (por defecto, NULL): representa los puntos donde el usuario desea que se dibujen glifos. El tensor es interpolado en estos puntos. Si no se indican estos puntos, se representan glifos en los puntos de la imagen donde el tensor esté definido.

```
vtkPoints *inputPoints;
```

- *PlanoZ, Tiempo* (por defecto, 0 y 0): indican el corte que se desea representar y el instante de tiempo (tercera y cuarta dimensión de la imagen, respectivamente).

int PlanoZ, Tiempo;

- *Scaling* (por defecto, 1): indica si se le aplica un factor de escala a los glifos.

int Scaling;

- *ScaleFactor* (por defecto, 1.0): factor de escala aplicado a los glifos. Por defecto el factor de escala es 1 (no hay escalado), pero el tamaño de los glifos está normalizado.

double ScaleFactor;

- *ColorMode* (por defecto, INV): indica el tipo de coloreado que se aplica a los glifos.

int ColorMode;

Cada una de estas variables tiene su correspondiente método *Set/Get*. Como se ha indicado, todas tienen un valor por defecto excepto *input*, por lo que la clase puede usarse indicando únicamente la imagen tensorial de entrada.

7.6.3 Método GetOutput()

El método *GetOutput()* es el encargado de procesar los datos de entrada, generar los glifos y devolverlos al usuario. Los pasos que sigue la función son los siguientes:

1. Creación de la fuente. Se emplea la clase *vtkGlyphSource2D*, y el tipo de glifo escogido es “diamante” (*diamond* en inglés), que genera un cuadrado con sus diagonales orientadas según los ejes X e Y.
2. Cálculo del número de puntos de la imagen, a partir de las dimensiones de la misma en X e Y.
3. Se utiliza un bucle para recorrer los píxeles de la imagen como una cuadricula. El bucle realiza las siguientes tareas:

- a. Obtener el índice del píxel siguiente, y extraer dicho píxel (tensor) de la imagen.
 - b. Calcular los autovectores y autovalores del tensor. La clase *StrainTensor* dispone de un método para ello.
 - c. Calcular la posición del glifo y trasladarlo al punto calculado. La posición se determina a partir del índice del píxel y las características de la imagen, y la traslación se aplica con la clase *vtkTransform*, que se va a encargar también de las demás transformaciones geométricas.
 - d. Rotar el glifo. Para ello, se calcula el ángulo del primer autovector respecto al eje X.
 - e. Modificar la forma del glifo, en función de sus autovalores, tal y como se explica en [31].
 - f. Aplicar el factor de escala, si la opción está activada.
 - g. Determinar qué escalar se ha elegido para el color, y calcular su valor.
4. Comprobar si el usuario ha introducido un conjunto de puntos donde dibujar glifos. En ese caso, se recorre un bucle similar al anterior con la diferencia de que, en este caso, el tensor debe interpolarse en cada punto. Para la interpolación se emplea el método presentado en [36].
 5. Se reserva espacio de memoria suficiente para todas las celdas, y se insertan las celdas de todos los glifos en el objeto de salida.
 6. Se añaden al objeto de salida los nuevos puntos, escalares y normales creados.

7.6.4 Uso de la clase

La clase *vtkTensorGlyphStrain* se sitúa al principio del pipeline de VTK, al servir de puente entre la imagen tensorial de Saturn, de ITK, y la visualización con VTK.

El uso de la clase es sencillo. Después de crear una instancia de la clase, se debe usar el método *SetInput()* para indicar la imagen tensorial de entrada, y los

métodos *SetPlanoZ()* y *SetTiempo()* para indicar el corte y el instante temporal a visualizar. Estos dos parámetros tienen un valor por defecto, cero, pero es recomendable especificarlos. Además se pueden modificar el tipo de coloreado o el factor de escala con los métodos Set correspondientes.

A continuación se debe realizar una llamada al método *GetOutput()*. Este método devuelve un puntero a un objeto del tipo *vtkPolyData*, con el que se puede completar el resto del pipeline (*mapper*, *actor*, etc.) para visualizar los glifos.

7.7 Métodos en TensorConsole

La interfaz cuenta con tres métodos, encargados de gestionar la visualización del tensor de esfuerzo, que sirven de nexo entre las decisiones del usuario y su efecto en la visualización.

