UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

OPTIMIZACIÓN COMPUTACIONAL DE ALGORITMOS DE ANÁLISIS DE BIOMARCADORES DE IMAGEN DE RESONANCIA MAGNÉTICA MEDIANTE MODELOS DINÁMICOS EN ONCOLOGÍA

Alejandro Torres Alberola

Tutor: Ignacio Bosch Roig

Cotutor: Amadeo Ten Esteve

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2018 - 2019

Valencia, 3 de julio de 2019

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universitat Politècnica de València
Edificio 4D. Camino de Vera, s/n, 46022 Valencia
Tel. +34 96 387 71 90, ext. 77190
www.etsit.upv.es

VLC/ CAMPUS
VALENCIA, INTERNATIONAL CAMPUS OF EXCELLENCE

EMAS
Gestión ambiental verificada
REG.NO. ES-CV-000030

_ TELECOM ESCUELA
TÉCNICA VLC SUPERIOR
DE UPV INGENIEROS DE
TELECOMUNICACIÓN

Resumen

El trabajo se ha realizado en un marco colaborativo con el Grupo de Investigación Biomédica en Imagen (GIBI230) del Hospital Universitario y Politécnico La Fe. Parte de un desarrollo previo de algoritmos para la extracción de biomarcadores de imagen en el ámbito oncológico a partir de imágenes de Resonancia Magnética. En concreto en este trabajo se pretende optimizar computacionalmente aquellos que utilizan el ajuste de curvas, como son los estudios de difusión-perfusión, para reducir su tiempo de ejecución.

El trabajo describe además las diferentes partes del proceso: lectura de imágenes, preprocesado, ajuste de curvas por mínimos cuadrados y extracción de biomarcadores, haciendo especial hincapié en la optimización computacional del proceso de ajuste de curvas, en busca de una reducción en el tiempo de ejecución. Para ello, se ha partido de algoritmos desarrollados en MatLab, donde la tarea computacionalmente más demandante la realiza la función iterativa *Lsqcurvefit*, que ajusta los modelos dinámicos a los datos de forma no lineal mediante mínimos cuadrados. Los resultados obtenidos han sido comparados con el desarrollo original con el fin de confirmar la reducción del tiempo de ejecución y la validez de los resultados.

El grado de éxito final del proyecto viene dado al comparar el tiempo de ejecución original y el alcanzado durante el proyecto, así como el error cometido en el cálculo del biomarcador de imágenes, tomando como referencia los resultados obtenidos con el desarrollo previo.

Resum

El projecte es realitzarà en un marc col·laboratiu en el Grup d'Investigació Biomèdica en Imatge (GIBI230) de l'Hospital Universitari i Politècnic La Fe i part d'un desenvolupament previ d'algoritmes per al processament d'imatges mèdiques per a l'extracció de biomarcadors d'imatge a partir de imatges de Ressonància Magnètica de Difusió i Perfusió en l'àmbit oncològic. Els algoritmes previs han estat ja desenvolupats i implementats, però poden optimitzar-se computacionalment per reduir el seu temps d'execució.

Este projecte se centrarà en realitzar una anàlisi detallada de les diferents parts del procés: lectura d'imatges, preprocessat, ajust de corbes per mínims quadrats i extracció de biomarcadors. L'objectiu serà aconseguir una optimització del procés perseguint una acceleració de l'execució i cometent el menor error possible en els càlculs dels biomarcadors. Per a això, es partirà d'algorismes en MatLab, on la tasca computacionalment més demandant la realitza la funció iterativa *Lsqcurvefit*, que ajusta els models dinàmics a les dades de forma no lineal mitjançant mínims quadrats.

Finalment es comprovarà el grau d'èxit del projecte en comparar el temps d'execució original i l'aconseguit durant el projecte així com l'error comès en l'anàlisi, prenent com a referència els resultats que s'estan obtenint en l'actualitat.

Abstract

The project will be carried out in a collaborative framework with the Grupo de Investigación Biomédica en Imagen (GIBI230) of the Hospital Universitario y Politécnico La Fe and starts from a previous development for the medical image processing for extracting image biomarkers from Magnetic Resonance Imaging of Diffusion and Perfusion in the oncologic domain. The previous algorithms have already been developed and implemented, but can be optimized computationally to reduce their execution time.

This project focuses on a detailed analysis of the different parts of the process: Reading of images, preprocessing, adjustment by least squares curves and extraction of biomarkers. The objective will be to achieve optimization of the process by pursuing an acceleration of execution and making the smallest possible error in the calculations of the biomarkers. To do this, we will start with algorithms in MatLab, where the computationally most demanding task is performed by the *Isqcurvefit* iterative function, which adjusts the dynamic models to the data in a nonlinear way using least squares.

Finally, the degree of success of the project will be checked by comparing the original execution time and the time achieved during the project, as well as the error committed in the analysis, taking as reference the results that are currently being obtained.

Gracias a Amadeo y Nacho, por su apoyo y sobre todo por su paciencia.

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

Índice

Capítulo 1. Introducción 1

1.1. Marco del proyecto....1

1.2. Introducción al biomarcador de imagen 1

1.3. La función Lsqcurvefit....5

1.4. El algoritmo de Levenberg-Marquardt....6

1.4.1. Elección del algoritmo....6

1.4.2. Contexto....6

1.4.3. El algoritmo de Levenberg-Marquardt....6

1.4.4. Elección del parámetro de amortiguación....7

1.4.5. Limitaciones en la búsqueda del mínimo global....7

Capítulo 2. Objetivos 8

Capítulo 3. Materiales y métodos 9

3.1. Gestión del proyecto....9

3.2. Distribución en tareas....9

3.3. Diagrama temporal....10

3.4. Introducción al problema....10

3.5. Camino hacia una solución....10

3.6. Primeras versiones de milevmar....15

3.6.1. Versiones iniciales....15

3.6.2. Versiones de milevmar3 (de la 1 a la 3)....16

3.6.3. Versiones de milevmar3 (de la 4 a la 5)....17

3.6.4. Versiones de milevmar3 (de la 6 a la 10)....18

3.7. Aspectos a considerar acerca del código en la versión final de milevmar3....20

3.7.1. Contador de fallos....20

3.7.2. Variación de λ....21

3.7.3. Resolución del sistema de ecuaciones no lineales....23

3.8. Problemática de las matrices singulares....23

Capítulo 4. Resultados.... 24

4.1. Introducción a las imágenes utilizadas....24

4.2. Prueba 1: imágenes de 36x36 píxeles....25

4.3. Prueba 2: imágenes de 71x71 píxeles....27

4.4. Prueba 3: imágenes de 160x160 píxeles....29

4.5. Prueba 4: imágenes de 293x293 píxeles....32

4.6. Rendimiento de la solución……………………………………………………….34

Capítulo 5. Conclusiones y líneas futuras…………………………………………………36

Bibliografía…………………………………………………………………………………….37

Anexos………………………………………………………………………………..……..38

Capítulo 1. Introducción

Este trabajo se ha planteado como un ejercicio en el que se pretendía optimizar el ajuste de curvas empleado en la obtención de biomarcadores de imágenes basados en difusión y perfusión. Durante el proceso se estudiaron diferentes alternativas, finalmente optándose por una solución que logra reducir el tiempo de ejecución del algoritmo bajo ciertas condiciones (concretamente, con imágenes de tamaño reducido).

A continuación, se van a exponer las partes que son necesarias para comprender el desarrollo de este estudio. En este orden, se explicará qué es un biomarcador y cuál es su utilidad, así como el algoritmo de Levenberg-Marquardt utilizado en el ajuste de las curvas. También se expondrá brevemente en qué consiste la función de MatLab *Lsqcurvefit*, utilizada como referencia a la hora de valorar los objetivos, además de servir como punto de partida a la hora de crear el algoritmo propio, que ha sido llamado *milevmar3*.

1.1. Marco del proyecto

El trabajo se ha realizado dentro de un marco colaborativo con el Grupo de Investigación Biomédica en Imagen (GIBI230) del Hospital Universitario y Politécnico La Fe. Parte de un desarrollo previo de algoritmos para la extracción de biomarcadores de imagen en el ámbito oncológico a partir de imágenes de Resonancia Magnética. En concreto en este trabajo se pretende optimizar computacionalmente aquellos algoritmos que utilizan el ajuste de curvas, como son los estudios de difusión y perfusión.

1.2. Introducción al biomarcador de imagen

Los biomarcadores son indicadores que permiten caracterizar de manera objetiva algún parámetro o característica biológica, ya sean propiedades tisulares o ~~relacionados~~relacionadas con un proceso patológico. Se pueden emplear tanto para diagnóstico como para seguimiento y monitorización, algunos ejemplos de biomarcadores son por ejemplo la presión sanguínea o el volumen de una determinada región.

También permiten diagnósticos menos invasivos de los pacientes, minimizando las intervenciones para tomar muestras y permitiendo toma de decisiones en menor tiempo.

Existen diversos tipos de biomarcadores. Algunos de ellos son bioquímicos (como podrían ser las enzimas o las células extraídas de una muestra de cáncer) mientras que otros pueden ser alteraciones genéticas en el ADN [1]~~.,~~. Los biomarcadores de imagen son aquellos que son extraídos a partir de imágenes médicas, siendo estos precisamente en los que se centra este trabajo.

Para obtener y utilizar los biomarcadores de imagen el paso inicial es la lectura de imágenes. Éstas comúnmente siguen el protocolo DICOM, que es un estándar de comunicación y almacenamiento de imágenes médicas, ampliamente utilizado en hospitales. DICOM permite la integración de hardware muy diferente dentro de una misma red de información, facilitando las comunicaciones [2]. Una característica notable de DICOM es que las imágenes siempre van unidas a los datos del paciente e identificadas con un número único, evitando que se extravíen o confundan la información de los pacientes.

Una vez leído el archivo DICOM, se extrae la imagen y se procesa mediante algoritmos. A partir de imágenes de resonancia magnética, es posible caracterizar los diferentes tejidos mediante los biomarcadores T1 y T2. El origen de ambos biomarcadores se basa en los principios

de la resonancia magnética nuclear, donde se excitan moléculas mediante pulsos de radiofrecuencia (a la frecuencia de resonancia, la frecuencia de Larmor) haciendo que se magneticen, existiendo entonces un intercambio de energía entre los núcleos excitados y el resto del tejido.

En ese contexto, T1 se define como el tiempo necesario para que se recupere el 63% de la magnetización longitudinal (M_z). De esta forma, T1 depende del campo magnético aplicado (que es conocido) y de la estructura molecular del tejido, permitiendo identificar y analizar los tejidos. Por ejemplo, un tejido con una frecuencia de vibración muy cercana a la frecuencia de Larmor tendrá un T1 muy corto, mientras que los tejidos con poca posibilidad de vibrar a la frecuencia de Larmor tendrán un T1 largo [3].

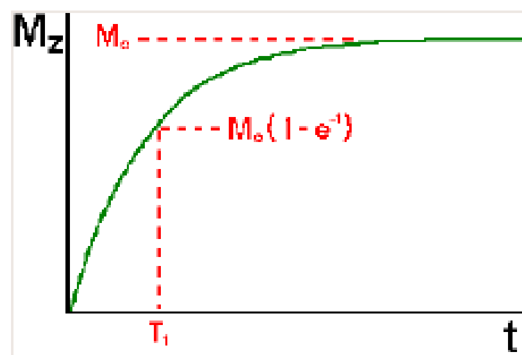

Figura 1. T1, en la aplicación de un pulso de radiofrecuencia de 90º [3]

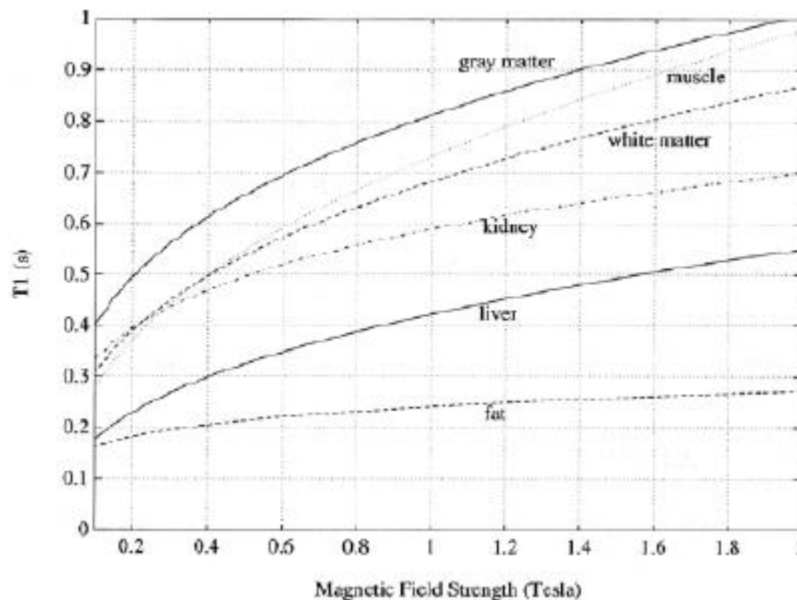

Figura 2. T1 según el campo magnético de diferentes tejidos [3]

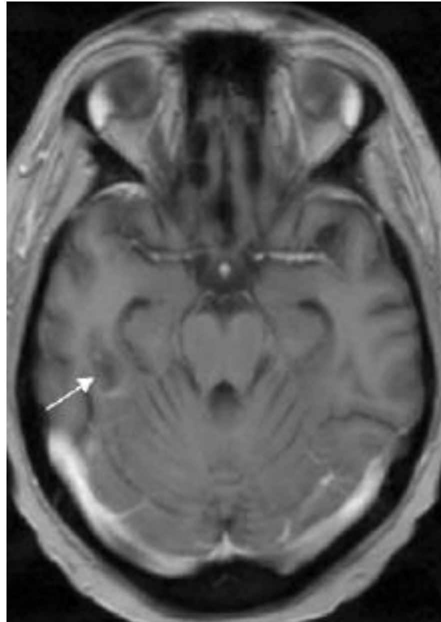

Figura 3. Imagen de una resonancia magnética con ponderación T1 [3]

Por otro lado, el tiempo de relajación transversal resulta ser T2, definido como el instante en el que la magnetización posee el 37% del valor en que cesa el pulso de radiofrecuencia. T2 tiene la particularidad de no depender del campo magnético aplicado, sino solamente del tejido donde se aplica. Esto ocurre porque las inhomogeneidades en la estructura del tejido provocan inhomogeneidades en el campo magnético, habiendo frecuencias de resonancia ligeramente diferentes y causando un desfase entre los spines.