El primero de estos métodos es *ViewStrainSlice3D()*, y su función es generar un mapa de escalares a partir de los tensores y mostrarlo en el visor. En primer lugar se extrae de la imagen tensorial 4D una imagen tridimensional, correspondiente al instante de tiempo indicado por el usuario en la interfaz. Esta imagen 3D se pasa por un filtro (del tipo *ComputeStrainScalars*, comentado anteriormente), que convierte la imagen tensorial en una imagen de escalares, según el escalar que haya escogido el usuario. Por último, esta imagen se le pasa al visor 3D a la vez que se le indica el corte que se desea visualizar.

El segundo métodos lleva el nombre de *verGlifosStrain()*, y se encarga de mostrar los glifos de tensor de esfuerzo, para lo se utiliza la clase *vtkTensorGlyphStrain*. Como se ha comentado anteriormente, esta clase es sencilla de usar, y el cometido de la función es pasarle a la clase la imagen tensorial y los parámetros indicados por el usuario para obtener el volumen de salida, que es visualizado por el visor 3D.

El tercer método, *borrarGlifosStrain()*, se encarga de eliminar los objetos que representan a los glifos. Este método se utiliza cuando se quieren ocultar los glifos que se están visualizando actualmente, o sustituirlos por otros diferentes.

7.8 Interfaz

La introducción de la imagen por tensor de esfuerzo entre las posibilidades de Saturn ha requerido añadir una serie de elementos a su interfaz. Estos nuevos elementos permiten cargar y guardar, manejar y visualizar las imágenes de tensor, y para facilitar su uso y compatibilidad son similares a los ya existentes en la interfaz para DTI.

La Figura 7.1 muestra el aspecto de la nueva interfaz para tensor de esfuerzo. Esta interfaz contiene un navegador para manejar los datos cargados, un panel de configuración y un visor 3D, además de varios elementos de menú nuevos. Esta vista aparece al cargar una imagen de tensor de esfuerzo o al pulsar el botón *Strain* en la parte superior derecha de la interfaz.

En el área de datos (zona superior izquierda) se muestra una lista con los volúmenes cargados. Esta zona permite visualizar un conjunto de datos concreto, modificar su nombre o eliminarlo.

El panel de configuración aparece en la parte inferior izquierda, y su función por ahora es la de escoger el escalar por el que se van a colorear los glifos de tensor de esfuerzo. Las opciones son el invariante del tensor, sus elementos o sus autovalores.

El visor 3D es la parte más extensa de la interfaz, situada en la parte derecha. En esta zona se van a visualizar los tensores de dos formas. Por un lado, un widget permite mostrar un mapa de escalares representando alguna característica del tensor (el invariante, por defecto). Por otro, se representan glifos mostrando las direcciones principales del esfuerzo y sus magnitudes.

Este visor cuenta con una serie de controles para manejar la vista. El botón *Mostrar* activa y desactiva la visualización de tensores. Las barras de desplazamiento *Slice* y *Time* permiten elegir qué plano y qué instante de tiempo se desea visualizar. Los botones *Color* y *Gray* permiten incluir o eliminar el color. Los botones *Zoom +* y *-* permiten acercar o alejar la imagen. El deslizador *Opacity* controla la opacidad de la imagen, y el botón *Center* ajusta la vista para que la imagen aparezca en el visor entera y centrada.

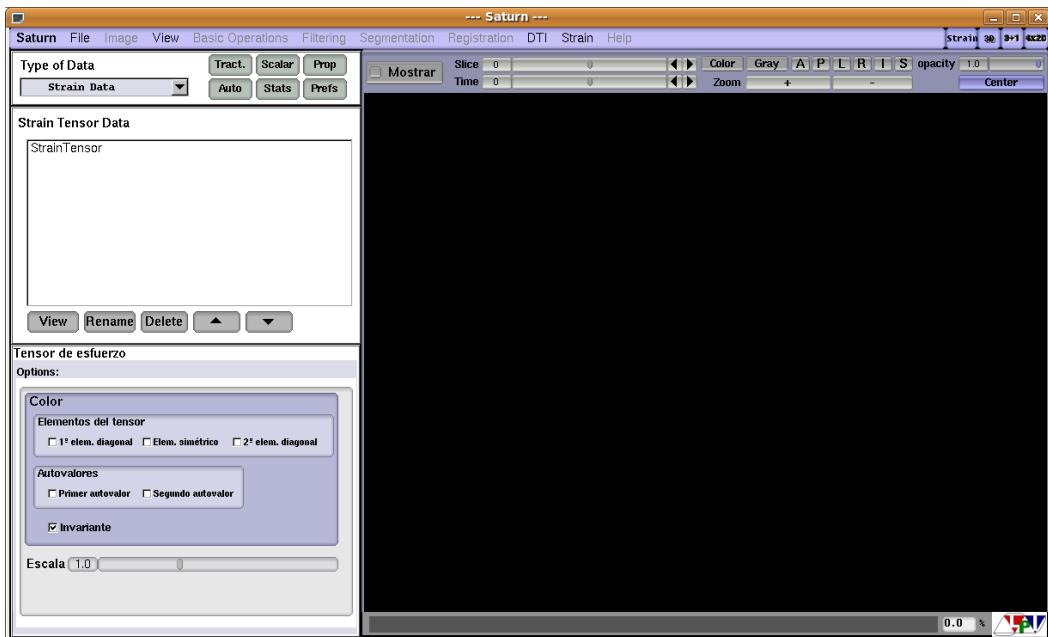


Figura 7.1. Aspecto de la interfaz para tensor de esfuerzo

7.8.1 Uso de la interfaz de usuario

Para visualizar una imagen de tensor de esfuerzo se deben seguir los siguientes pasos:

1. Abrir la aplicación Saturn, acudir al menú *File* y seleccionar la opción *Open Strain Tensor*.
2. En el explorador de archivos, navegar hasta el directorio donde se encuentran los ficheros de tensor de esfuerzo, y abrir el que tenga el número 00. Por ejemplo, *StrainTensor00.vtk*. La interfaz cambiará para mostrar el visor de tensor de esfuerzo.
3. Seleccionar el botón *Mostrar* en la parte superior. Aparecerá la primera visualización en el visor 3D.
4. Utilizar los deslizadores para cambiar el corte o el instante de tiempo mostrados. Utilizar el resto de controles para modificar la vista (p.ej. zoom) y el panel de configuración para cambiar el tipo de coloreado.

Referencias

- [1] Kak AC, Slaney M. Principles of computerized tomographic imaging. New York, IEEE NY, 1988, pp. 77–86.
- [2] Slichter CP. Principles of Magnetic Resonance. Springer, 1990.
- [3] A. Fick, Über Diffusion Annalen der Physik und Chemie von J.C. Poggendorff 94, 59 (1855).
- [4] Schaefer PW, Grant PE, Gilberto Gonzalez R, Diffusion-weighted MR Imaging of the Brain, *Radiology* 217 (2000), pp. 331-345.
- [5] P.J. Basser and D.K. Jones, Diffusion-tensor MRI: theory, experimental design and data analysis—a technical review, *NMR Biomed* 15 (2002), pp. 456–467.
- [6] Torrey HC. Bloch equations with diffusion terms. *Phys. Rev.*, 104(3) (1956), pp. 563-565.
- [7] Stejskal EO, Tanner JE. Spin diffusion measurements: spin echoes in the presence of a time-dependent field gradient. *J. Chem.Phys.* 42 (1965), pp. 288–292.
- [8] Westin C., Maier S., Mamata H., Nabavi A., Jolesz F., and Kikinis R. Processing and visualization for diffusion tensor MRI. *Medical Image Analysis*, 6(2) (2002), pp. 93–108.
- [9] Basser P, Pierpaoli C. Microstructural and physiological features of tissues elucidated by quantitative-diffusion-tensor MRI. *J. Magn. Reson. Ser. B* 111 (1996), pp. 209–219.
- [10] Westin CF, Peled S, Gudbjartsson H, Kikinis R, Jolesz F. Geometrical diffusion measures for MRI from tensor basis analysis. *ISMRM 1997*, Vancouver, Canada, p. 1742.
- [11] Goldberg-Zimring D, Mewes AUJ, Maddah M, Warfield SK. Diffusion Tensor Magnetic Resonance Imaging in Multiple Sclerosis. *J Neuroimaging*, col. 15, no. s4 (2005), pp. 68S-81S.
- [12] Sundgren PC, Dong Q, Gómez Hassan D, Mukherji SK, Maly P, Welsh R. Diffusion tensor imaging of the brain: review of clinical applications. *Neuroradiology*, vol.46 (2004), pp. 339-350.
- [13] Kubicki M, Westin CF, McCarley RW, Shenton ME. The application of DTI to Investigate White Matter Abnormalities in Schizophrenia. *Ann. N.Y. Acad. Sci.*, vol. 1064 (2005), pp. 134-148.
- [14] Talos IF, O'Donnell L, Westin CF, Warfield SK, Wells III W, Yoo SS et al. Diffusion Tensor and Functional MRI Fusion with Anatomical MRI for Image Guided Neurosurgery. *Med. Image Comput. Comput.-Assisted Interv.* (MICCAI 2003), pp. 407-415, Montréal, Canada.
- [15] M. Filippi, M. Cercignani, M. Inglesi, M. A. Horsfield and G. Comi , Diffusion tensor magnetic resonance imaging in multiple sclerosis. *Neurology* 56 (2001), pp. 304–311.