Este trabajo se ocupa precisamente de optimizar algoritmos que buscan extraer el biomarcador T2, mediante una serie de operaciones a partir de algoritmos como el que se ha utilizado en este trabajo.

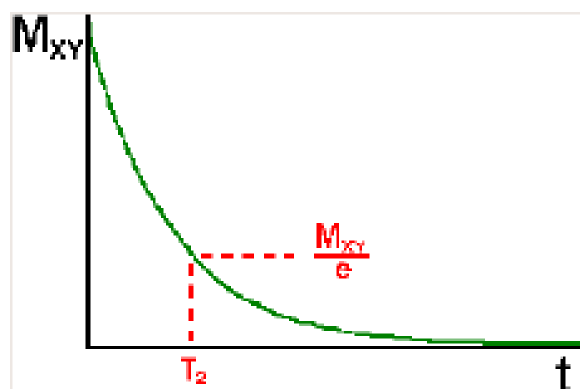

Figura 4. T2, en la aplicación de un pulso de radiofrecuencia de 90º [3]

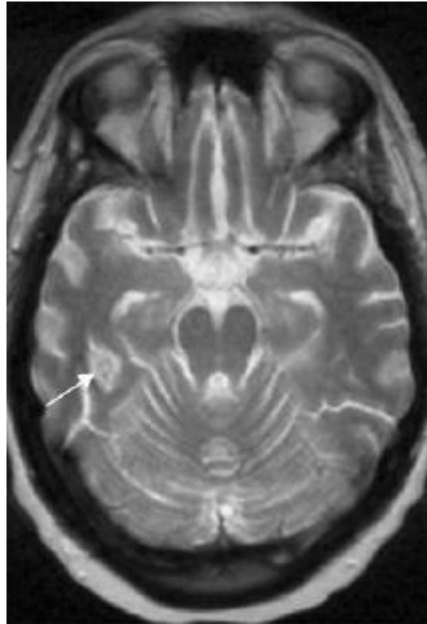

Figura 5. Imagen de una resonancia magnética con ponderación T2 [3]

Materia gris	Materia blanca	Músculo	Grasa	Riñón	Hígado
100	92	47	85	58	43

Tabla 1. T2 (en milisegundos) según el tejido [3]

Ambos tiempos, T1 y T2, son independientes entre sí y ocurren simultáneamente.

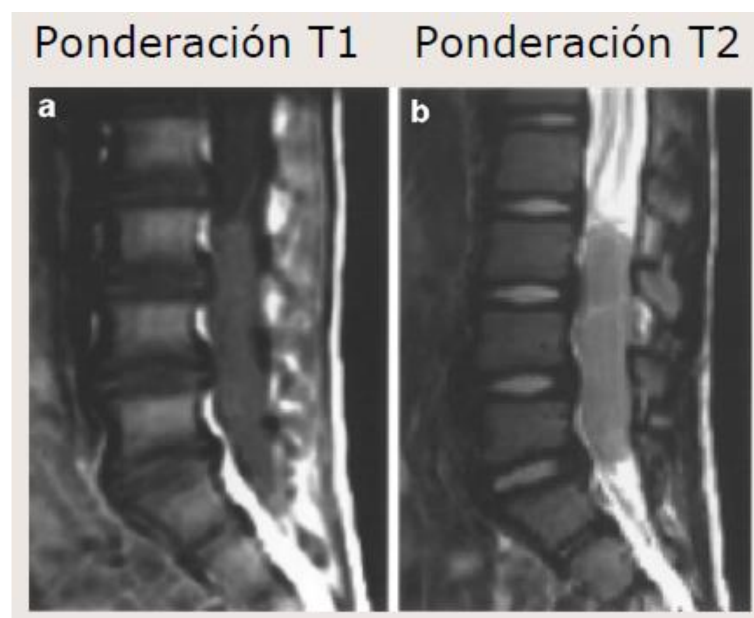

Figura 6. Comparación entre ponderación T1 y T2 en la misma imagen [3]

Figura 7. Imagen sagital de la rodilla en ponderación T2, mostrando la segmentación de la rótula [4]

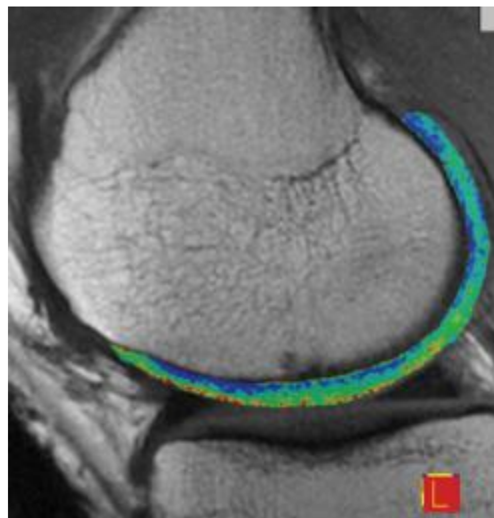

Figura 8. Resonancia magnética multi-eco cuantitativa ponderada en T2. Se muestra un paciente de 35 años, 2 años después de un trasplante de condrocitos autólogos en el cóndilo femoral medial. Se puede observar el tejido reparado. [4]

1.3. La función *Lsqcurvefit*

La función de MatLab *Lsqcurvefit* [5] se utiliza para resolver problemas de ajuste de curvas no lineales, en el sentido de mínimos cuadrados, utilizando el algoritmo de Levenberg-Marquardt (se expondrá en el siguiente apartado). Básicamente, Lsqcurvefit busca los coeficientes x que resuelven la ecuación 1.3.1.:

$$min_x\|F - ydata\|^2 = min_x \sum_i (F_i - ydata_i)^2 \qquad \textbf{(1.3.1.)}$$

Donde F es una función que depende de los coeficientes x y de los datos de entrada *xdata*.

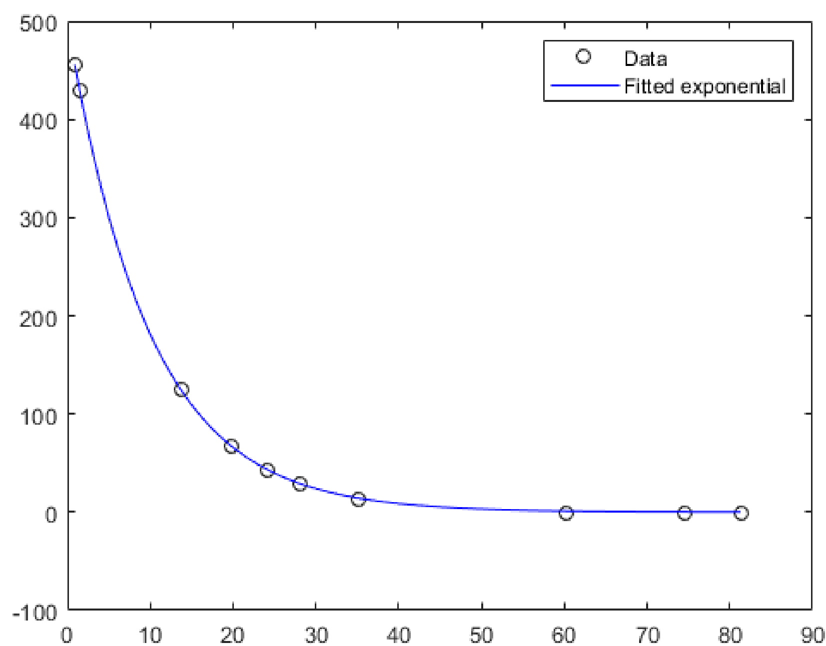

Figura 9. Ejemplo de ajuste de datos experimentales mediante *Lsqcurvefit* [5]

Lsqcurvefit es una parte clave de este trabajo por dos motivos. En primer lugar, porque es la función de referencia a la hora de evaluar los objetivos y el grado de éxito alcanzado. El segundo motivo es que, ya que se desea replicar el funcionamiento de *Lsqcurvefit*, resulta necesario estudiar el algoritmo de Levenberg-Marquardt, que utiliza.

1.4. El algoritmo de Levenberg-Marquardt

1.4.1. Elección del algoritmo

Lsqcurvefit tiene dos algoritmos que puede utilizar: El método de Levenberg-Marquardt y el algoritmo de confianza-región-reflexivo [5]. Ambos métodos devuelven soluciones idénticas, poseyendo cada uno ciertas limitaciones. Dadas las características del problema a resolver, las limitaciones de ambos métodos no restringen en realidad, por lo que no son un motivo para elegir uno sobre otro. Por lo tanto, a igualdad del resto de factores, se elige el método de Levenberg-Marquardt, ya que comúnmente necesita menos iteraciones.

1.4.2. Contexto

Lsqcurvefit, es una función que realiza un ajuste de curvas no lineales en sentido de mínimos cuadrados. Lo que se debe lograr es hallar los coeficientes x que resuelven el problema:

$$min_x \|F - ydata\|^2 = min_x \sum_i (F_i - ydata_i)^2 \quad \textbf{(1.3.1.)}$$

En nuestro caso concreto, *xdata* e *ydata* son las imágenes de entrada y salida, respectivamente, mientras que la función F es parte del proceso de extracción del biomarcador T2:

$$F = ydata_{\ 1} * e^{-xdata * x0_1 * 10^{-3}} \quad \textbf{(1.4.1.2)}$$

1.4.3. Algoritmo de Levenberg-Marquardt

Buscando resolver la ecuación (1.4.1.1.) el algoritmo realiza una serie de iteraciones [6]. Al iniciar, se le debe proporcionar una estimación inicial β. En cada iteración se va variando β, que toma cada vez un nuevo valor β+ *δ*. Para lograr determinar *δ*, se realiza la siguiente aproximación:

$$f(x_i, \beta + \delta) = f(x_i, \beta) + J_i\delta_i \quad \textbf{(1.4.3.1)}$$

Siendo:

$$J_i = \frac{\partial f(x_i, \beta)}{\partial \beta} \ \delta_i \quad \textbf{(1.4.3.2)}$$

La suma de desviaciones cuadradas será llamada S (β). Tendrá un mínimo en el gradiente de cero respecto a β, siendo su aproximación de orden 1 la siguiente:

$$S(\beta + \delta) = \sum_{i=1}^{m} [y_i - f(x_i, \beta) - J_i\delta]^2 \quad \delta_i \quad \textbf{(1.4.3.3)}$$

Tomando la derivada de S (β +δ) con respecto a δ, estableciendo el resultado en cero y añadiendo un término de amortiguación λI (siendo I la matriz identidad y λ el factor de amortiguamiento) obtenemos:

$$(J^T J + \lambda\mathrm{I})\delta = J^T[y - f(\beta)] \quad \textbf{(1.4.3.4)}$$

Este algoritmo presenta problemas en situaciones donde λ tome un valor grande. En el caso que nos ocupa, esto resulta bastante común, ya que el factor de amortiguamiento λ varía en muchos órdenes de magnitud durante el proceso iterativo. Para solucionar este problema, se realiza otra modificación (donde se sustituye la matriz identidad):

$$[(J^T J + \lambda \, \mathrm{diag}(J^T J\,)]\delta = \; J^T\,[y - f(\,\beta)]\,[y - f(\,\beta)] \qquad \textbf{(1.4.3.5)}$$

1.4.4. Elección del parámetro de amortiguación

No es posible demostrar teóricamente qué valor de λ es adecuado a todas las situaciones. Sí que resulta posible afirmar que ciertos λ conseguirán que el método converja, pero a costa de efectos indeseados como una convergencia demasiado lenta. En la práctica, cada problema determina la elección de su λ más adecuado.

1.4.5. Limitaciones en la búsqueda del mínimo global

El algoritmo de Levenberg-Marquardt, igual que el de confianza-región-reflexivo, solamente ~~encuentran~~encuentra un mínimo local, sin garantizar que sea un mínimo global. Es posible introducirle una aproximación inicial β vectorial con varios valores, de forma que se calculen varios mínimos potencialmente diferentes, pero que igualmente sigue sin garantizar que el mínimo global sea hallado.

Capítulo 2. Objetivos

El problema principal del que se parte es el alto tiempo de ejecución del algoritmo utilizado para extraer los biomarcadores (concretamente, el biomarcador T2) de imágenes de resonancia magnética. Si lo que se pretendía era emplear el algoritmo para conseguir diagnósticos más rápidos y eficientes, el alto coste computacional del mismo (que se traduce en un coste en tiempo) provoca que la su utilidad a la hora de realizar los estudios se vea limitada por el coste en tiempo requerido.

El algoritmo tiene como elemento más demandante las llamadas a la función de MatLab *Lsqcurvefit* (expuesta en el punto 1.3 de este documento) que realiza ajustes de curvas no lineales por mínimos cuadrados. MatLab utiliza un lenguaje propio que no requiere compilador ni ensamblador, sino que utiliza un intérprete. Esta cualidad le da a MatLab flexibilidad, pero al mismo tiempo provoca cierta lentitud en su ejecución al tener que realizar un proceso de traducción para las instrucciones antes de ejecutarlas. Conociendo esta limitación, y teniendo el tiempo como un factor decisivo, es buena idea tratar de realizar el mismo algoritmo (que ya ha sido probado y demostrado su buen funcionamiento) en otro lenguaje que permita ejecuciones más rápidas, como lo es el lenguaje C.

El objetivo principal de este trabajo es optimizar el tiempo de ejecución del ajuste de curva que realiza el algoritmo con una llamada a la función *Lsqcurvefit* de MatLab, utilizando para ello un algoritmo equivalente pero más veloz. El éxito se medirá en función de lo veloz que sea la nueva solución en comparación con *Lsqcurvefit*.

Durante el trabajo se valoraron diversas alternativas, optándose finalmente por una solución que, bajo ciertas condiciones, logra reducir el tiempo y cumplir el objetivo bajo esas condiciones limitadas (que resulta ser el tamaño de la imagen).

Como objetivo secundario, este trabajo trata de lograr el mínimo error de cálculo posible, tomando como referencia los resultados obtenidos ejecutando el algoritmo con la función de MatLab *Lsqcurvefit* sin realizar ninguna modificación. Por lo tanto, no se trata de lograr un ajuste por curvas más preciso que el que realiza Lsqcurvefit, sino un ajuste que, obteniendo el mismo resultado, consuma una menor cantidad de tiempo.

Este trabajo también tuvo objetivos intermedios, necesarios para el desarrollo, pero no presentes en la solución final. Estos objetivos fueron los algoritmos diseñados para realizar una transición progresiva hacia el código final. Comenzando por un ajuste de curvas lineales con vectores, pasando por curvas no lineales y posteriormente matrices, para finalmente dar el salto al uso de imágenes.

Capítulo 3. Materiales y métodos

Este trabajo fue planteado para ser realizado entre noviembre de 2018 y junio de 2019, planificando las tareas siguiendo un orden cronológico. De todas las tareas a realizar, la más significativa prevista era la realización del código final que realizase el ajuste de curvas con imágenes, y a ella se le reservó la mitad del tiempo total disponible. El resto de tareas, más previsibles y acotadas en el tiempo, se planificaron para dejar la mayor cantidad posible de tiempo a la tarea principal de realizar el código final, teniendo en cuenta que previsiblemente la mayor parte de contratiempos e imprevistos ocurrirían en esa fase.

3.1. Gestión del proyecto.

El proyecto se planteó inicialmente como una búsqueda de información, seguida por una transición progresiva desde un código que funcionara solo con vectores hacia uno que funcionara con imágenes. Se consideró que este paso, al ser gradual, podría evitar que los problemas se acumulasen en la fase final, resolviendo algunos de ellos previamente.

3.2. Distribución en tareas

- *Investigación y recopilación de información*
  - Esta tarea consiste principalmente en la consulta de los manuales del software utilizado, así como búsquedas por Internet y adquisición de la bibliografía.
- *Preparación del software*
  - Se busca lograr hacer funcionar todo el software necesario, debido a que se necesitan varios programas actuando conjuntamente y de forma coordinada.
- *Realización de la 1º versión del código (unidimensional)*
  - Se requiere un código MEX capaz de realizar ajuste por mínimos cuadrados con ecuaciones no lineales, pero solamente utilizando vectores.
- *Realización de la versión bidimensional*
  - Se actualiza la versión anterior, buscando que pueda ejecutarse sobre matrices.
- *Producción y optimización del código final para imágenes*
  - Tarea más demandante y exhaustiva. Partiendo de la versión para matrices, se pasará a usarse sobre imágenes, buscando alcanzar finalmente los resultados deseados en el proyecto.
- *Recopilación de resultados*
  - Se diseñan una serie de pruebas para comprobar el funcionamiento del código creado y permitir que se pueda analizar su comportamiento.
- *Redacción y revisión del documento*
  - Fase final donde se crea el documento donde se expondrá todo lo realizado en el proyecto.

3.3. Diagrama temporal

Fecha de inicio: 01/11/2018

Fecha final: 25/6/2019

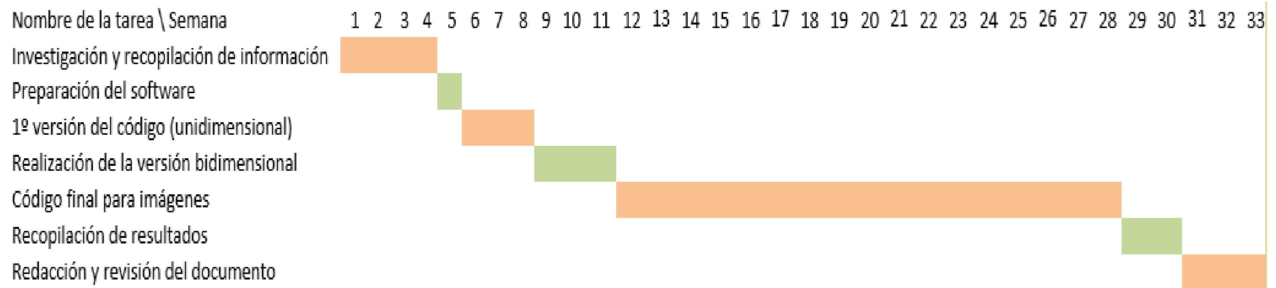

Figura 10. Diagrama temporal

3.4. Introducción al problema

La primera aproximación al problema fue la de reescribir el código existente en MatLab a un lenguaje más desarrollado y rápido como es el lenguaje C. Posteriormente, al analizar el código en MatLab, se comprobó que, con la excepción de las llamadas a la función *Lsqcurvefit*, el código en MatLab no era computacionalmente demandante. Siendo así, se pensó que era preferible no rehacer algo que ya funciona correctamente, centrando los esfuerzos en mejorar la velocidad de ejecución de la función *Lsqcurvefit*.

Durante el proceso de investigación, se llega a la conclusión de que MatLab ya posee una herramienta precisamente para situaciones donde se necesita mejorar la velocidad de un código sin salir del entorno de MatLab. Esta herramienta son los de archivos MEX, que permiten ejecutar llamadas a subrutinas escritas en C, C++ o Fortan desde un archivo MEX. Así resulta posible tener un programa de MatLab donde el intérprete demanda la ejecución del código externamente (ejecutándolo como C en el caso de este trabajo) obteniendo un incremento en la velocidad, devolviendo la respuesta al entorno de MatLab.

Debido a lo anterior, se opta por realizar *Lsqcurvefit* como un archivo MEX.

3.5. Camino hacia una solución

La primera aproximación a los archivos MEX fue realizando un código de prueba en C, convirtiéndolo en MEX y ejecutándolo en MatLab. La cantidad de errores al convertir el archivo de C a MEX era grande y constante, siendo más difíciles de solucionar de lo habitual debido a la forma en la que se expresaban los errores.

Los problemas iniciales fueron consistieron en incompatibilidades entre los diferentes programas necesarios para hacer funcionar MatLab (versión del año 2018) con archivos MEX, ya que se requiere un compilador compatible con MatLab y los MEX. Después de tratar de utilizar varios compiladores, se escoge el "MinGW", versión 6.3 para C/C++. Otros compiladores, como el Microsoft Visual C++, pcc 1.1.0, GCC 8.1 y el lcc-win64, dieron problemas en alguna parte del proceso y se descartaron.

Utilizando el compilador escogido, el MinGW, se encontraron errores inesperados. Por ejemplo, al multiplicar dos matrices de tamaño 10x10 ambas, el compilador expresaba errores informando del tamaño no válido de las matrices. Esta misma operación compilada en C no daba

ningún tipo de problema y devolvía la solución correcta, pero sin embargo al convertir el archivo de C a MEX ya no era válida. La dificultad a la hora de resolver este tipo de problemas al crear los archivos MEX consiste en que, así como un compilador de C o un programa de MatLab informa de los errores y resulta de gran ayuda, el compilador MinGW al crear los archivos MEX es excesivamente escueto o entrega respuestas desconcertantes.

Se intentó utilizar entonces código libre para solucionar el problema, ya que existen librerías como Clapack que poseen archivos C y MEX que realizan el algoritmo de Levenberg-Marquardt. El problema fue que esos archivos están muy desactualizados (la mayoría con 15 años de antigüedad o más) teniendo por lo tanto muchos problemas de compatibilidad con MatLab y el compilador. La cuestión del tiempo es especialmente significativa debido a que MatLab cambió los comandos y las funciones de los MEX después de que esas librerías aparecieran. Por lo tanto, esas librerías funcionarían solamente con versiones de MatLab antiguas.

Adaptar el código antiguo a los nuevos requisitos de MatLab resultó similar a rehacer el código por completo, ya que muchas funciones cambiaban o directamente desaparecían.

La nueva opción explorada fue realizar el código de *Lsqcurvefit* en MatLab, convertir ese código a C de forma automatizada utilizando herramientas de MatLab, y desde ahí convertirlo en un archivo MEX. De esta forma, las incompatibilidades de C a MatLab desaparecen al partir el código del propio MatLab.

Aparecen entonces algunas limitaciones:

- La función *Lsqcurvefit* posee su algoritmo integrado en MatLab y no es posible convertirla a C ni a MEX de forma directa. Se debe hacer un nuevo código.
- No es posible convertir de MatLab a C si en el código hay funciones que poseen parámetros de entrada funciones. *Lsqcurvefit*, sin contar su algoritmo, posee múltiples llamadas a funciones que poseen como entrada otras funciones. Esto obliga a tener que realizar una a una cada función que llame ~~*Lsqcurfefit*~~*Lsqcurvefit*.

Finalmente se opta por realizar el código en MatLab que replique el comportamiento de la función *Lsqcurvefit*.

Aparecieron entonces varios problemas derivados de la misma cuestión: Los mínimos relativos. Es necesario que la función creada sea capaz de buscar activamente una solución con un error cuadrático acorde a las necesidades demandadas. Es muy común que en esa búsqueda se llegue a un mínimo, detectándose que las iteraciones no consiguen reducir el error. Si ese mínimo resulta ser un "valle", anterior a otro aún menor, entonces estamos ignorando soluciones mejores. El caso de la figura inferior (figura 11) es muy habitual: Se encuentra pronto un "valle" con el primer mínimo local (mínimo local 1). Si se detuviera al encontrar la siguiente "colina" no encontraría el mínimo local 2, y tampoco el supuesto "mínimo global" detrás de la siguiente "colina". En el problema que nos ocupa, donde el eje horizontal se corresponde con las iteraciones, todo se complica al no estar acotada la gráfica por la derecha, lo que causa que nunca se tenga la certeza de si el mínimo encontrado es global o solo local. Esto último obliga a que se deba limitar el número de iteraciones o se elija una tolerancia máxima a partir de la cual se considera que los resultados son adecuados.

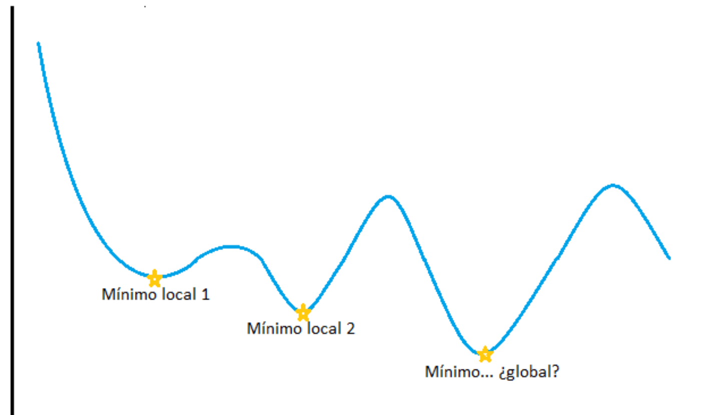

Figura 11. Situación habitual en la búsqueda de mínimos globales

La primera solución que se adoptó es permitir un mayor número de iteraciones, y que el algoritmo, al encontrar un mínimo, siguiera buscando en resultados menores y mayores. Esta solución fue poco efectiva por varios motivos:

- Requiere demasiadas iteraciones, lo que consume demasiado tiempo.
- Aunque las soluciones aportadas son mejores (de esta manera siempre se encuentra el “valle” donde está la solución más adecuada) no insiste buscando el tiempo suficiente alrededor de la solución, perdiendo precisión en el resultado.

Se modificó el código, optando por otro tipo de estrategia de cara a los mínimos locales:

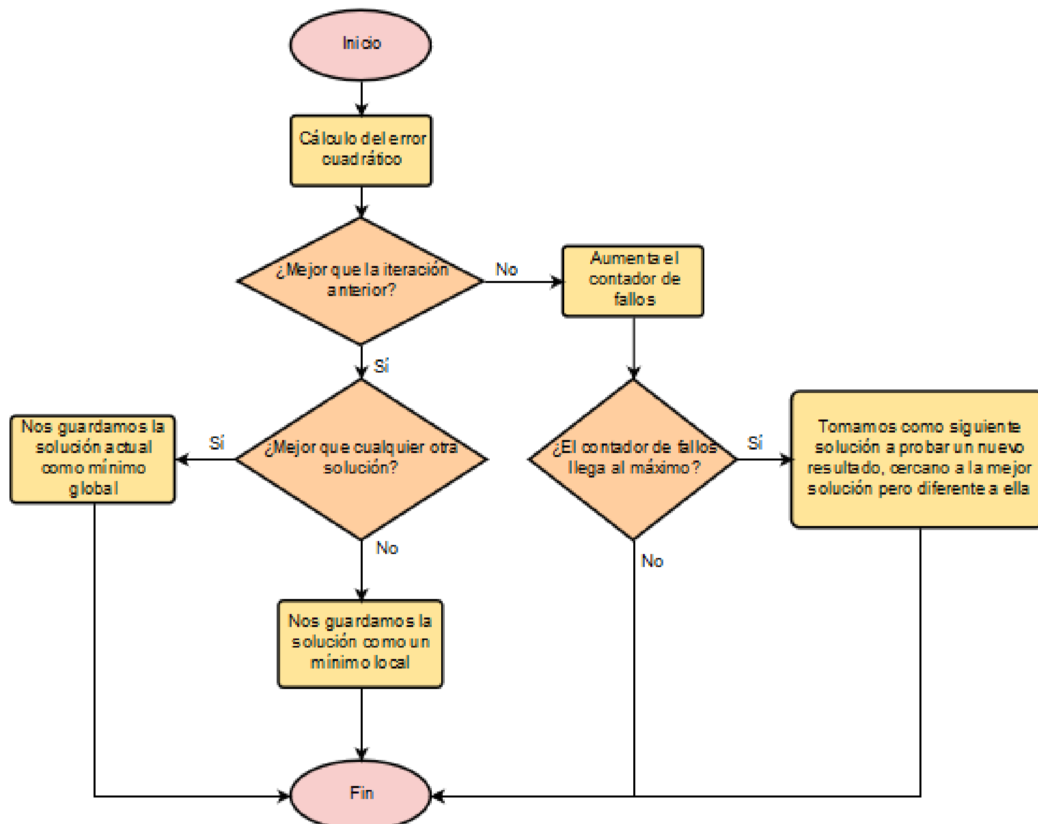

Figura 12. Diagrama de la estrategia con los mínimos locales

Esta estrategia lo que causa es que, cuando se está en un valle y una solución mayor o menor lo que hace es empeorar el resultado, se prueba a utilizar un nuevo resultado para intentar lograr salir de los valles. Por defecto, el resultado nuevo al que se salta es un valor igual al mejor resultado hasta el momento multiplicador por un valor al azar. Este valor generado al azar puede ser modificado con un parámetro, permitiendo regular el margen de azar. Concretamente, aumentar el rango de valores posibles al azar provoca:

- Ventajas:
  - Es posible salir de valles más anchos.
  - Es menos sensible a malas estimaciones iniciales.
- Desventajas:
  - Consume más iteraciones y tiempo.

El valor más adecuado para este parámetro de azar se determinó experimentalmente. Se realizaron pruebas con valores de este parámetro entre 0,5 y 2, comparando los resultados, evaluando tanto el número de iteraciones como el error cometido. Se encontró que el parámetro funciona mejor cuando vale entre 0,8 y 1,2, utilizándose ese margen a partir de entonces.