- [16] Kindlmann G. Superquadric Tensor Glyphs. *Proc IEEE TVCG/EG Symp Vis* (2004), pp. 147-154.
- [17] C.-F. Westin, A tensor framework for multidimensional signal processing. *Dissertation 348, Linköping Studies in Science and Technology, Linköpings Universitet, Linköping, Sweden*, 1994.
- [18] V. Arsigny, P. Fillard, X. Pennec and N. Ayache, Log-Euclidean metrics for fast and simple calculus on diffusion tensors, *Magnetic Resonance in Medicine*. **56** (2) (2006), pp. 411–421
- [19] G. Kindlmann and C.-F. Westin, Diffusion Tensor Visualization with Glyph Packing, *IEEE Trans. Visualization and Computer Graphics*, vol. 12, no. 5 (Sept.-Oct. 2006), pp. 1329-1336.
- [20] P. Basser, S. Pajevic, C. Pierpaoli, J. Duda, and A. Aldroubi, In vivo fiber tractography using DT-MRI data, *Magnetic Resonance in Medicine*, **44** (2000), pp. 625–632
- [21] Guttman M.A., E.A. Zerhouni, and E.R. McVeigh. Analysis of cardiac function from MR images, *IEEE Comput. Graphics Applic.* (1997), pp. 30-38.
- [22] Ozturk, C., J.A. Derbyshire, and E.R.M. McVeigh, Estimating motion from MRI data. *Proceedings of the IEEE* 91(10) (2003), pp. 1627-1648.
- [23] Zerhouni E, et al. Human Heart: Tagging with MR Imaging—A Method for Noninvasive Assessment of Myocardial Motion. *Radiology* 169 (1) (1988), pp. 59–63.
- [24] Axel L, Dougherty L. MR Imaging of Motion with Spatial Modulation of Magnetization. *Radiology* 171 (3) (1989), pp. 841.
- [25] Bryant DJ, Payne JA, Firmin DN, Longmore DB. Measurement of flow with NMR imaging using a gradient pulse and phase difference technique. *J. Comput. Assist. Tomog* 8 (2984), pp. 588–593
- [26] Suryan G. Nuclear resonance in flowing liquids. *Proc. Indian Acad. Sci* (1951), pp.33:107
- [27] B. Wuensche and R. Lobb. A toolkit for the visualization of stress and strain tensor fields in biological tissue, *Proceedings of VIP* (1999).
- [28] A. Bajo, MJ. Ledesma Carbayo, C. Santa Marta, E. Pérez David, MA. García Fernández, M. Desco, A. Santos, Cardiac motion analysis from magnetic resonance imaging: cine magnetic resonance versus tagged magnetic resonance, *Computers in Cardiology* 34 (2007), pp. 81-84.
- [29] Ledesma MJ, Bajo A, Santa Marta C, Pérez E, Caso I, García MA, Santos A, Desco M. Cardiac Motion Analysis From Cine MR Sequences using Non-Rigid Registration Techniques. *Comput. Cardiol.*, vol33, pp. 65-68, 2006.
- [30] G. Vegas-Sánchez-Ferrero, A. Tristán Vega, L. Cordero Grande, P. Casaseca de la Higuera, S. Aja Fernández, M. Martín Fernández, C. Alberola López, Strain rate tensor estimation in cine cardiac MRI based on elastic image registration, *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops* (2008).
- [31] G. Vegas-Sánchez-Ferrero, A. Tristán Vega, L. Cordero Grande, S. Merino Caviedes, P. Casaseca de la Higuera, M. Martín Fernández, Estimación robusta y visualización densa del tensor de strain cardiaco en MRI, XXVI