Con las características anteriores, el código es capaz de proporcionar soluciones en las que su error cuadrático varía en el sexto decimal respecto al proporcionado por *Lsqcurvefit*. Sin embargo, el error absoluto (la resta de la función evaluada en la solución menos los datos de salida) varía en gran medida, de media un 20% respecto a *Lsqcurvefit.* Además de lo anterior, el código necesita muchas más iteraciones para mejorar. En las pruebas realizadas, para mejorar un orden de magnitud era necesario multiplicar el número de iteraciones por 10, aproximadamente.

Mirando la gráfica de velocidad de cambio de lambda en la figura 10 se puede entender qué está ocurriendo: El código valora con un 1 si lambda cambia muy rápido, 0 si cambia a velocidad media y -1 si varía lentamente. Se puede ver que, con lambda variando tan rápido, las soluciones (los valores de x) varían mucho, lo que dificulta en gran medida la tarea de encontrar una solución (convirtiéndolo en cuestión de suerte).

Obviamente resultaba un consumo de tiempo excesivo y no era una solución admisible.

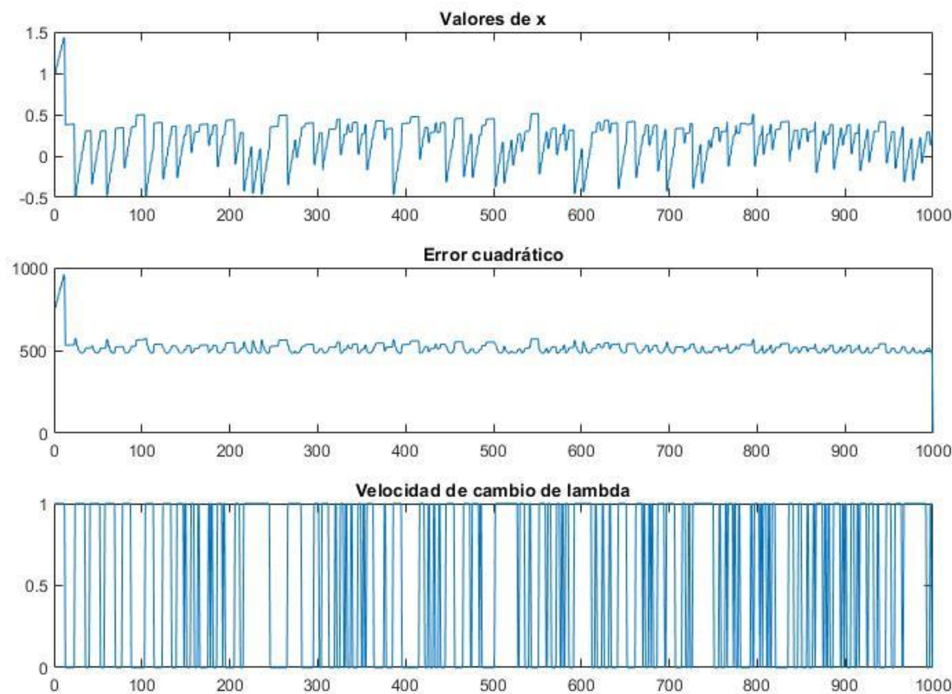

Figura 13. Resultados de una de las primeras soluciones

Como se ha explicado, lo que limita la precisión a la hora de lograr los resultados no está siendo el error cuadrático (que es lo que se optimiza en el algoritmo de ~~Levenberg-Marquadt~~Levenberg-Marquardt, expuesto en el punto 1.4 de este trabajo) sino que el error que es necesario reducir es el obtenido al evaluar la función con el resultado obtenido, comparándolo con el valor de los datos de salida. Al ser esta la situación, se varía el algoritmo para buscar reducir el error:

- La condición de permanencia en el bucle según el error relativo se elimina, pasando a vigilarse el error absoluto, como la resta entre la función evaluada en la solución menos los datos de salida
- A la hora de variar el valor de lambda, se evalúa el error absoluto en lugar del cuadrático a la hora de decidir cuál será el siguiente valor. Además, se hace que el valor de lambda no dependa solo del error absoluto, sino de la tolerancia elegida, escalándose lambda según las necesidades.

Realizar los cambios anteriores hace que el algoritmo utilizado no siga al pie de la letra el algoritmo de ~~Levenberg-Marquadt~~Levenberg-Marquardt (ver el punto 1.4 de este trabajo) pero de esta forma se consigue reducir muy notablemente el error cometido.

3.6. Primeras versiones de milevmar

A continuación, se comentarán algunas de las primeras versiones del código *milevmar*, creado en este trabajo para replicar el comportamiento de *Lsqcurvefit* y trasladarlo posteriormente a un archivo MEX.

Los resultados se expresarán con algunos gráficos y tablas. Respecto a estas últimas, cabe comentar cómo se expresan los datos:

- *Tiempo Lsqcurvefit*: Es el tiempo que le cuesta (de media) a MatLab resolver el problema. Todas las pruebas de este apartado están realizadas sobre el mismo problema, de modo que los tiempos son comparables.
- *Tiempo milevmar3*: Es el tiempo total de ejecución del código realizado, que interesa minimizar.
- *Error absoluto comparado*: Se calcula el error absoluto cometido por *milevmar3* (evaluando la función en el resultado obtenido y restándolo a la salida considerada como correcta) y se compara con el error absoluto cometido por Lsqcurvefit. Esta diferencia se expresa como un porcentaje. Esta medida tiene la particularidad de que no mide si *milevmar3* proporciona resultados similares a *Lsqcurvefit*, sino que mide cómo de preciso es. En ocasiones, el error absoluto cometido por Lsqcurvefit y por milevmar3 puede ser muy similar, pero sin embargo cada uno entregar como solución un mínimo local diferente.
- *Error cuadrático comparado*: Se calcula el error cuadrático de *milevmar3* y el de *Lsqcurvefit*, comparándose ambos y expresando esa diferencia como un porcentaje.
- *Solución*: La diferencia entre el valor final propuesto por *Lsqcurvefit* y por *milevmar3*, en porcentaje.

3.6.1. Versiones iniciales

El proyecto se inició realizando el algoritmo para vectores (*milevmar*) y más tarde se desarrolló para matrices (*milevmar2*). Los objetivos deseados al realizar estas versiones eran ir familiarizándose con el entorno y los algoritmos mientras se creaba de forma gradual y estructurada un código de tamaño creciente.

3.6.2. Versiones de milevmar3 (de la 1 a la 3)

A partir de este punto se trabajó ya siempre con imágenes (*milevmar3*). Inicialmente no se obtuvieron muy buenos resultados, como se puede ver en la parte inferior.

Estas versiones utilizaban la siguiente estrategia para ir ajustando el resultado que, involuntariamente, no seguía el algoritmo de Levenberg-Marquardt. La estrategia poseía los siguientes pasos:

1. Se evalúa la función con la mejor solución encontrada hasta el momento, obteniendo una cierta imagen.
2. Al resultado anterior se le restaba, valor a valor, la imagen de salida *ydata*.
3. El último resultado obtenido resulta ser una matriz que contiene valores positivos donde la función posee un valor superior al de *ydata* y negativos donde es menor. Esto se aprovecha para comparar cada valor de la función con cero, obteniéndose así una matriz de ceros y unos.
4. El número total de ceros y unos era comparado con el tamaño de la matriz para decidir si era necesario aumentar los valores de la matriz en la siguiente iteración o disminuirlos, buscando lograr que hubiese el mismo número de ceros y unos.

En realidad, esta estrategia llevaba a bucles, hallándose la mejor solución pronto (en las primeras iteraciones) y luego repitiéndose en un ciclo de longitud variable (según la aproximación inicial y el problema).

En la tabla 2 se exponen los resultados: Se puede apreciar como, aunque el tiempo de ejecución de *milevmar3* es bueno (menos de un cuarto del tiempo utilizado por *Lsqcurvefit*) y el error cuadrático de ambas funciones es similar, la solución obtenida es muy diferente. De esta forma se lograba el objetivo principal del trabajo (acelerar el algoritmo) pero se incumplía el secundario (mantener el error bajo control y obtener los mismos resultados que *Lsqcurvefit*).

Tiempo Lsqcurvefit	Tiempo milevmar3	Error absoluto comparado	Error cuadrático comparado	Solución
1.8380 s	0.4025 s	84.1296 %	0.0668 %	3735 %

Tabla 2. Diferencia entre Lsqcurvefit y milevmar3 (versión 1)

En la figura 14 se ve la progresión de valores de la solución x y el error cuadrático. Se puede ver que hay un bucle, repitiéndose una sucesión de valores, lo que causa que milevmar3 no esté calculando correctamente la solución.

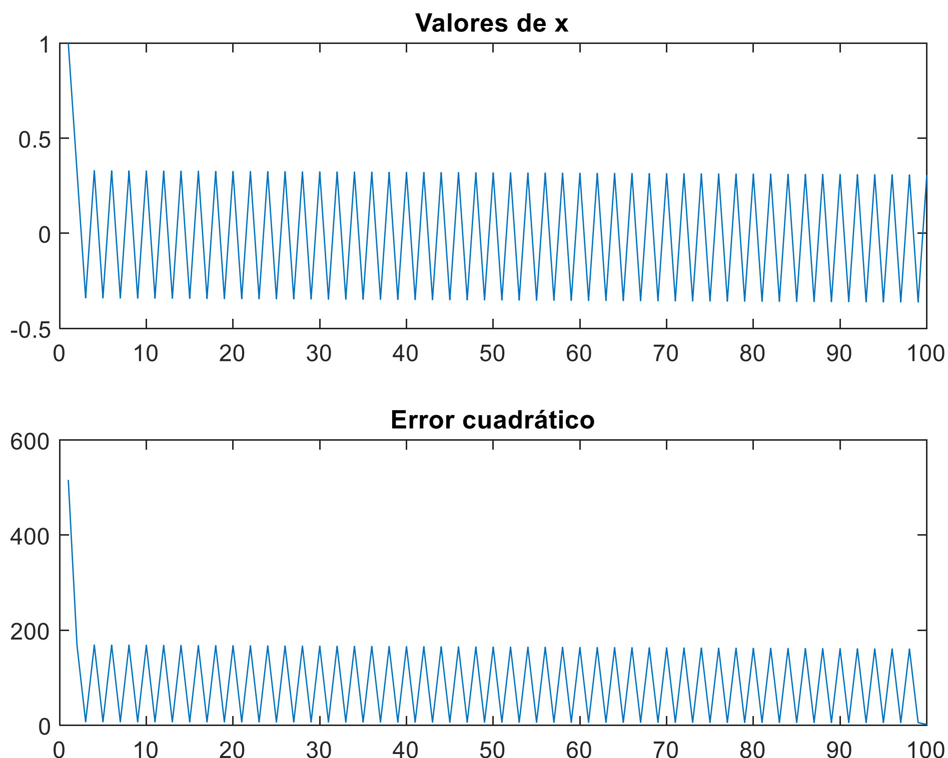

Figura 14. Resultados milevmar3 (versión 1)

3.6.3. Versiones de milevmar3 (de la 4 a la 5)

En estas versiones se abandonó la idea de las versiones 1 a 3 de *milevmar* y se aplicó un algoritmo muy fiel al de Levenberg-Marquardt (expuesto en el punto 1.4.).

En la tabla 3 se puede ver como se siguen cumpliendo los requisitos de tiempo y de error cuadrático, pero el error cometido por *milevmar3* a la hora de proporcionar una solución igual a la de *Lsqcurvefit* sigue siendo alto.

Tiempo Lsqcurvefit	Tiempo milevmar3	Error absoluto comparado	Error cuadrático comparado	Solución
1.8380 s	0.5913 s	69.1153 %	0.0456 %	3020 %

Tabla 3. Diferencia entre Lsqcurvefit y milevmar3 (versión 5)

La figura 15 ilustra cuál es el problema que está ocurriendo: Cuando se aproxima a la solución, en lugar de detenerse y buscar en ese entorno, *milevmar3* sigue adelante, limitando la precisión en los resultados.

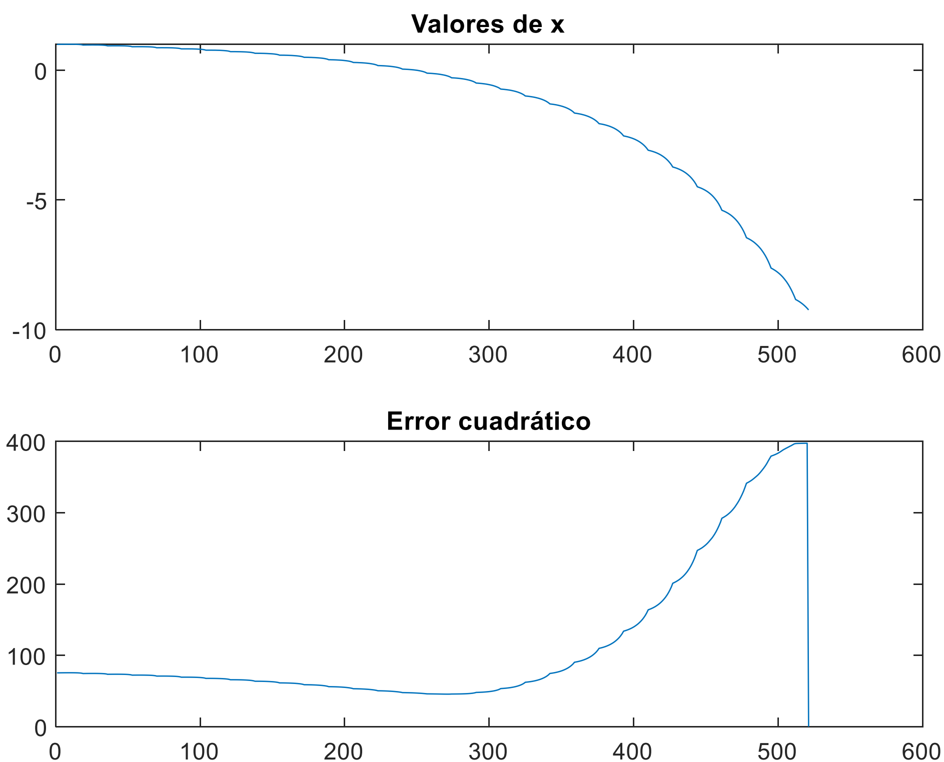

Figura 15. Resultados milevmar3 (versión 5)

3.6.4. Versiones de milevmar3 (de la 6 a la 10)

Este grupo de versiones similares a la solución final (que resultó ser la versión 11). Lo que cambia de una a otra es en esencia la forma en la que varía λ, siendo el λ parámetro de amortiguación del algoritmo de Levenberg-Marquardt, según la ecuación 1.4.3.5 del punto 1.4.

$$[(J^T J + \lambda\, \mathrm{diag}(J^T J\,)]\delta = \;\; J^T\, [y - f(\,\beta)]\, [y - f(\,\beta)] \qquad \textbf{(1.4.3.5)}$$

El valor de λ es clave, ya que define tanto la velocidad de convergencia como la capacidad del algoritmo para conseguir converger a un resultado satisfactorio.

Debido a que no es posible demostrar teóricamente qué valor de λ es adecuado a todas las situaciones, el algoritmo debe ser capaz de manipular el valor de λ para intentar asegurar la convergencia y mantener bajo control tiempo necesario para ella.

Anteriormente, en las versiones 4 y 5 de *milevmar3*, λ únicamente poseía una velocidad, aumentando multiplicándose por 1,1 en cada iteración.

- Versión 6:
  - Se decide la velocidad de cambio de λ comparando un valor derivado de la matriz de ecuaciones no lineales con un valor arbitrario relacionado con la tolerancia.
  - λ tiene dos velocidades: Multiplicado por 10 o por 1.1.
- Versión 7:
  - Se decide la velocidad de cambio de λ comparando cuánto ha cambiado el valor de la solución en la última iteración con un valor arbitrario relacionado con la tolerancia.
  - λ tiene tres velocidades: Multiplicado por 10, por 1.1 y por 1.01.
- Versión 8 a 10:
  - Se decide la velocidad de cambio de λ comparando el error absoluto con un valor arbitrario relacionado con la tolerancia.
  - λ tiene tres velocidades: Multiplicado por 10, por 1.1 y por 1.01.

La tabla 4 ilustra los resultados obtenidos con la versión 6 de *milevmar3*. El error cometido es mucho menor que en las versiones anteriores, aunque a costa de aumentar el tiempo de ejecución, hasta tardar casi lo mismo que *Lsqcurvefit*. En las versiones posteriores los principales cambios realizados son sobre la elección de λ, mejorándose la velocidad y la precisión.

Tiempo Lsqcurvefit	Tiempo milevmar3	Error absoluto comparado	Error cuadrático comparado	Solución
1.8380 s	1.6438 s	0,4858 %	$2.3034 * 10^{-6}$ %	21,27 %

Tabla 4. Diferencia entre Lsqcurvefit y milevmar3 (versión 6)

La figura 16 muestra la evolución de valores de la solución x, el error cuadrático y la velocidad a la que varía λ, para *milevmar3* versión 6. Se puede ver como el algoritmo propone soluciones oscilando alrededor de la solución correcta.

A diferencia de las versiones anteriores, aunque pueda parecer que hay un bucle y que no se consigue mejorar la solución en el trascurso de las iteraciones, en realidad sí se está mejorando: En las primeras 50 iteraciones logra resultados como los de las versiones anteriores, dedicando el resto de iteraciones a refinar el resultado. Esta última afirmación también podemos realizarla

observando la velocidad de cambio de λ: La velocidad de cambio del factor de amortiguación λ a menudo toma el valor 0, utilizado en milevmar3 versión 6 para identificar cuando el algoritmo detecta que está próximo a la solución y varía más lentamente las soluciones propuestas.

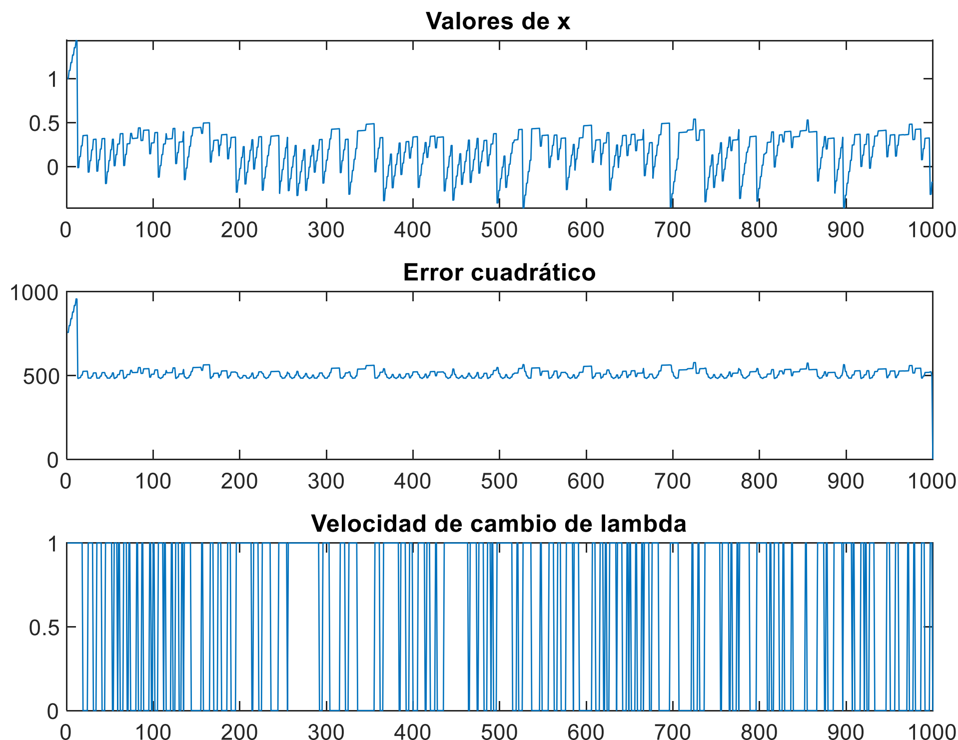

Figura 16. Resultados milevmar3 (versión 6)

3.7. Aspectos a considerar del código de la versión final de milevmar3

En el anexo se incluye el código íntegro. A continuación, se comentarán algunas partes.

3.7.1 Contador de fallos

Esta parte del código, en cada iteración, se comprueba si se está avanzando hacia una mejor solución o si se está estancado. Se corresponde a la figura 12 (*Diagrama de estrategia con los mínimos locales*).

Se utilizan dos parámetros que se definen al iniciar la función milevmar3:

- *margendefallosS*: Es un parámetro que controla cuánta paciencia tiene el algoritmo antes de decidir que está yendo en la dirección incorrecta, dejando de buscar un mínimo en esa dirección para volver a la mejor aproximación que tenía anterior (pero no exactamente en ese resultado, sino en un punto al azar cercano).
- *Multiplicadormargendefallos*: Este parámetro controla cuánto se desvía al no encontrar un mejor mínimo. Con 0 no tiene efecto. Si es positivo aumenta la desviación y si es negativo la disminuye. Valores típicos de 0 a 0.2.

```
if S(iter)<mejorS
    mejorX=x;
    mejorS=S(iter);
    contadordefallosS=0;
else
    contadordefallosS=contadordefallosS+1;
    if contadordefallosS>=(margendefallosS*(1+multiplicadormargendefallos))
        contadordefallosS=0;
        x=mejorXdetodos*((0.4+multiplicadormargendefallos)*rand(1)+0.8-
multiplicadormargendefallos);
    end
end

if S(iter)<mejorSdetodos
    mejorXdetodos=x;
    mejorSdetodos=S(iter);
end
```

El sentido al diferenciar entre “mejorS” y “mejorSdetodos” es el mismo que con “mejorX” y “mejorXdetodos”, estando relacionado con la búsqueda del mínimo error cuadrático, intentando hallar el mínimo global sin conformarse con un mínimo local. De esta forma, controlamos con “mejorS” y “mejorX” si estamos mejorando iteración a iteración. Si dejáramos de mejorar, sabríamos que hemos encontrado un mínimo.

Así, tenemos en “mejorSdetodos” y “mejorXdetodos” los valores del mínimo global hasta el momento. Cada iteración se comprueba si un mínimo local encontrado puede convertirse en el nuevo mínimo global.

3.7.2 Variación de λ

A continuación, se muestra la parte del código que controla el valor de λ en cada iteración. El valor de *multiplicadordevelocidaddelambda* típicamente es entre 1 y 100.

En esta parte del código, lo primero que se comprueba es cómo es el error absoluto que comete *milevmar3* respecto a la salida *ydata*, comparándolo con la tolerancia para decidir cómo de próximo se está a la solución. Si el error es mayor que 100 veces la tolerancia, entonces se elige que λ cambie rápidamente. Si el error está entre 100 y 50 veces la tolerancia, se toma la velocidad intermedia, y si es menor que 50 veces la tolerancia, se toma la menor velocidad.

En este proceso también se incluye un sistema para restaurar el valor inicial de λ en caso de que aumente demasiado. El margen en el que es restaurado depende de la velocidad a la que se esté: Si es media o baja, entonces si λ supera el valor de λ inicial 5 veces, se recupera el valor de λ inicial. Sin embargo, en el caso de que λ esté variando rápidamente, el valor máximo que se

le permite tomar es muchísimo mayor (por ejemplo, 7 órdenes de magnitud por encima del λ inicial, dependiendo de la tolerancia y del valor de *multiplicadordevelocidaddelambda*).

Como se trató en el punto 1.4.4. , ya que no existe un λ inicial ideal (depende de los datos del problema y de la aproximación inicial) se podría haber tomado un valor de λ inicial diferente, aunque esto a su vez habría cambiado la forma en la que evoluciona λ. Se eligió un λ dependiente de la tolerancia para poderlo usar conjuntamente en la comparación con el error absoluto y la tolerancia, en lugar de haber escogido otro valor arbitrario. De esta manera, independientemente del problema, λ se reinicia en el momento adecuado.

```
if errorabsoluto<=(tol*100) % Valor arbitrario, y decidimos la velocidad de lambda
   if errorabsoluto<=(tol*50) % Valor arbitrario
      if lambda>(lambdainicial*5)
        % Condición de reinicio de lambda
         lambda=lambdainicial;
      end
      lambda=lambda*1.01;     % Velocidad baja
      velocidad(iter)=-1;
   else
      if lambda>(lambdainicial*10)   % Velocidad media
         lambda=lambdainicial;
      end
      lambda=lambda*1.1;
      velocidad(iter)=0;
   end
else   % Máxima velocidad
   lambda=lambda*10*multiplicadordevelocidaddelambda;
   velocidad(iter)=1;
end
if lambda>(lambdainicial/tol*multiplicadordevelocidaddelambda)
      % Condición de reinicio de lambda
   lambda=lambdainicial;
end
```

3.7.3 Resolución del sistema de ecuaciones no lineales

La función milevmar3 se encuentra aproximadamente la mitad del tiempo realizando estas operaciones en cada iteración. Es la parte más demandante computacionalmente. Para mejorar el código se podría intentar buscar una forma de que estas operaciones se realicen más rápido.

```
delta(:,:,1)=(JJ(:,:,1)+lambda.*D(:,:,1))\(JJ(:,:,1));
delta(:,:,2)=(JJ(:,:,2)+lambda.*D(:,:,2))\(JJ(:,:,2));
delta(:,:,3)=(JJ(:,:,3)+lambda.*D(:,:,3))\(JJ(:,:,3));
```

3.8. Problemática de las matrices singulares

Al resolver los sistemas de ecuaciones en *milevmar3*, a partir de cierto tamaño de matriz y cierto número de iteraciones, aparece el problema de las matrices singulares, avisando MatLab del suceso.

Si se examinan más en profundidad esas matrices, encontramos que están repletas de valores muy pequeños o directamente nulos. Si se cae en el caso de resolver iterativamente sistemas de este tipo, siendo en cada iteración las cantidades más pequeñas, finalmente se llega a una matriz prácticamente nula.

Durante el transcurso del proyecto se descubrió que, en la función propia *milevmar3*, si lambda posee un margen superior más alto y una capacidad de aumentar más rápido, este efecto se ve minimizado, aunque la convergencia es mucho más lenta al oscilar demasiado por los valores elevados de lambda. Se ha encontrado que, en los casos estudiados, un factor 100 resulta un equilibrio adecuado entre coste en tiempo y precisión en el resultado.

Este problema no es exclusivo de milevmar3: La función de MatLab *Lsqcurvefit* también posee esta limitación, aunque requiere de imágenes de mayor tamaño que *milevmar3* (a partir de imágenes de 450 píxeles se pueden comenzar a observar estos efectos, informando MatLab de que se están resolviendo matrices casi singulares).

Capítulo 4. Resultados.

A continuación, se exponen los resultados obtenidos, utilizando imágenes reales recortadas y la versión final del código *milevmar3*.

Las tablas presentadas en este capítulo siguen la misma nomenclatura que las presentadas en el punto 3.6.

4.1. Introducción a las imágenes utilizadas

Para realizar el trabajo se han utilizado una serie de imágenes multieco. Realmente, para el propósito de este trabajo, no resulta importante qué parte de la imagen se utilice o cómo se recorte, siempre que la imagen de entrada (*xdata* en el algoritmo) haya sido tratada como la imagen de salida (*ydata* en el algoritmo). Por ejemplo, si en la entrada se recortó la imagen tomando un cuadrado desde el píxel (135,200) al (335-400), de la imagen de salida deben recortarse esos mismos píxeles.

El hecho de que no sea importante qué parte de la imagen se utiliza se debe a los objetivos perseguidos: Optimizar el tiempo del ajuste de curva, obteniendo los mismos resultados. No se están evaluando los algoritmos de preprocesado y extracción de biomarcadores (el T2 en este caso) sino que, independientemente del desempeño de estos, se busca mejorar su velocidad sin alterar el resultado.

Las imágenes que se muestran proceden de archivos DICOM, siendo recortes algunas de ellas (figuras 17,18,20,21,23,24) y otras siendo la imagen completa (figuras 27 y 28). Los datos de cada imagen son tres matrices cuadradas, de modo que, por ejemplo, una imagen de 50x50 píxeles está compuesta por 3 matrices de 2500 (50x50) valores cada una.

4.2. Prueba 1: imágenes de 36x36 pixeles.

En esta prueba se utilizaron imágenes de 36x36 píxeles, siendo la prueba más pequeña realizada con imágenes en este proyecto. Se puede observar en la tabla 5 que el tiempo que utiliza *milevmar3* para resolver el problema es muy inferior al de *Lsqcurvefit*, obteniendo prácticamente la misma solución.