- Congreso Anual de la Sociedad Española de Ingeniería Biomédica. Valladolid, España, Octubre de 2008.*
- [32] Wuensche, B. The visualization of 3D stress and strain tensor fields. *Proceedings of the 3rd New Zealand Computer Science Research Student Conference, University of Waikato, Hamilton, New Zealand, Abril de 1999*, pp. 109-116.
 - [33] Maurice R, Daronat M, Ohayon J, Stoyanova E, Foster F, Cloutier G. Non-invasive high frequency vascular ultrasound elastography. *Phys. Med. Biol.*, vol. 50, pp. 1611-1628, 2005.
 - [34] Krouskop T, Wheeler T, Kallel F, Garra B, Hall T. Elastic moduli of breast and prostate tissue under compression. *Ultrason. imaging*, vol. 20, pp. 260-274, 1998.
 - [35] P. Selskog, E. Heiberg, T. Ebbers, L. Wigström, M. Karlsson, Kinematics of the heart: strain-rate imaging from time-resolved three-dimensional phase contrast MRI, *IEEE Transactions on Medical Imaging*, vol. 21, no. 9 (2002).
 - [36] S. Merino Caviedes, M. Martín Fernández, A general interpolation method for symmetric second-rank tensors in two dimensions, *International Symposium on Biomedical Imaging, Paris, France*, Mayo de 2008.
 - [37] Absjorn Stoylen, Strain rate imaging of the left ventricle by ultrasound. Feasibility, clinical validation and physiological aspects.
 - [38] Constantine G, Shan K, Flamm SD, Sivananthan MU. Role of MRI in clinical cardiology. *The Lancet* 363 (2004), pp. 2162–2171.
 - [39] Sutherland GR, Di Salvo G, Claus P, D'hooge J, Bijnens B., Strain and strain rate imaging: a new clinical approach to quantifying regional myocardial function, *Journal of the American Society of Echocardiography* 17 (2004), pp. 788-802.
 - [40] Luis Ibáñez, Will Schroeder, Lydia Ng, Josh Cates and the Insight Software Consortium. The ITK Software Guide. Segunda edición, Noviembre de 2005.
 - [41] Insight Segmentation and Registration Tool (ITK). <http://www.itk.org/>. Online: Julio de 2010.
 - [42] Schroeder W.J., Martin K, Lorensen B, The Visualization Toolkit, An Object-Oriented Approach to 3D Graphics. Tercera edición (2002).
 - [43] Schroeder W.J., Avila L.S., Hoffman W., Visualizing with VTK: A Tutorial. *IEEE Computer Graphics and Applications*, vol. 20, no. 5, pp.20–27, 2000
 - [44] The Visualization Toolkit (VTK). <http://www.vtk.org/>. Online: Julio de 2010.
 - [45] Constantini F, Gibson D, Melcher M, Schlosser A, Spitzak B, Sweet M. *FLTK Programming Manual*. Abril de 2010.
 - [46] Fast Light Toolkit (FLTK). <http://fltk.org/>. Online: Julio de 2010.
 - [47] CMake, the cross-platform, open-source build system. <http://www.cmake.org/>. Online: Julio de 2010
 - [48] R. Cárdenas Almeida, UsimagTool: a tool for ultrasound images visualization and processing.
 - [49] R. Cárdenas Almeida, A. Tristán Vega, G. Vegas Sánchez-Ferrero, S. Aja Fernández, V. García Pérez, E. Muñoz Moreno, R. de Luis García, J. González Fernández, D. Sosa Cabrera, K. Krissian, S. Kieffer, UsimagTool: an sopen

- source freeware software for ultrasound imaging and elastography, *Proceedings of the eINTERFACE'07 Workshop on Multimodal Interfaces, Istanbul, Turkey*, Julio-Agosto de 2007.
- [50] UsimagTool Software. Laboratorio de Procesado de Imagen (LPI) <http://www.lpi.tel.uva.es/usimagtool/>
 - [51] S. Merino Caviedes, M. Martín Fernández, User Interfaces to Interact with Tensor Fields.
 - [52] 3D Slicer. Medical Visualization and Processing Environment for Research. <http://www.slicer.org/>. Online: Junio 2010.
 - [53] P. Fillard, N. Toussaint, X. Pennec, MedINRIA: DT-MRI processing and visualization software, *Similar NoE Tensor Workshop, Las Palmas*, Noviembre de 2006.
 - [54] MedINRIA. Asclepios Research Project - INRIA Sophia Antipolis <http://www-sop.inria.fr/asclepios/software/MedINRIA/>. Online: Junio 2010.