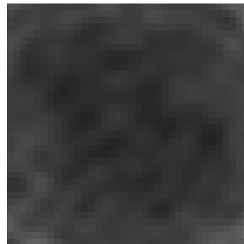

Figura 17. Imagen de entrada de la prueba 1

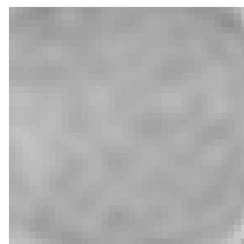

Figura 18. Imagen de salida de la prueba 1

Tiempo Lsqcurvefit	Tiempo milevmar3	Error absoluto comparado	Error cuadrático comparado	Solución
1.8380 s	0.2123 s	0.0012 %	$1.3226 * 10^{-11}$ %	$9.0562 * 10^{-6}$ %

Tabla 5. Diferencia entre Lsqcurvefit y milevmar3

En la figura 19 se aprecia la evolución de la solución x, así como el error cuadrático y la velocidad de cambio de λ y también los valores de λ. Se puede decir que en este caso milevmar3 ha cumplido todos los objetivos propuestos.

Respecto a la solución, se observa que llega su límite de mil iteraciones, aunque sin embargo ya ha conseguido una solución válida (igual a la de *Lsqcurvefit*) sigue iterando debido a que el error absoluto cometido considera que es demasiado elevado. Realmente, en el caso de *Lsqcurvefit* también resulta elevado el error absoluto resultado de restar a la función evaluada en la solución los valores de *ydata*. La diferencia entre *milevmar3* y *Lsqcurvefit* en este caso resulta ser que milevmar3 insiste un poco más para intentar lograr una mejor solución.

Se consideró la posibilidad de reducir la exigencia de *milevmar3* respecto al error absoluto, pero se prefirió no alterar el código debido a que, aunque podía mejorarse los resultados en este caso concreto, al hacerlo empeoraba resultados en otras pruebas. Esto se debe a que la convergencia depende en gran medida de cada problema concreto y de la aproximación inicial.

λ evoluciona correctamente: Al inicio va creciendo rápidamente, volviendo a tomar su valor inicial cuando crece demasiado, para posteriormente, una vez cerca de la solución, cambiar despacio para conseguir precisión en la solución.

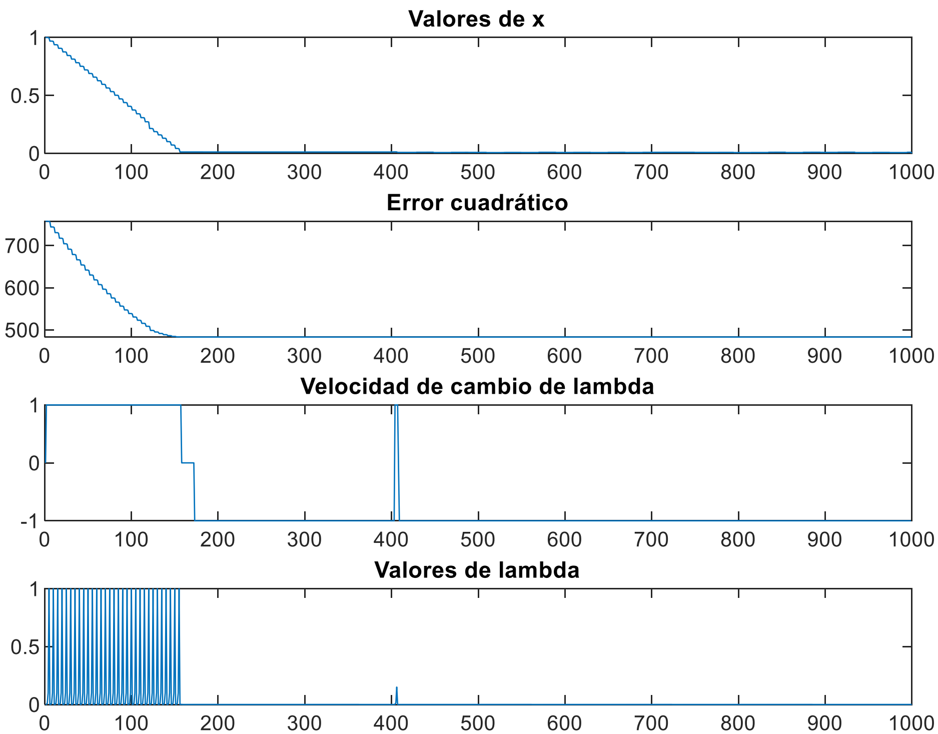

Figura 19. Resultados obtenidos en la prueba 1

4.3. Prueba 2: imágenes de 71x71 píxeles.

En esta prueba se utilizaron imágenes de 71x71 píxeles. Los resultados son buenos en esta prueba también, aunque menos que en la prueba anterior: El tiempo es algo más de la mitad respecto a *Lsqcurvefit*, existiendo error cometido, aunque mínimo.

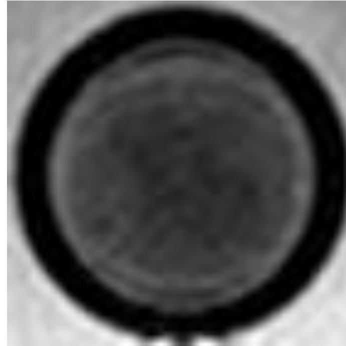

Figura 20. Imagen de entrada de la prueba 2

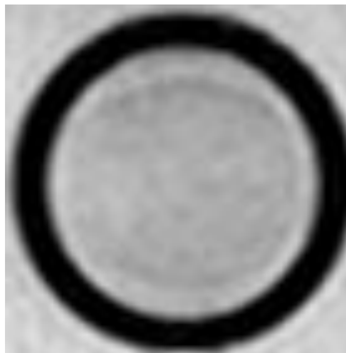

Figura 21. Imagen de salida de la prueba 2

Como se puede ver en la tabla 6, *milevmar3* en este caso también cumple los objetivos propuestos en el trabajo. El error cometido y la diferencia entre ambas soluciones se mantienen en valores bajos. El tiempo de ejecución de *milevmar3*, aunque es la mitad que el de *Lsqcurvefit*, ha aumentado en una mayor proporción respecto el que aumentó *Lsqcurvefit*. En el punto 4.6 se analiza esta cuestión acerca del rendimiento.

Tiempo Lsqcurvefit	Tiempo milevmar 3	Error absoluto comparado	Error cuadrático comparado	Solución
3.3015 s	1.5938 s	$4.0902 * 10^{-5}$ %	$3.5880 * 10^{-10}$ %	$5.1533 * 10^{-4}$ %

Tabla 6. Diferencia entre Lsqcurvefit y milevmar3

En la figura 22 vemos la evolución de los resultados de esta prueba. Viendo la gráfica de la velocidad de cambio de λ, vemos que en este caso se logró muy rápidamente acercarse a la solución, ya que pronto λ inició su modo de cambio lento (indicado con el valor -1 en la gráfica

de la figura 22). Este comportamiento ha sido bastante habitual durante el trabajo: Se alcanza muy rápidamente un valor cercano a la solución, siendo muy costoso a partir de ahí lograr mejorar hacia una solución más precisa.

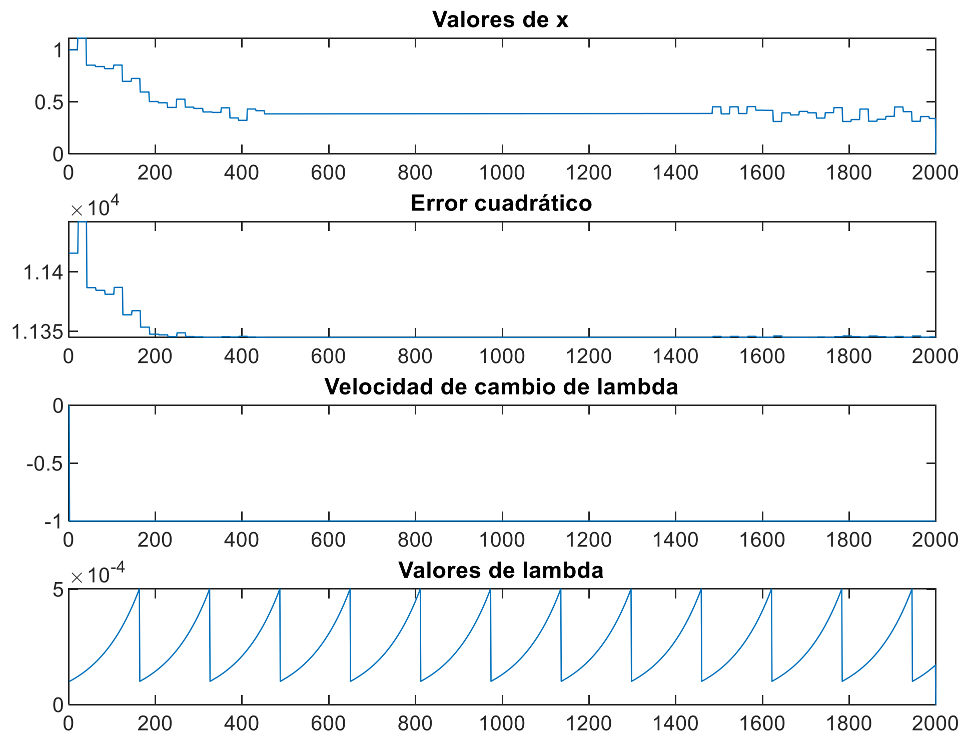

Figura 22. Resultados obtenidos en la prueba 2

4.4. Prueba 3: imágenes de 160x160 píxeles.

En esta prueba se utilizaron imágenes de 160x160 píxeles. Con imágenes de este tamaño o superior, aparecen problemas en la ejecución de *milevmar3* si no se realiza ningún ajuste adicional: MatLab advierte de que está trabajando con matrices singulares. Concretamente, son las matrices que se resuelven en cada iteración y que contienen las ecuaciones no lineales. En el punto 4.7 de este proyecto se trata más extensamente esta cuestión.

La forma de resolver esta situación es utilizando un parámetro de *milevmar3*, que permite alterar la velocidad a la que cambia lambda y su margen superior. Gracias a darle un valor más alto a ese parámetro, *milevmar3* devuelve una más correcta solución, aunque a coste de aumentar sensiblemente el coste en tiempo.

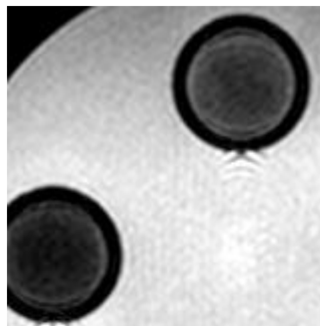

Figura 23. Imagen de entrada de la prueba 3

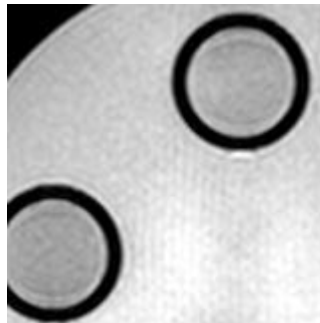

Figura 24. Imagen de salida de la prueba 3

Los resultados expuestos a continuación son los obtenidos con diferente número de iteraciones de *milevmar3*.

En esta situación se puede analizar el problema: milevmar3 puede lograr cumplir el objetivo de reducir el tiempo (tabla 7 y figura 25) pero no en gran medida y a costa de sacrificar en parte el objetivo secundario del error.

Tiempo Lsqcurvefit	Tiempo milevmar3	Error absoluto comparado	Error cuadrático comparado	Solución
5.9976 s	4.574 s	0.0464 %	0.0167 %	0.01 %

Tabla 7. Diferencia entre Lsqcurvefit y milevmar3 (con 1500 iteraciones)

Si se dobla el número de iteraciones (tabla 8 y figura 26) el tiempo aumenta proporcionalmente doblándose, pero sin embargo sí se logra una mejora considerable en el error.

Tiempo Lsqcurvefit	Tiempo milevmar3	Error absoluto comparado	Error cuadrático comparado	Solución
5.9976 s	10.462 s	$9.8468 * 10^{-5}$ %	$2.9803 * 10^{-5}$ %	$1.5142 * 10^{-5}$ %

Tabla 8. Diferencia entre Lsqcurvefit y milevmar3 (con 3000 iteraciones)

Sin embargo, si el número de iteraciones se multiplica por 100 (tabla 9) no se consigue una mejora significativa en el error, teniendo asegurado un gran coste en tiempo.

Tiempo Lsqcurvefit	Tiempo milevmar3	Error absoluto comparado	Error cuadrático comparado	Solución
5.9976 s	17 minutos y medio	$6.3708 * 10^{-5}$ %	$1.5257 * 10^{-5}$ %	$7.5104 * 10^{-6}$ %

Tabla 9. Diferencia entre Lsqcurvefit y milevmar3 (con 300.000 iteraciones)

En las figuras 25 y 26 vemos los resultados con 1500 iteraciones y con 3000, respectivamente. En esta primera (figura 25) se puede ver parte del problema en los valores de la solución x: Aunque está el algoritmo ofreciendo soluciones en el margen correcto (entre -30 y -36 en este caso) este margen es demasiado amplio como para lograr afinar con precisión la solución. La velocidad de cambio de λ también avisa de esta situación, ya que se mantiene oscilando entre cambiar rápido y lento, señal de que no se consigue aproximarse de forma consistente a la solución, sino que existen unas oscilaciones excesivas.

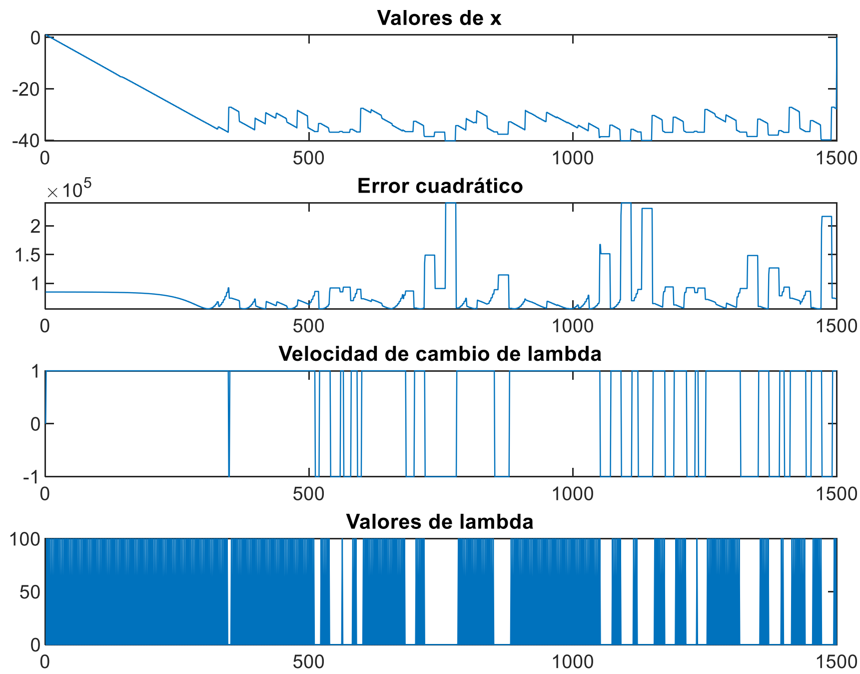

Figura 25. Resultados obtenidos en la prueba 3, con 1500 iteraciones

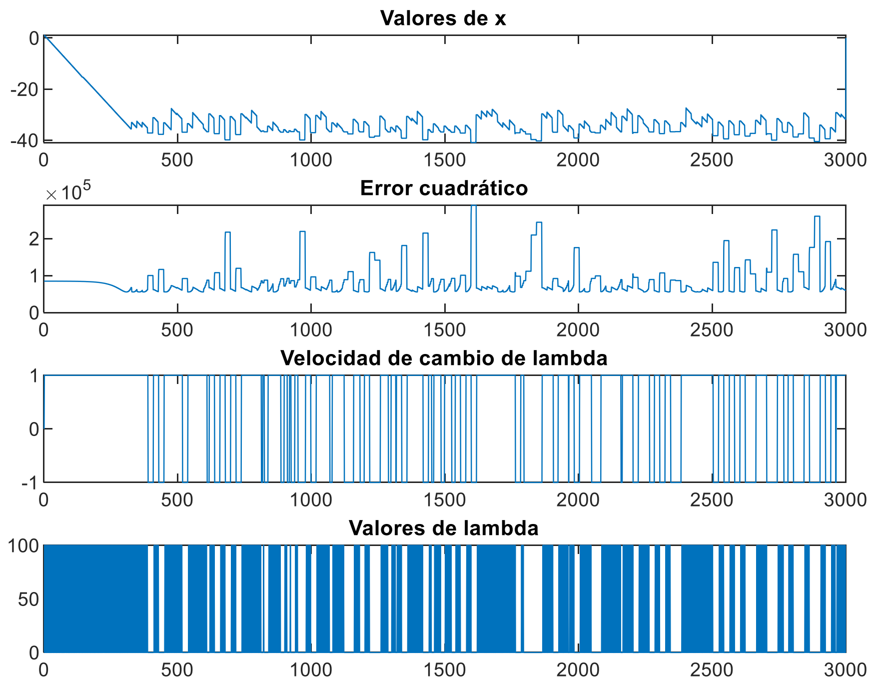

Figura 26. Resultados obtenidos en la prueba 3, con 3000 iteraciones

4.5. Prueba 4: imágenes de 293x293 píxeles.

En esta prueba se utilizaron imágenes de 293x293 píxeles (la totalidad de la imagen necesaria para este proyecto). En este caso, ocurre algo muy similar al anterior, pudiéndose resolver los problemas de la misma manera y con las mismas consecuencias.

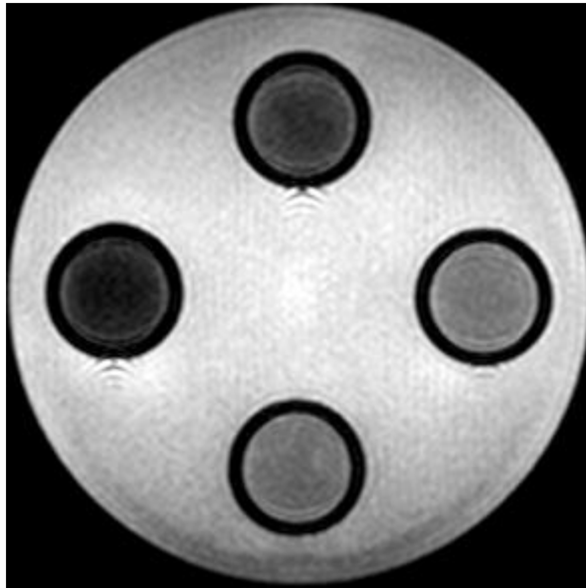

Figura 27. Imagen de entrada de la prueba 4

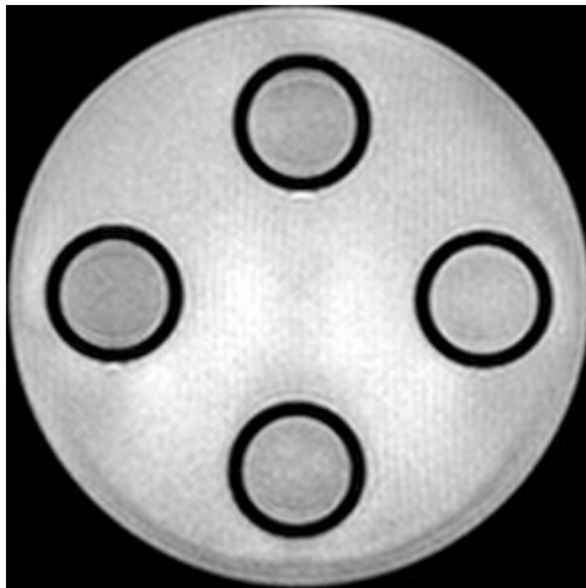

Figura 28. Imagen de salida de la prueba 4

En la tabla 10 se exponen los resultados de la prueba 4. En ella, vemos que de nuevo se cumple el objetivo secundario de mantener el error controlado, pero sin embargo el coste en tiempo aumenta en gran medida en *milevmar3* respecto a *Lsqcurvefit*, incumpliendo el objetivo principal.

Tiempo Lsqcurvefit	Tiempo milevmar3	Error absoluto comparado	Error cuadrático comparado	Solución
13.382 s	35.53 s	$1.4375 * 10^{-3}$ %	$5.3065 * 10^{-4}$ %	0.0100 %

Tabla 10. Diferencia entre Lsqcurvefit y milevmar3

Observando la figura 29, se entiende que el comportamiento de *milevmar3* en esta prueba es similar al que tenía en la prueba anterior, encontrándose el mismo problema: La convergencia no se da porque oscila demasiado.

Se pensó que una solución podría ser alterar λ para reducir la oscilación, pero esto ni empeora ni mejora la situación. Esto se debe a que, aunque al oscilar menos se gana precisión, por otro lado, tarda más en moverse por los “tramos rápidos”, lo que causa iteraciones adicionales y ralentización.

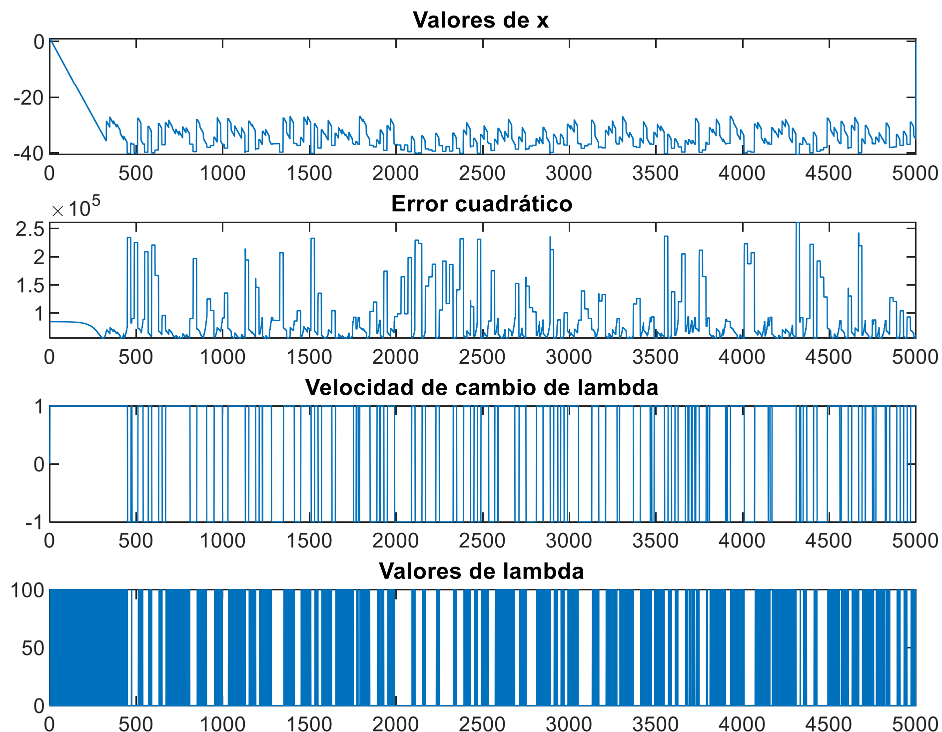

Figura 29. Resultados obtenidos en la prueba 4

4.6. Rendimiento de la solución

Como se ha podido ver, la solución propuesta en este proyecto resulta funcionar adecuadamente con tamaños de imagen reducidos, presentando problemas cuanto mayor es la imagen.

Si representamos (figura 30) el coste en tiempo frente al tamaño de la imagen, podemos ver que con imágenes de tamaño reducido *milevmar3* es más rápido que *Lsqcurvefit*, mientras que con imágenes de gran tamaño ocurre lo contrario.

Tanto *milevmar3* como *Lsqcurvefit* presentan un comportamiento similar al incrementarse el tamaño de las imágenes, pero con las imágenes grandes el coste en tiempo aumenta de forma más rápida en *milevmar3*.

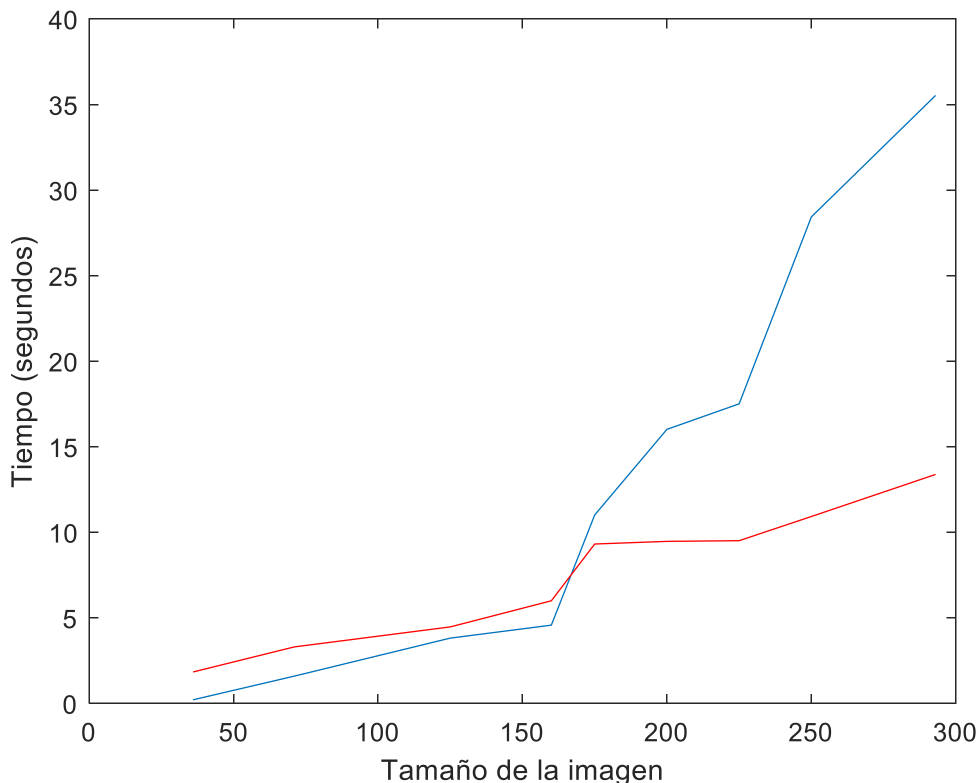

Figura 30. Tiempo de ejecución según el tamaño de la imagen. En rojo *Lsqcurvefit* y en azul *milevmar3*

Del mismo modo, abajo (figura 31) se representa el rendimiento de cada función, según el tamaño de la imagen. El rendimiento ha sido calculado como el cociente entre las dimensiones de la imagen y el tiempo necesario para realizar el ajuste de la curva.

Se puede observar que, inicialmente, *milevmar3* resulta mucho más eficiente que *Lsqcurvefit*, pero esta situación se invierte posteriormente. *Lsqcurvefit* se mantiene aproximadamente estable en su rendimiento, casi independientemente del tamaño de la imagen, mientras que *milevmar3* (observando la figura 31) oscila entre un rendimiento 9 veces mejor (180 frente a 20) y la mitad de rendimiento (10 frente a 20).

Según la documentación que ofrece MatLab acerca de los archivos MEX, se puede esperar lograr con ellos una ejecución nueve veces más rápida como máximo. En este caso, *milevmar3* sí alcanza ese rendimiento máximo teórico, bajo la condición de que la imagen sea de pequeño tamaño.

El motivo del rendimiento decreciente al aumentar el tamaño de la imagen no se ha averiguado, pero sería una línea futura a seguir.

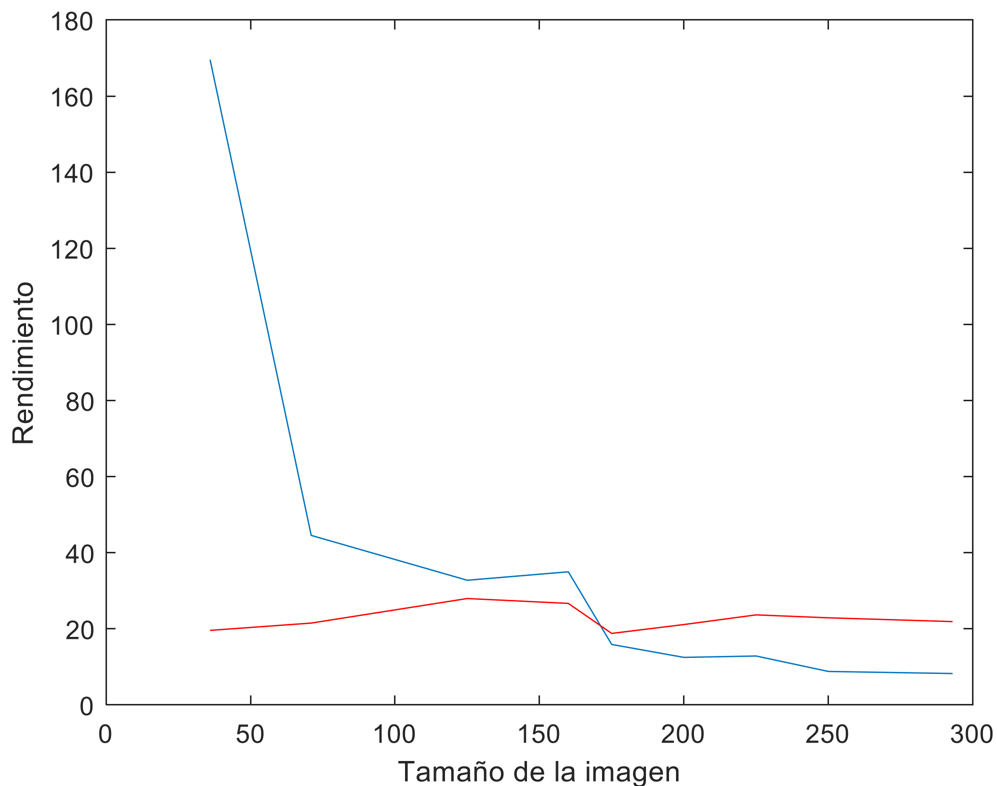

Figura 31. Rendimiento del algoritmo según el tamaño de la imagen. En rojo *Lsqcurvefit* y en azul *milevmar3*

Capítulo 5. Conclusiones y líneas futuras

Se ha conseguido cumplir con el objetivo principal, la optimización del tiempo de ejecución del ajuste de curva, parcialmente. Con imágenes de pequeño tamaño, el tiempo de ejecución de *milevmar3* es el mínimo que se podría esperar utilizando archivos MEX, según la documentación de MatLab, que son tiempos acortados nueve veces. Sin embargo, con imágenes grandes *milevmar3* se vuelve muy ineficiente, siendo más lento que *Lsqcurvefit*.

El objetivo secundario de lograr el mínimo error de cálculo posible respecto a *Lsqcurvefit* se ha alcanzado en mayor medida: Con imágenes medianas y pequeñas el error es despreciable, mientras que con imágenes de gran tamaño se mantiene dentro de un margen aceptable.

Una posible línea futura sería ampliar el rango de funcionamiento de la función *milevmar3*, haciendo que admitiese mejor, imágenes de mayor tamaño. Posiblemente el límite práctico al que se podría llevar la función sería similar al de *Lsqcurvefit*, no mayor, debido a que en ese punto las limitaciones son del hardware y del propio algoritmo.

Otra línea futura sería tratar de averiguar qué ocurre para que el rendimiento de *milevmar3* sea tan decreciente con el tamaño de la imagen.

El código milevmar3 también podría mejorarse revisando la forma en la que se asigna valores al factor de amortiguación λ, para lograr que sea un algoritmo menos sensible a malas aproximaciones iniciales, así como que sea capaz de resolver eficientemente un margen más amplio de problemas.

Por último, existe otra posibilidad para resolver los problemas a los que se enfrentó este proyecto, consistente en utilizar *NAG Toolbox de MatLab*, utilizando el lenguaje Fortran para realizar una versión de *Lsqcurvefit* en Fortran, ejecutable desde MatLab de forma similar como se hizo con el archivo MEX de *milevmar3* [7].

Bibliografía.

[1] Clinic Cloud (noviembre 2018-abril 2019). Clinic Cloud. Recuperado de https://clinic-cloud.com/blog/dicom/

[2] DICOM (noviembre 2018-abril 2019). DICOM Standard. Recuperado de https://www.dicomstandard.org/

[3] José Millet Roig, Antonio Cebrián Ferriols y María Guillem Sánchez. "Intrumentación biomética – Tema 9: Resonancia magnética". Universitat Politècnica de València. Febrero 2017.

[4] Hamza Alizai & Ali Guermazi. "Quantitative MRI on cartilague – A focus on T2 mapping". Aspetar – Orthopedic and Sports Medicine Hospital. Marzo 2017.

[5] Mathworks (noviembre 2018-abril 2019). MATLAB Documentation. Recuperado de https://es.mathworks.com/help/optim/ug/Lsqcurvefit.html?searchHighlight=Lsqcurvefit&s_tid=doc_srchtitle

[6] Wikipedia (diciembre 2018-marzo 2019). Algoritmo de Levenberg-Marquardt. Recuperado de https://es.wikipedia.org/wiki/Algoritmo_de_Levenberg-Marquardt

[7] Croucher, Mike (mayo 2019). High Performance Computing (HPC) On premise or cloud? The cost cuestion. Recuperado de http://www.walkingrandomly.com/?p=3535

[8] Mathworks (noviembre 2018-abril 2019). MATLAB Answers – MATLAB Central. Recuperado de https://es.mathworks.com/MatLabcentral/answers/24907-mex?s_tid=srchtitle

[9] Curry T, Dowdey J and Murry R. Ed Lea & Febiger. "Christensen's introduction to the physics of diagnostic radiology" Philadelphia: Lea and Febiger, 1984. | 3rd. ed

[10] Mathworks (noviembre 2018-abril 2019). MATLAB Documentation. Recuperado de https://es.mathworks.com/help/MatLab/call-mex-file-functions.html?searchHighlight=mex&s_tid=doc_srchtitle

[11] Mathworks (noviembre 2018-abril 2019). MATLAB Documentation. Recuperado de https://es.mathworks.com/help/MatLab/ref/mex.getcompilerconfigurations.html?searchHighlight=mex&s_tid=doc_srchtitle

[12] Alicia Cordero Barbero, José Luis Hueso Pagoaga, Juan Ramón Torregrosa Sánchez. "Cálculo numérico – Teoría y problemas". Tema 1- Soluciones numéricas a sistemas de ecuaciones no lineales

[13] Academia Europea de Pacientes (junio 2019). Biomarcadores - EUPATI. Recuperado de https://www.eupati.eu/es/medicina-genomica-personalizada/biomarcadores

[14] Mathworks (noviembre 2018-abril 2019). MATLAB Documentation. Recuperado de https://es.mathworks.com/help/MatLab/ref/mex.html?searchHighlight=mex&s_tid=doc_srchtitle

[15] Mathworks (noviembre 2018-abril 2019). MATLAB Documentation. Recuperado de https://es.mathworks.com/help/optim/ug/fsolve.html?searchHighlight=fsolve&s_tid=doc_srchtitle

Anexos.

Código final de milevmar3

```
function [sol,iter] = milevmar3(x0,xdata,ydata)
%x es el escalar del que se parte para calcular la solución, que luego será
%un vector.
%xdata son los datos de entrada
%ydata son los datos de salida (que en GIBI_T2 se llama "s" a veces)



maxiter=1000; % Arbitrario
tol=1e-4; % Tolerancia
lambdainicial=tol;
margendefallosS=20 ; % Es un parámetro que controla cuánta paciencia tiene el algoritmo antes
de decidir
% que está yendo en la dirección incorrecta, dejando de buscar un mínimo en
% esa dirección para volver a la mejor aproximación que tenía anterior (pero no exactamente en
ese resultado,
% sino en un punto al azar cercano... ).
multiplicadormargendefallos=0; % Este parámetro controla cuánto se desvía al no encontrar un
mejor mínimo.
% Con 0 no tiene efecto. Si es positivo aumenta la desviación y si es negativo
% la disminuye. Valores típidos de 0 a 0.2.
multiplicadordevelocidaddelambda=100;
%si aumenta, aumenta la velocidad a la que aumenta lambda cuando va rápido
% también aumenta el valor máximo de lambda
%Con 1 no tiene efecto

tiempobucle=zeros(1,maxiter);
tiemporesol=zeros(1,maxiter);


velocidad=zeros(1,maxiter-1);  % vamos a ir guardando la velocidad a la que aumenta lambda
% Serán maxiter-1 valores, que si no al final nos sobra 1...

%xdata e ydata deben tener el mismo tamaño, obviamente... De eso que se
%preocupe quien llama a la función, que para eso son los datos de
%entrada...

% Tenemos que recortar la imagen para que sea cuadrada... Si no, da
% problemas por tener matrices no cuadradas a la hora de usar el método...

tam=size(xdata);

if tam(1)>tam(2)
   limite=tam(2);
else
```

```
   limite=tam(1);
end

xdata=xdata(1:limite,1:limite,:);
ydata=ydata(1:limite,1:limite,:);

incrbucle=tol+1; % Hay que iniciarlizarlo... Para MatLab iba bien sin hacerlo, pero para C sí que hace falta.
aux3=0; % Hay que iniciarlizarlo... Para MatLab iba bien sin hacerlo, pero para C sí que hace falta.
delta=zeros([tam]); % Hay que iniciarlizarlo... Para MatLab iba bien sin hacerlo, pero para C sí que hace falta.
contadordefallosS=0;

% Queremos que S ( suma de los cuadrados de (ydata-modelT2(x,xdata,ydata))) sea mínima
% Tenemos que buscar qué valor de x hace mínima S.
% Tenemos x0 como aproximación inicial
x=x0;
modelT2=ydata(1)*exp(-xdata*x*1e-3);
J=-xdata.*modelT2.*1e-3;

JJ=zeros([tam]); % Hay que iniciarlizarlo... Para MatLab iba bien sin hacerlo, pero para C sí que hace falta.

% Hacemos el equivalente a MatLab a hacer J.*J' pero en C.
Jt=permute(J,[2,1,3]);
for i=1:tam(1)
   for j=1:tam(2)
      for k=1:tam(3)
         JJ(i,j,k)=J(i,j,k)*Jt(j,i,k);
      end
   end
end



D=zeros([tam]);
for i=1:tam(1)
   for j=1:tam (2)
      for k=1:tam(3)
         if i==j
            D(i,j,k)=1;
         end
      end
   end
end

for i=1:limite
   for k=1:tam(3)
      D(i,i,k)=JJ(i,i,k);
   end
```

```
end

tolim=limite*limite*tol;
S=(tolim*64)*ones(1,maxiter-1); % Serán maxiter-1 valores, que si no al final nos sobra 1...
iter=1;
unos=ones(size(xdata));
matrizx=zeros(1,maxiter);
mejorX=0;
mejorXdetodos=0;
mejorS=0;
mejorSdetodos=0;
lambda=lambdainicial;

% Tenemos que considerar el caso de que la aproximación inicial sea muy
% buena... Si no lo consideramos, al dar una buena aproximación los
% resultados empeoran.
  r=ydata(1).*exp(-xdata.*x.*1e-3); % Evaluamos la función con el x actual

S(iter)=0;
for i=1:tam(1)
  for j=1:tam (2)
    for k=1:3
        S(iter)=S(iter)+abs((ydata(i,j,k)-r(i,j,k)))^2;
    end
  end
end
S(iter)=S(iter).^(1/2);
mejorX=x;
mejorXdetodos=x;
mejorS=S(iter);
mejorSdetodos=S(iter);
matrizx(iter)=x;

listalambda=zeros(1,maxiter);
listalambda(1)=lambdainicial;

iter=2; % Para contar este primer cálculo y que quede reflejada la aproximación inicial.
errorabsoluto=mean(mean(mean(ydata-r)));
%---------------------------------------
%-- BUCLE PRINCIPAL -------------------
% while iter<maxiter && mejorSdetodos>tol
while iter<maxiter && abs(errorabsoluto)>tol/100
  %tiniciobucle=tic;
  r=ydata(1).*exp(-xdata.*x.*1e-3); % Evaluamos la función con el x actual
  % Obtenemos delta resolviendo (J'.*J+lambda.*D).*delta=J'.*(ydata-r)
  % Hacemos el equivalente a MatLab a hacer J.*J' pero en C.
  aux3=ydata-r;
  for i=1:tam(1)
    for j=1:tam(2)
      for k=1:tam(3)
        JJ(i,j,k)=J(i,j,k)*aux3(i,j,k);
```

```
            end
        end
    end
    %delta=(JJ+lambda.*D)\(aux2); % Resolver tridimensionalmente está
    %complicado, así que lo vamos a separar por componentes...
    %tinicioresol=tic;
    delta(:,:,1)=(JJ(:,:,1)+lambda.*D(:,:,1))\(JJ(:,:,1));
    delta(:,:,2)=(JJ(:,:,2)+lambda.*D(:,:,2))\(JJ(:,:,2));
    delta(:,:,3)=(JJ(:,:,3)+lambda.*D(:,:,3))\(JJ(:,:,3));
    %tiemporesol(iter)=toc(tinicioresol);

    % Actualizamos la aproximación
    mediadelta=mean(mean(mean(delta)));
    if errorabsoluto<0
        x=x+lambda*mediadelta;
    else
        x=x-lambda*mediadelta;
    end



S(iter)=0;
for i=1:tam(1)
    for j=1:tam (2)
        for k=1:3
                S(iter)=S(iter)+abs((ydata(i,j,k)-r(i,j,k)))^2;
        end
    end
end
S(iter)=S(iter).^(1/2);

    if S(iter)<mejorS
        mejorX=x;
        mejorS=S(iter);
        contadordefallosS=0;
    else
        contadordefallosS=contadordefallosS+1;
        if contadordefallosS>=(margendefallosS*(1+multiplicadormargendefallos))
            contadordefallosS=0;
            x=mejorXdetodos*((0.4+multiplicadormargendefallos)*rand(1)+0.8-
multiplicadormargendefallos);
        end
    end

    if S(iter)<mejorSdetodos
        mejorXdetodos=x;
        mejorSdetodos=S(iter);
    end

    if errorabsoluto<=(tol*100) % Valor arbitrario
        if errorabsoluto<=(tol*50) % Valor arbitrario
```

```
            if lambda>(lambdainicial*5)
                lambda=lambdainicial;
            end
            lambda=lambda*1.01;
            velocidad(iter)=-1;
        else
            if lambda>(lambdainicial*10)
                lambda=lambdainicial;
            end
            lambda=lambda*1.1;
            velocidad(iter)=0;
        end
    else
        lambda=lambda*10*multiplicadordevelocidaddelambda;
        velocidad(iter)=1;
    end
    if lambda>(lambdainicial/tol*multiplicadordevelocidaddelambda)     % Tenemos que reiniciar
el valor de lambda un poco si crece demasiado...
        lambda=lambdainicial;
    end

    matrizx(iter)=x;
    errorabsoluto=mean(mean(mean(ydata-r)));
    listalambda(iter)=lambda;
    %tiempobucle(iter)=toc(tiniciobucle);
    iter=iter+1;

end % del while
%sol=matrizx;
sol=mejorX;

subplot(4,1,1)
plot(matrizx)
title('Valores de x')
subplot(4,1,2)
plot(S)
title('Error cuadrático')
subplot(4,1,3)
plot(velocidad)
title('Velocidad de cambio de lambda')
subplot(4,1,4)
plot(listalambda)
title('Valores de lambda')
listalambda;
%tiempobucletotal=sum(tiempobucle)
%tiemporesoltotal=sum(tiemporesol)
%  mejorX
%  mejorS
%  mejorI
% matrizx
```

```
if iter==maxiter
    disp('Insuficientes iteraciones')
end
% ------------------------------------------
% ------------------------------------------

end % de la función
```